StallFreeSeek: Exploiting Better Network Conditions to Improve Video Seek Experience

JIAYU ZHU, University of Illinois, Urbana-Champaign, USA
BO-RONG CHEN, University of Illinois, Urbana-Champaign, USA
JINHUI SONG, University of Illinois, Urbana-Champaign, USA
YIFANG ZHANG, University of Illinois, Urbana-Champaign, USA
YIH-CHUN HU, University of Illinois, Urbana-Champaign, USA
TIANRUI WANG, University of Michigan, Ann Arbor, USA
SIBO ZHANG, University of Washington, USA

Streaming video dominates traffic on today's Internet. For many content types, users do not watch the entire video, but rather skip around within the video. When a user skips to a time not yet buffered by the client software, current systems stall the video playback while fetching the relevant chunk. Meanwhile, end-user bandwidth continues to increase with increased deployment of fiber-to-the-home and 5G services, as CDN appliances inside of ISPs drive down round-trip time (RTT).

This paper introduces StallFreeSeek (SFS), which can provide nearly stall free seek for users by proactively prefetching the latency-sensitive portion of future video chunks without increasing buffer waste. SFS carefully rechunks the video such that the bandwidth overhead for prefetching is small and the buffer can be refilled after user seek without stalling video playback. We evaluated SFS using various video genres, different models of user seeks using over fifty thousand simulated video sessions and nearly two thousand real-world video sessions. In these evaluations, SFS consistently outperforms Dash.js in QoE, stall time, and buffer waste. For example, in our ABR experiments on real-world network performance traces, SFS improved QoE by 25 percentage points, reduced seek-related stall times by 79%, while reducing buffer waste by 44%.

CCS Concepts: • **Information systems** → **Multimedia streaming**; • **Networks** → *Network performance analysis*; • **Human-centered computing** → *Empirical studies in HCI*.

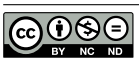
Additional Key Words and Phrases: Video streaming, Quality of Experience (QoE), Stall reduction, User study, Video chunking

ACM Reference Format:
Jiayu Zhu, Bo-Rong Chen, Jinhui Song, Yifang Zhang, Yih-Chun Hu, Tianrui Wang, and Sibo Zhang. 2026. StallFreeSeek: Exploiting Better Network Conditions to Improve Video Seek Experience. *Proc. ACM Netw.* 4, CoNEXT1, Article 8 (March 2026), 25 pages. https://doi.org/10.1145/3786291

1 Introduction

Online video streaming services account for 65% of global network traffic [32]. To improve streaming quality across network conditions, platforms and research communities have developed Quality of Experience (QoE) metrics and Adaptive BitRate streaming (ABR) techniques [1, 6, 18, 26, 36–38, 42].

Authors' Contact Information: Jiayu Zhu, University of Illinois, Urbana-Champaign, Urbana, IL, USA, jiayuz6@illinois.edu; Bo-Rong Chen, University of Illinois, Urbana-Champaign, Urbana, IL, USA, borongc2@illinois.edu; Jinhui Song, University of Illinois, Urbana-Champaign, Urbana, IL, USA, jinhuis2@illinois.edu; Yifang Zhang, University of Illinois, Urbana-Champaign, Urbana, IL, USA, zhang303@illinois.edu; Yih-Chun Hu, University of Illinois, Urbana-Champaign, Urbana, IL, USA, yihchun@illinois.edu; Tianrui Wang, University of Michigan, Ann Arbor, Ann Arbor, MI, USA, tianruiw@umich.edu; Sibo Zhang, University of Washington, Seattle, WA, USA, sz330@uw.edu.



ACM 2834-5509/2026/3-ART8
https://doi.org/10.1145/3786291

Traditional ABR strategies typically maintain a long linear buffer [36], which is ideal when users watch videos from start to finish without seeking forward or backward. This linear buffer approach can lead to significant buffer waste, particularly when user behavior includes many seeks. More critically, it results in inefficient bandwidth use: for users with good network performance, the buffer could be more strategically allocated to enhance experience during seeks, rather than merely extending the linear buffer as a hedge against an unlikely network performance drop-off.

Network connections also continue to get faster, both in bandwidth and in latency. Increased deployment of broadband, fiber [13] and 5G [7] continues to increase last-mile bandwidth [30], while the expansions of Content Distribution Networks, Content Provider Appliances [29], and last-mile optimization [39] drive down latency.

Motivated by improving network conditions and this limitation of existing ABRs, we propose a new streaming system, StallFreeSeek (SFS), designed to improve user experience when seeking. SFS introduces a novel video chunking method that reduces buffer stalling during seeks. Additionally, it incorporates a chunk prefetching strategy that aims to eliminate all seek-related stalls. With YouTube alone serving over one billion hours of video per day since 2017 [16] and one billion hours of video per day to smart TVs alone since 2024 [40], even small improvements to seek-related stalls could save tens of millions of hours *each day*. (In our User Study (§5.7), excluding users who watch no video, stall time totaled 9.8% of watch time in Dash.js but only 2.8% of watch time with our SFS system; even if most users have worse network performance than in our user studies, and if most videos experience fewer seeks, reducing stall time by 1% of watch time would save over ten million hours per day.) This system is built on Dash.js [9], and operates alongside traditional ABR algorithms, while providing better QoE to users with sufficiently good network condition.

Contributions. Our contributions are:

- SFS chunking, which reduces startup delays and minimizes stall times for a given video bitrate, as compared to traditional MPEG DASH [36] (§3.1).
- Our SFS system, which integrates into existing ABR algorithms, provides an enhanced delivery option without compromising the performance of other bitrates (Figure 3).
- With sufficient network quality, our SFS system nearly eliminates seek-related stalls and increases QoE (e.g., Figures 5, 7, 9, 11, 12a).
- Our SFS system decreases buffer waste by using a shorter linear buffer when network conditions allow (e.g., Figures 7c, 12b).

Ethical Considerations. Our IRB-approved user study uses informed consent, does not personally identify users, and collects no more than industry-standard telemetry about video playback (though our reporting interval is more frequent). The only data we log that is likely to be privacy-sensitive are IP addresses, which we purge within 3 months of collection. We allow users to opt-out of the study and view the video of their choice on a platform without any telemetry. Before conducting the user study, we had a preliminary version of the results presented in §5.4, from which we reasonably anticipated (and this paper shows) that SFS results in better user experience than Dash.js delivery. These safeguards ensure that there should be minimal risk of harm to participants.

2 Motivation

We begin this section with a thought experiment: *what if a user has infinite bandwidth and a perfect transport protocol, but non-zero latency?* When she watches the video from beginning to the end, she will experience a single 1 RTT stall when she loads the video. However, each time she seeks outside the buffered video, she will experience another 1 RTT stall while the requested chunk is fetched (even with infinite bandwidth, modern video streaming systems do not buffer the entire video; for example, YouTube limits buffering to 60–80 s [23] beyond the current playback time).

A naïve solution to this user's problem is to simply buffer the entire video, ensuring that every seek is within the buffered video. However, this solution wastes significant amounts of bandwidth whenever the user does not watch the entire video. For example, based on data collected by Chen et al. [4, 5], a large sample of videos showed a mean watch percentage of 23.69%. That is, in this dataset, buffering the entire video results in over four frames buffered for every one frame watched. To reduce the waste while still eliminating stall times, we explore alternative chunking methodologies.

2.1 A Tale of Two Parts

To allow seeking within a video without buffering the entire video, we observe that for a given user's bandwidth and round trip time, each video chunk has a *latency-sensitive part* and a *bandwidth-sensitive part*. The latency-sensitive part is the smallest part at the beginning of the chunk that enables stall-free download of the rest of the chunk. Thus, a user that downloads the latency-sensitive part of a chunk, then seeks to the beginning of that chunk can use the playback duration of the latency-sensitive part to complete the download of the remainder of the chunk. Because the latency-sensitive part is defined to be the *smallest* part of the chunk that achieves this property, buffering less would force a stall some time during this chunk, and buffering more is unnecessary to achieve the stall-free property. With infinite bandwidth and perfect transport, the latency-sensitive part is the first RTT of each chunk; with finite bandwidth, the latency-sensitive is larger.

To solve the problem of seek stalling, instead of storing the entire chunk, we need only buffer the latency-sensitive portion, which could be as small as the first frame of each chunk, but in practice is several frames long. Then, when the user seeks to a chunk that is not in her buffer, she can begin playback directly from the keyframe, while simultaneously downloading the rest of the chunk. To explain this division in greater detail, we begin by explaining the contents of a video chunk.

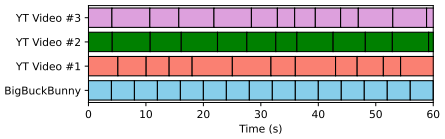
2.2 Video Chunk Structures and Key Frame Placement

Modern video delivery systems divide an encoded video into chunks. These chunks can be kept as individual files, or can be fetched from one large file using HTTP's range request. Because each chunk starts with a key frame, a chunk can be played independently, without reference to previous chunks other than the init chunk, which stores metadata for decoding. (Non-key frames are encoded as deltas from other frames, so they cannot be played without previous frames.) The way a video is divided into chunks depends on the streaming system— chunks can be fixed length or variable length, depending on how the video encoder generates key frames. Figure 1a illustrates some key frame strategies for Big Buck Bunny [3] and the YouTube videos with the lexicographically smallest unique IDs that we use in our experiments (§5.5), which we fetch with yt-dlp [44].

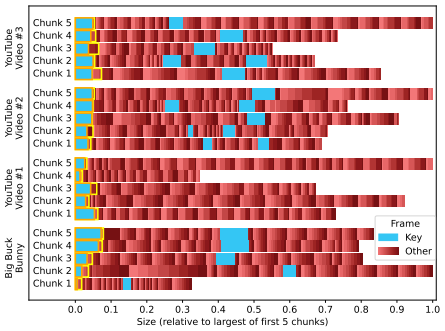
To show the structure of encoded video chunks, we take the videos shown in Figure 1a and illustrate their first five chunks (because the first chunk is often not representative of the video as a whole). Each chunk represents 4–7 seconds of video at 25–30 fps. Figure 1b illustrates the size of each frame of each of these chunks, though smaller frames may be too small to individually distinguish. Each chunk starts with a key frame, allowing it to be played without video content from previous chunks.

2.3 Calculating the Latency-Sensitive Part

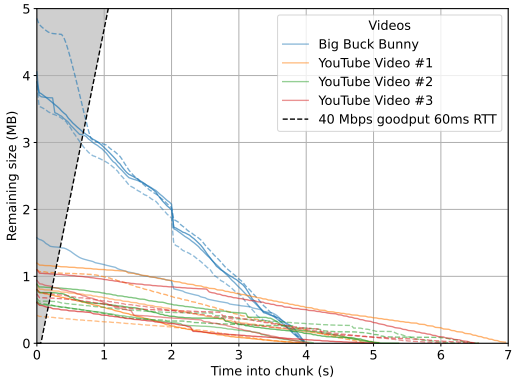
To illustrate the latency-sensitive part of a chunk in more detail, we take our infinite-bandwidth perfect-transport user and assign her an RTT of 20 ms (or any number less than 33 ms, the duration of a single frame). Her latency-sensitive part is the first frame. Figure 1b shows the size of the first frame (highlighted with an orange box) relative to all other frames, thus displaying the relative sizes of the latency-sensitive part of each chunk (the first frame) and the bandwidth-sensitive part (the rest of the frame).



(a) Examples of chunking strategies for the first minute of variously encoded videos.



(b) Relative sizes of video-on-demand chunks. Each chunk is scaled relative to the largest displayed chunk from that video to accommodate videos of different bitrates.



(c) Example of splitting latency- and bandwidth-critical sections. Each decreasing line reflects the video remaining in the corresponding chunk at that playback time. Each of four videos is represented by one color, and the line style and transparencies within that color represent chunks within that video. The shaded region is the latency-critical region; if we buffer the shaded area of each chunk, the remaining frames can be downloaded while that shaded area is played back.

Fig. 1. Illustrations of chunking behavior of some videos from our evaluations.

More generally, Figure 1c illustrates the first 5 chunks from 4 different videos. Each colored line reflects the video content remaining (in bytes) in the corresponding chunk for each playback time within that chunk. The color reflects the video and the line style and transparencies reflect chunks within that video. The dashed black line indicates the amount that can be fetched within a given amount of time over a link with 40 Mbps goodput and 60 ms RTT, ignoring transport-layer effects. (We chose 40 Mbps and 60 ms to illustrate a hypothetical link, showing how to read this graph.) The point at which this dashed line crosses each chunk line reflects the end of the latency-sensitive period for that chunk at the illustrated level of network performance. In other words, the shaded gray area reflects the latency-sensitive region of each chunk, and the area with the white background reflects the bandwidth-sensitive region.

2.4 A Simple Buffering Strategy

Once we determine the latency-sensitive and bandwidth-sensitive portions of each chunk, we divide that chunk into two *subchunks*; for example, chunk 1 would be subdivided into subchunk 1.0 and 1.1, where the first frame of chunk 1 becomes subchunk 1.0, and the remaining frames become subchunk 1.1. *We do not perform any re-encoding*, so subchunk 1.1 might not be able to be played back without subchunk 1.0.

Next, we modify the buffering strategy on our client (we use Dash.js) to download a limited-size linear buffer, then use any additional bandwidth to download the one-frame ".0" subchunks. These small subchunks allow for the stall-free playback of the first frame of each original chunk.

Finally, we modify the seek behavior of the client. Because it is somewhat improbable that a user seeks exactly to the beginning of a chunk, we instead rewrite seek times to correspond to the beginning of the nearest chunk. This rewriting ensures that the .1 subchunk is never requested without the .0 subchunk. Though this rewriting slightly reduces seek accuracy, traditional seek

bars already have limited resolution due both to screen size and the user interface; for example, MacKenzie et al. [25] show that mouse movements have mean inaccuracy of 2.5 pixels. Commonly used keyboard shortcuts also have limited accuracy, jumping 10 seconds forward or backward. Though modern seek bar innovations allow for more accurate seeking, our approach chooses a point in the design space where seeks can be stall-free *in exchange for* accuracy in seeking. §3.3 describes implementation details of our seek time rewriting.

2.5 More than Two Subchunks

In the real world, users have limited bandwidth and imperfect transport; thus, instead of dividing each chunk into two subchunks as described above, our algorithm further subdivides this segment into *multiple subchunks*. As in the previous section, the first subchunk is used to hide the latency of the request for the second subchunk, but also needs to be large enough to cover the *transfer time* of the second subchunk. The second subchunk, which is downloaded while the first subchunk is played back, needs to be large enough to hide the latency and transfer time for the third subchunk. In general, the i-th subchunk is sized to allow the download of the $i + 1$-st subchunk, as illustrated in Figure 2. With a sufficiently high bandwidth and low RTT, the first subchunk can potentially be very small, e.g., on the order of 150 ms. To illustrate the size of these first subchunks, Figure 1b highlights the first 5 frames ($166\frac{2}{3}$ ms) with a yellow border. These shorter (in duration) subchunks can then be pre-buffered during the early stages of the user's video playback. Then, when the user skips to a new playback time, our algorithm immediately plays back the small, pre-buffered subchunk while fetching each remaining subchunk in turn, allowing seamless jumps without stalling playback, contrary to previous schemes which would need to load an entire segment prior to restarting playback.

3 StallFreeSeek (SFS)

StallFreeSeek (SFS) builds the multiple subchunks as described above (§2.5), starting from a video that has already been divided into segments, each of which can be decoded without reference to other segments, as illustrated in Figures 1a and 1b. SFS divides each chunk into a small latency-sensitive subchunk that is prefetched, and a series of additional subchunks which are fetched during the playback of the previous subchunk. §3.1 describes the re-chunking algorithm.

As with our two-subchunk strategy, jumps need to be rewritten to the beginning of each original chunk, because these subchunks may not begin with key frames and thus cannot be played independently. §3.3 describes implementation details of our seek time rewriting.

SFS tries to prefetch the first (latency-sensitive) subchunk (which we call the *SFS chunk*) of each original chunk before the user jumps to that location, so that it can be played back while the next subchunk is downloaded. In videos around 15 minutes long or less, prefetching *all* SFS chunks is a viable strategy. For longer videos, buffering SFS chunks densely across the video trades off seek precision and buffer waste; we cover this in Appendix A.

Because not all users have sufficient network quality to support SFS, we dynamically select whether or not to enable SFS through the *Adaptive BitRate* (ABR) mechanism. In many streaming services, each video is encoded at several bitrates (usually by choosing a lower resolution to achieve a lower bitrate), and the ABR mechanism is responsible for choosing the bitrate that best balances the user's network condition and playback quality. We expand the ABR mechanism to decide when to enable SFS, as described in §4.

3.1 SFS chunking

To achieve seamless playback with minimal network requirements and reduced server overhead (as measured by buffer waste, the portion of buffered video that is not played by the client), we introduce

a novel chunking method called *StallFreeSeek (SFS) chunking*. This method divides traditional video chunks into two types of subchunks: **SFS chunks** and **ramp chunks**.

- **SFS Chunks:** These are small, prefetchable subchunks containing around 150 ms of video (four or more video frames at 24–30fps), designed for quick download. An SFS chunk must be playable without any previous or subsequent chunk; specifically, it must start with a key frame. If the client downloads an SFS chunk but the user skips that part of the video, the smaller size of the SFS chunk helps reduce the buffer waste incurred by SFS. When the user skips to an SFS chunk that is already downloaded, its playback duration provides sufficient time to start downloading the subsequent ramp chunk.
- **Ramp Chunks:** These chunks are made during the chunking process based on a target bandwidth and RTT, as illustrated in Figure 2. Frames are added to each ramp chunk, in order, up to a size budget, which is calculated to ensure that each ramp chunk can be downloaded while the previous chunk is being played.

One major distinction between ramp chunks and MPEG DASH [36] chunks is that ramp chunks are not intended to be decoded by themselves, but are allowed to depend on previous chunks, up to and including the preceding SFS chunk. Thus, ramp chunks need not begin with a keyframe. To prevent playback from beginning inside a ramp chunk, we rewrite seek times, using the methodology described in §3.3.

We now describe in detail the algorithm for rechunking, which is given as Algorithm 1. The algorithm takes as input the RTT r and a maximum target bandwidth $B_{\max}$ (we use 5 times the average encoding bandwidth, chosen to be commonly achievable in modern networks) and a target RTT r. First, we describe the core part of the algorithm (lines 6–16), which starts by making an SFS chunk that is 4 frames long (line 6) (the initial value of `framesz` for 24–30 fps sources) and tries to make ramp chunks given a bandwidth limit B. For each ramp chunk, we calculate a *size budget* $B(t-r)$ (line 11), where t is the duration of the previous subchunk (as updated in lines 8 and 16). We can then populate the next ramp chunk using frames starting from the frame following the previous subchunk, adding each contiguous frame up to the size budget calculated above (lines 12–15). We then repeat the process to generate up to 6 ramp chunks (the limit on number of ramp chunks is a tradeoff between MPD size and the ability to accommodate lower-bandwidth users; empirically, we observed diminishing returns around 6 ramp chunks), until the original chunk has been fully rechunked into the SFS chunk and the ramp chunks.

The lines surrounding the core portion of the algorithm implement a binary search for a feasible bandwidth target— the bandwidth B used by the core part of the algorithm is a proposed bandwidth target chosen by the binary search algorithm. When B is large enough, each frame will be in the SFS chunk or the ramp chunks (line 17), so our algorithm updates B_{top} to avoid choosing higher bandwidths. When B is too small, each subchunk will be shorter (in duration) than the previous, so the size budget will also shrink subchunk-by-subchunk; in this case, the original chunk cannot be fully rechunked. If the rechunking cannot be completed within 6 ramp chunks, our algorithm updates B_{bot} to avoid choosing lower bandwidths (line 20). This binary search calculates the minimum bandwidth at which SFS can operate, reducing the importance of predicting specific network performance values.

Finally, there is a chance that $B_{\max}$ is too small (for example, when RTT r exceeds the length of the SFS chunk). In this case, we make the SFS chunk one frame longer and try again.

Figure 3 shows the architecture of the SFS system. The SFS Chunker rechunks an encoded video and outputs chunks and parameters that are used by the client (§4.2). The client consists of a bandwidth and latency estimator, which is used by the SFS success estimator, which interfaces with

for framesz *from* 4 *to* ∞ **do**
  $B_{\text{bot}} \leftarrow 0$; // Find $B \in (0, B_{\max})$
  $B_{\text{top}} \leftarrow B_{\max}$;
  while $B_{top} - B_{bot} > 1$ *bps* **do**
    $B \leftarrow (B_{\text{top}} + B_{\text{bot}})/2$;
    SFS frame $\leftarrow$ frames$[0 : \text{framesz}]$;
    Current frame number
     $f \leftarrow \text{framesz} + 1$;
    Previous chunk duration
     $t \leftarrow \text{framesz}/\text{frame\_rate}$;
    for $i \leftarrow 1$ **to** 6 **do** // Fill i^{th} ramp
      $f_{\text{old}} \leftarrow f$;
      Budget $b \leftarrow B \cdot (t - r)$;
      while $f < len(\text{frames})$ ***and***
       $\text{frame}[f].\text{size} \le b$ **do**
        Add $\text{frame}[f]$ to ramp
         chunk i;
        $b \leftarrow b - \text{frame}[f].\text{size}$;
        $f \leftarrow f + 1$;
      Previous chunk duration
       $t \leftarrow (f - f_{\text{old}})/\text{frame\_rate}$;
      if $f = len(\text{frames})$ **then**
        $B_{\text{top}} \leftarrow B$;
        break;
      if $t = 0$ *or* $i = 6$ **then**
        $B_{\text{bot}} \leftarrow B$;
        break;
  if $B_{top} \neq B_{\max}$ **then**
    break;

Algorithm 1: Turning one video chunk into its associated SFS chunk and ramp chunks.

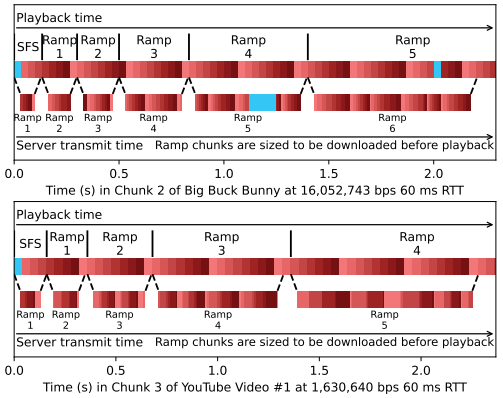


Fig. 2. SFS ramp illustration. The top blocks show video frames by playback time, the middle section shows the 1 RTT fetch delay, and the bottom blocks show chunks as they are sent over the network at the indicated bandwidth. Light blue blocks are video keyframes, red frames are other frames. Different shades of red distinguish adjacent frames; the colors repeat every 8 frames.

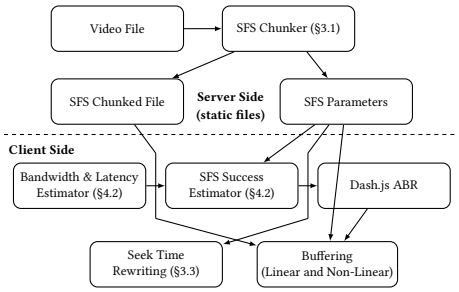
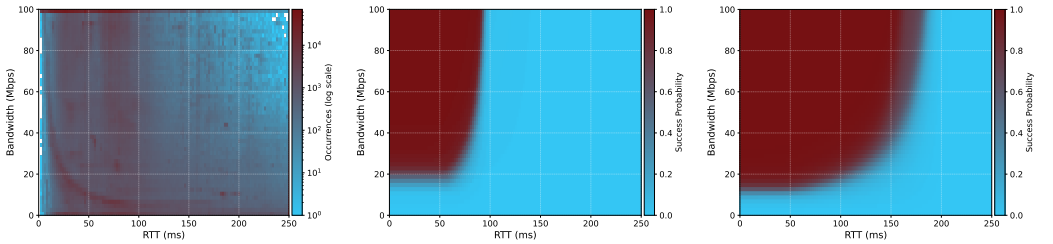


Fig. 3. SFS video streaming system diagram with server and client side components.



(a) Bandwidth/latency conditions in the Puffer dataset.
(b) Feasible regions for Big Buck Bunny.
(c) Feasible regions for YouTube experiment

Fig. 4. Puffer network conditions and SFS feasible regions.

the Dash.js ABR to determine when to switch to and from SFS. The client also handles seek time rewriting and video buffering.

3.2 Feasible Regions

To evaluate the network conditions under which SFS can operate, we considered the Big Buck Bunny video used in our Fixed Bitrate and ABR experiments (§5.3–5.4), and ran the chunking algorithm from §3.1 across a variety of possible RTTs. Specifically, we set $B_{\max}$ to 5 times the average video-encoding bitrate (for example, for a video encoded at 8 Mbps, we choose $B_{\max} = 40$ Mbps). For this evaluation, we quantize network performance to 2 Mbps and RTT to 2 ms, and evaluate network bandwidth between 0–100 Mbps and RTT between 0–250 ms.

To show the prevalence of network performance characteristics in video streaming clients, we analyzed the 5,207,574 bandwidth and RTT data points from Puffer [41] traces from July 1–7, 2024, quantized them to 2 Mbps bandwidth (rounding down) and 2 ms RTT (rounding up), and plotted their distribution in Figure 4a. These results show that many users have high-bandwidth, low-latency connections suitable for SFS.

Because it is often impractical to store a per-RTT chunking of every video at each distribution point, we ran Algorithm 1 to perform an SFS rechunking optimized for $r = 60$ ms RTT, and calculated the fraction of segments that could successfully complete a stall-free seek for each quantized bandwidth B and RTT r in our matrix. We performed this analysis both for Big Buck Bunny, as used in our fixed bitrate and ABR experiments (§5.3–5.4), the results of which are shown in Figure 4b, and for the YouTube videos used in our YouTube experiments (§5.5), the results of which are in Figure 4c. These figures show that for RTTs under 100 ms and bandwidth over 20 Mbps is generally sufficient to achieve stall-free seeks with SFS, and that videos that are encoded at lower bandwidth need less bandwidth to achieve stall-free seeks with SFS.

3.3 Rewriting Seek Times

Whenever the user seeks, SFS works only if the seek time is within one of the prefetched SFS chunks, and is most effective when the seek destination is the beginning of an SFS chunk. (Starting playback in the middle of an SFS chunk requires far higher network performance to successfully retrieve the ramp chunks.) To ensure that user seeks always start at an SFS chunk, we take control of the users' UI seek mechanisms. Whenever the user commands a seek, we take control of the progress bar [8] to rewrite the seek time. This approach avoids having to directly interface with the media playback engine. To support seek time rewriting, when chunking the video, we also collect timing information about the start of each SFS chunk and output a JavaScript array that is passed as input to the control bar. When the user seeks using the progress bar, both the seek time and the displayed time is rewritten to be the start of the nearest SFS chunk, allowing for immediate resumption of playback. We also rewrite the targets of keyboard shortcuts in this way.

4 SFS Implementation

4.1 SFS Chunking

Since existing tools such as FFmpeg [12] do not support arbitrary chunk durations, we created our own Python tools based on FFmpeg and Bento4 [24]. We first use FFmpeg's DASH encoder to encode the video into one-chunk segments of 3–5 seconds. We then use Bento4's `mp4dump` to inspect the frame information from FFmpeg DASH encode results, and use Algorithm 1 to calculate the size of each SFS chunk and ramp chunk. We then extract the frames from the segments and update the headers with new sequence numbers (`moof/mfhd`) and frame information (`moof/trun`). We also modify the MPD file generated by FFmpeg, updating the segment timeline information to reflect the additional frames created in this manner. We reuse the initialization segments from FFmpeg's DASH encoding, as there is no decoding difference between the SFS format and the traditional format. The result of this processing is an MPD file and associated chunks that can

be played just like MPD files directly generated by FFmpeg's DASH encoder. The new MPD files are not significantly larger than the original MPD file— across all 14 videos used in our YouTube experiments, the mean size increase is 18,009 bytes, negligible compared to video chunk sizes and compared to the buffer waste saved by SFS. Furthermore, the larger MPD is only fetched in high-bandwidth conditions when SFS is enabled, so low-bandwidth or high-latency users unable to benefit from SFS experience no penalty from the presence of SFS.

4.2 SFS Client

We build on Dash.js v4.7.4 [9], which is both a "prominent option for implementing production grade DASH-based applications and products, and is also widely used for academic research purposes" [35]. The ABR algorithm used by Dash.js is BOLA [38] modified with additional rules such as the abandon request rule, which abandons requests that have been pending for too long. These rules are useful in real networks which exhibit a wide range of performance. We enable SFS streaming by modifying the ABR controller and implementing a new video prefetching strategy. We continue to use MPEG DASH format for our videos, and implement SFS as a separate MPD that is swapped in by the ABR if the ABR considers the network conditions good enough for SFS. If network conditions cannot support SFS, the video plays as usual, since our goal is to build a system that is never worse than the underlying ABR.

We take the highest bitrate available, up to 1080p, and chunk it into the SFS format; this additional format is included as a virtual resolution in the regular (non-SFS) MPD. Adding this virtual resolution allows the ABR controller to propose a switch to SFS under favorable network conditions. We use Dash.js' record of recent network bandwidth and latency to calculate the average SFS success probability over time, based on the bandwidth and RTT requirements generated by the chunking. When the default ABR algorithm proposes a switch to the SFS resolution, our modifications consider the average SFS success rate to make the final switch or to reject the request. In our study, we do not attempt to generate SFS frames above 1080p because it is unclear whether 1080p without seek-related stalls results in better QoE as compared to a 4K video with seek-related stalls, and because of the significant network demand associated with using SFS with 4K video.

To ensure seamless transitions, we configure Dash.js [9] to treat the non-SFS format's highest resolution as equivalent to the SFS resolution since they share the same resolution, differing only in chunking format. This configuration prevents already-buffered non-SFS content at the highest bitrate from being deleted when using Fast Switching [37] to switch to SFS, and thus avoids fetching chunks that are already present in the media source buffer. Because the MPD change is handled within Dash.js, the browser's media player state is preserved, making the MPD switch seamless and transparent to the user.

We also change Dash.js' buffering rules when the SFS MPD is loaded. Specifically, once we reach the default stable buffer level (set at 10 seconds in our experiments), we maintain a linear buffer (that is, a buffer immediately after the current playback time) at this stable buffer level. Whenever this threshold is met, we request SFS chunks rather than extending the linear buffer. This prefetch allows the user to seek to any pre-downloaded SFS chunk without stalling. Even after all SFS chunks are downloaded, we continue to maintain this stable buffer level. We deliberately choose a stable buffer level significantly lower than the maximum size of the linear buffer in Dash.js (60 seconds), because whenever the user skips over a portion of the video, SFS will waste some buffer. We want to choose a lower linear buffer level so that when seeking, we can save some waste caused by the linear buffer to offset the waste caused by unused SFS chunks.

When using the SFS MPD, it is possible that network conditions deteriorate to the point that SFS is unlikely to be successful. In this case, the default ABR algorithm will propose a switch down from

SFS. We use the probability of SFS success, again based on the bandwidth requirements calculated by the re-chunking algorithm, to decide whether or not to perform the switch-down.

As described in §2.2 and 3.3, we also modify the progress bar to rewrite seek times to the closest preceding SFS chunk time whenever SFS is enabled. In our experiments, we use 4 second chunks, which allows for 2 second worst-case seek errors, and 1 second average-case seek error. The traditional seek bar user interface has even greater mean error; for example, in the Dash.js reference player, on a full-screen browser on a 1920x1080 monitor, the progress bar is only 436 pixels wide [8]; even with a screen-width seek bar, mouse control error of ±2.5 pixels [25] would result in a 1.56 second uncertainty region for a 10 minute video. Modern seek interfaces allow more precise seek; however, SFS provides reduced stall time. We can see SFS deployment as a *trade-off* between seek accuracy and stall time.

5 Evaluation

We conduct several experiments to evaluate the performance of SFS under different network conditions, video sources, and user behavior. We have released our full implementation pipeline on github [46] for researchers to reproduce our results and to build on our work.

5.1 Metrics

Because the goal of SFS is to reduce seek-related stall time, our primary metric is *seek-related stall time*. We also present total stall time, which is the sum of seek-related stall time and *initial stall time* (the amount of time it takes from when the page is loaded until the video begins to play). Finally, like previous work, we present a QoE score, using a definition similar to the one in SODA [6]. The QoE score consists of three components: mean utility, rebuffering ratio, and switching rate.

Mean Utility is a commonly-used logarithmic utility function that represents video quality as proportional to the logarithm of the encoding bitrate. (SODA samples once per chunk, but because SFS chunks are shorter in duration, we sample once per second to make each time segment equally important.) **Rebuffering Ratio** is the ratio of rebuffering time (stall time) to viewing duration, i.e., $\rho_{\text{rebuf}} = \frac{T_{\text{rebuf}}}{T}$. **Switching Rate** is number of bitrate switches divided by the viewing duration, i.e., $p_{\text{switch}} = \frac{N_{\text{switch}}}{T}$. We calculate switching rate on a per-second, rather than per-chunk, basis.

The QoE score is simply a linear combination of the three QoE components: $\text{QoE} = \bar{v} - \beta \cdot \rho_{\text{rebuf}} - \gamma \cdot p_{\text{switch}}$. We choose $\beta = 10$ and $\gamma = 1$. Although we create a virtual bitrate to allow dash's ABR to trigger SFS, we do not consider switching between the virtual bitrate and the highest normal bitrate as a quality switch, because the video encoding is identical.

In addition to these quality-related metrics, we also evaluate *buffer waste*, because a content provider's bandwidth usage is the sum of *viewed video*, *buffer waste*, and *protocol overhead*. Since viewed video is desirable, and protocol overhead represents a negligible number of bytes as compared to the size of high-definition video, buffer waste represents the most significant component of bandwidth usage for a given video quality and viewing duration.

To calculate buffer waste, we use the HTTP logs to determine which frames were transmitted by the server. On the client, we use the `requestVideoFrameCallback` API [17] from the `videoElement` in the HTML player to determine the frames that have been played. Where possible, we record these frame numbers directly at the client, but for the user test (§5.7), we transmit this information one time per second back to the server for logging in a database. We define *buffer waste* to be the aggregate size of frames that were transmitted and not played.

5.2 Experimental Setup

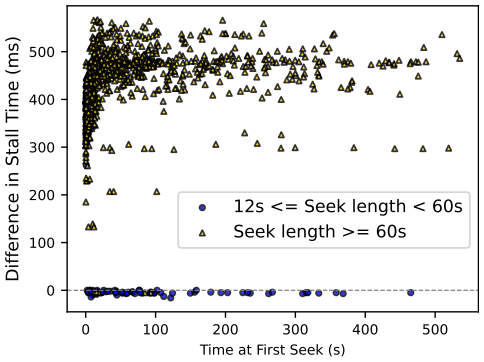
To simulate different network conditions, we use Traffic Control (`tc`) [19] to limit the bandwidth and increase the RTT of our network. Our experimental clients are connected to a server over a gigabit switch which imposes around 150 μs of round-trip latency in addition to the latency added by `tc`. The bandwidth and added RTT are sourced from the Puffer [41] dataset, specifically data from the dates of July 1–7, 2024, using only those sessions exceeding 10 minutes duration. We choose these dates because they are the first days from the second half of 2024.

Because SFS shows benefits only when a user seeks within a video, our evaluation uses traces that potentially seek within a video. We use two distinct approaches to obtain these traces. Our first approach is entirely synthetic. In our synthetic traces, playback starts at the beginning of the video. The user modeled in these traces initiates a seek action after a time drawn from an exponential distribution with a predefined rate parameter. Each seek terminates somewhere between 10 seconds following the current position in the video (calculated as the last seek destination plus the time elapsed since that seek) and the end of the video. (If this seek is not possible, as the current position is within 10 seconds of the end of the video, we do not seek, but rather play the video until the end). Once the seek is initiated (and potentially before playback resumes), we draw a new value from the exponential distribution for the next seek. Our seeks happen based on wall-clock time, not media position time, so that when we evaluate Dash.js and SFS using the same seek trace and the same network trace, both systems will experience identical network conditions— because both seeks will be initiated at identical wall-clock times, the network conditions at the time of the seek will be identical. Since we are unable to obtain sufficient traces from actual network users (with the exception of our user study, §5.7), our second approach is to use traces from vRetention [4, 5]. Though vRetention traces are synthetic, they sum to the shape of YouTube retention curves. We trigger trace-driven seeks using Selenium [27] to click on the progress bar.

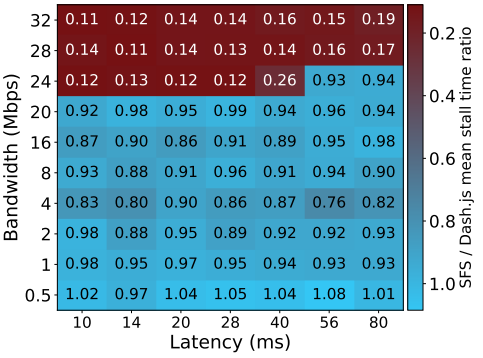
5.3 Fixed Playback Bitrate

We first evaluate whether SFS can reduce stall-related seek times under good network conditions. We use Big Buck Bunny [3] in this set of experiments, and we use Chrome DevTools on Chrome browser 135.0 to limit the network to 100 Mbps bandwidth and add 50 ms latency. We use Chrome DevTools instead of `tc` because DevTools does not allow any token-bucket-induced excursions from the desired rate, and we do not need the per-second adjustments of which `tc` is capable. We generate 50 synthetic seek traces as described in §5 for each exponential arrival rate, which we vary between 2 and 1000 seconds on a logarithmic scale. We use fixed (rather than adaptive) playback bitrates using the 8 Mbps 1080p encoding, running each synthetic trace on both the original Dash.js with the traditional MPEG DASH format and our modified Dash.js player using our SFS format and the prefetching rules described in §4.2.

To validate how SFS impacts seek latency, we compare the stall induced by the *first seek* of each run. By using the first seek, we present a worst-case for SFS, which waits until it buffers 10 seconds before beginning SFS chunk prefetch; in subsequent seeks, the SFS chunk is more likely to be available. Figure 5a plots the difference in stall time between original Dash.js and SFS against the (wall-clock) time of each seek, and distinguishes data points based on the "distance" jumped (that is, the number of seconds between the target time and the wall-clock time, which should be very close to the playback time, given the good network conditions). The groups we choose, jumps between 12 and 60 seconds and jumps exceeding 60 seconds, attempt to distinguish between jumps that likely land within Dash.js' 60 second buffer but outside SFS' 10 second buffer, and jumps that likely land outside both buffers. When jumps land outside Dash.js' buffer, SFS improves stall time by hundreds of milliseconds. When jumps land inside Dash.js' buffer, performance is approximately



(a) Differences in first-seek stall time between Dash.js and SFS.



(b) Stall time ratio in low-bandwidth runs: SFS and Dash.js.

Fig. 5. Stall time performance of SFS in fixed bitrate experiments.

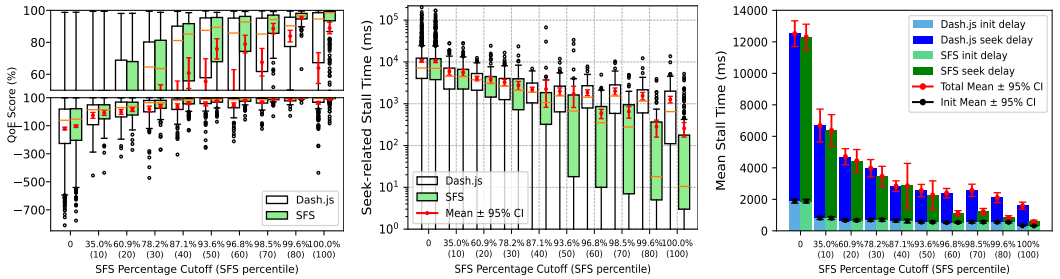
the same, with the slight difference of 5–10 ms reflecting SFS stalling while it loads the SFS chunk from a JavaScript variable into the browser's media player. For reference, previous work has shown that latencies under 10 ms are largely imperceptible in both user interface interaction [10, 20] and for audio [33]. In other words, under sufficiently good network conditions, whenever SFS is better, it is much better, and whenever SFS is worse, it is *imperceptibly* worse.

Low-Bandwidth Experiments. Because not all users have bandwidth sufficient for SFS, we use Dash.js with fixed bitrates and latencies to evaluate the performance of SFS across a variety of performance scenarios. In these experiments, we use Chrome DevTools to set bandwidth between 0.5–32 Mbps and latency between 10–80 ms. For each table entry, we perform 26 pairs of runs, one for each λ value, to evaluate a variety of user behaviors. We divide the stall time of Dash.js with SFS by the stall time of unmodified Dash.js and plot it as a heatmap in Figure 5b. These results show that at 20 Mbps and below, SFS is not activated, but the performance is not degraded (in fact, the seek time rewriting seems to slightly improve performance). These experiments reflect the feasible regions illustrated in Figure 5a, which shows that SFS cannot guarantee stall-free seeks at 20 Mbps. Thus, our algorithm does not allow shifts to SFS until just over 20 Mbps bandwidth. When we modify our algorithm parameters, we find that SFS can be safely triggered at 20 Mbps bandwidth and 10–80 ms latency with a lower switch-to-SFS threshold, but such parameter changes may impair performance when a user's bandwidth varies over time.

5.4 Adaptive BitRate

We deliver Big Buck Bunny [3] at nine video qualities from 180p through 1080p so that the Adaptive BitRate (ABR) algorithm has many choices to maximize quality for a given network condition. These experiments are designed to verify that (1) under poor network conditions, SFS is never triggered, and SFS does not negatively impact performance, and (2) when network conditions meet certain thresholds, SFS is triggered and results in significantly improved QoE.

We group Puffer [41] traces by average bandwidth, creating nine bandwidth groups: 0.5–2 Mbps, 2–5 Mbps, 5–10 Mbps, 10–15 Mbps, 15–25 Mbps, 25–50 Mbps, 50–75 Mbps, 75–100 Mbps, and 100–200 Mbps. For each bandwidth group, we randomly select 260 network traces, and 260 user click traces from the exponential distribution model, with 10 traces sampled from each λ (average time between jumps) value. We assign each network trace to a user trace, and compare Dash.js with its ABR against SFS with the ABR described in §4.2. We use `tc` to apply the network trace and Selenium to automate each seek.



(a) SFS improves QoE. (b) Seek-related stall time. (c) Total stall time.

Fig. 6. QoE and stall time in ABR experiments.

Stall Time. Because the Puffer server is at Stanford, traffic from the East Coast will experience large RTTs, meaning that high bandwidth may not correspond to being in the SFS Feasible Region (§3.2). A better metric for SFS' ability to reduce seek time is the *fraction of time that the ABR uses SFS*. We group this fraction into 11 bins: one where SFS is not used at all, and 10 that represent deciles of SFS usage, and plot the seek-related stall time for each bin in Figure 6b. We label the x-axis with the cutoff for each decile (that is, the maximum percentage of SFS that qualifies for that decile), and in parenthesis, the top percentile of the represented decile. For example, the 8th decile represents SFS usage between 98.5% and 99.6% and the corresponding bar is labeled (80), which is the top percentile in this decile. We represent two deciles together in the right bar, because significantly more than 10% of our runs that use SFS have 100% SFS usage, and we do not have a metric to decide which 100%-SFS runs should be in the 9th decile and which should be in the top decile. This graph shows that SFS never has worse seek-related stall times than Dash.js, and for higher levels of SFS usage, SFS has significantly lower seek-related stall time than Dash.js. Figure 6c uses the same grouping by SFS decile to plot the total stall time (stall-related and initial delay). This result shows that initial delay is about the same in Dash.js and SFS, because in both cases, ABR typically starts at a lower bitrate, and we only provide SFS chunks for 1080p.

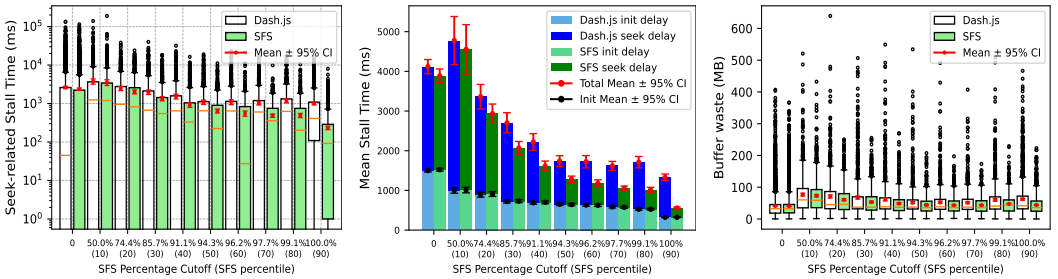
QoE. Figure 6a shows Quality of Experience plotted using the same SFS decile groups as above. When SFS is not used at all, SFS and Dash.js have similar QoE. Increasing SFS reflects better network conditions, so QoE improves for both Dash.js and SFS, but SFS improves more rapidly, showing that the improved stall times are not offset by decreases in other QoE parameters.

Buffer Waste. When SFS is not used, Dash.js and SFS have nearly identical buffer waste. Once SFS is used, SFS' average buffer waste is lower, with the difference increasing with SFS decile, and becoming statistically significant in the fourth decile.

With sufficiently good network conditions, SFS eliminates nearly all seek-related stall times, and in worse network conditions, SFS is unused and does not harm stall time, QoE, or buffer waste.

5.5 YouTube Videos

For our next set of experiments, we sample videos from YouTube to measure SFS performance across a variety of different videos. We enable SFS using the same ABR algorithm modifications described in §5.4. We obtain retention traces from vRetention [4, 5], from which we randomly choose 14 videos from seven country-category pairs chosen at random: India-Entertainment, Egypt-Music, Mexico-Music. Russia-Gaming, Russia-People/Blog, Spain-Sports, and Thailand-Entertainment. The selected videos are between 139 seconds and 1 hour long. We download each video at 1080p using yt-dlp [44], use FFmpeg to re-encode them at the bitrates recommended for YouTube ingest [43], with a keyframe every two seconds, and provide three resolutions: 480p, 720p, and 1080p. We use this chunking with Dash.js, and compare against an SFS-chunked version of the same file



(a) Seek-related stall time. (b) Initial delay is equal. (c) SFS wastes less buffer.

Fig. 7. SFS improves seek-related stall time and buffer waste in YouTube experiments.

and encoding. (As described previously, SFS chunking applies only to the highest resolution.) We perform no other post-processing to these videos, so to deploy SFS, a content provider needs only encode their videos with periodic keyframes, and rechunk them according to our SFS algorithm. These steps need only be performed once for each video.

For each video, the vRetention dataset provides 200 traces, so we choose 200 traces from each bandwidth group in the ABR experiments (§5.4), except that we eliminate 0.5–15 Mbps traces because SFS is never triggered at those bandwidths. This leaves five bandwidth groups, so for each YouTube video we choose, we perform 1000 runs for each of Dash.js and SFS.

Stall Time. Figure 7a plots the seek-related stall time in this experiment grouped by SFS decile (as in §5.4). These results show that SFS provides statistically significant improvement in seek-related stall times starting from the third decile of SFS percentage. At lower percentages of SFS usage, there is no significant difference between Dash.js and SFS, showing that deploying SFS does not increase stall times even when the network condition is insufficient for SFS.

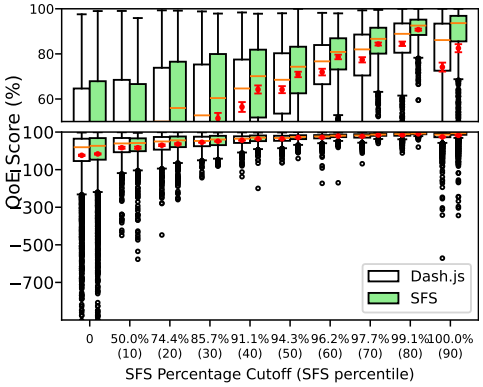
Figure 7b plots the total stall time in this experiment grouped by SFS decile. The initial delay of Dash.js and SFS (illustrated by the bottom, lighter bar) are comparable across all SFS deciles, showing that *SFS does not increase load time*. Further, the mean stall time under SFS is better than Dash.js *at every decile*, and is statistically significant by the third decile of SFS usage. The reason that SFS could potentially improve mean stall time even when SFS is not activated is that SFS rewrites seek times even when SFS is not activated. If a user attempts to seek to the last few frames of a chunk, in Dash.js, a double-stall is possible, as the first chunk finishes playback before the second chunk has finished loading. By contrast, a user seeking under SFS always seeks to the beginning of a chunk, reducing the instance of double stalls induced by the same seek.

QoE. Figure 8a plots the QoE from these experiments grouped by SFS decile. The y-axis is split to show both the entire QoE range and the range from 50–100% QoE. These results show that SFS improves the average QoE in every decile, and is statistically significant by the third decile.

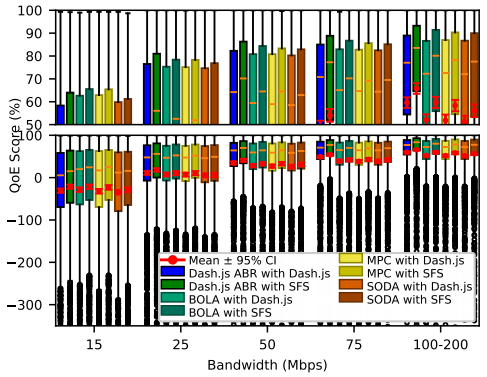
Buffer Waste. Figure 7c plots buffer waste against SFS decile. For low deciles, buffer waste is similar, since SFS is not enabled and thus the buffer strategy is the same. As SFS use increases, it reduces average buffer waste; the improvement is statistically significant by the second decile.

5.6 Additional ABR Algorithms.

To determine the compatibility of SFS with multiple ABRs, we evaluate Dash.js, BOLA [38], MPC [42], and SODA [6] ABRs with and without SFS using the same set of YouTube videos as our evaluations in § 5.5. These ABRs have different policies: BOLA is a buffer-based algorithm, MPC uses a system model and predictive control to account for bitrate switching, rebuffering time and quality, and SODA improves MPC by using a new score function, and limits it search to only

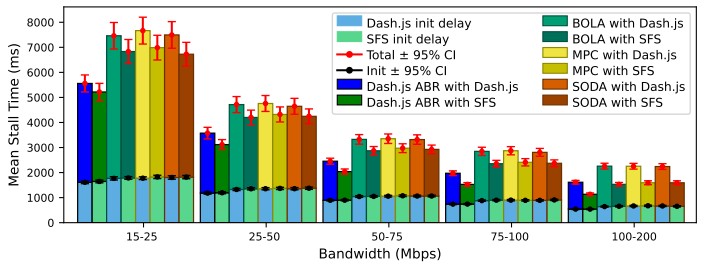


(a) SFS improves QoE. Two y-axis scales.

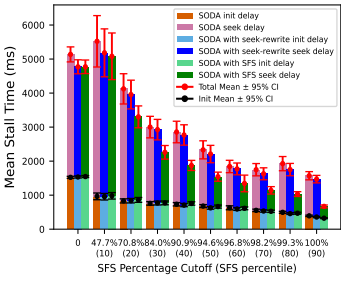


(b) SFS improves QoE with 4 different ABRs.

Fig. 8. Quality of Experience improvement in YouTube experiments, across a variety of ABRs.

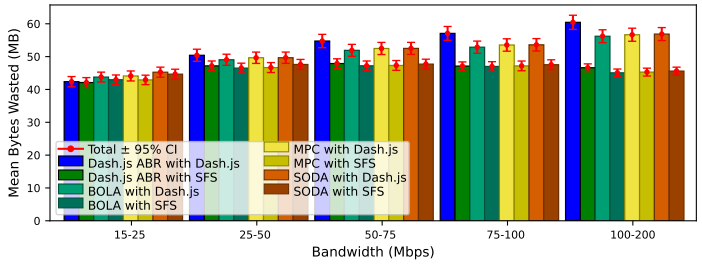


(a) SFS shows stall improvement with multiple ABR algorithms.

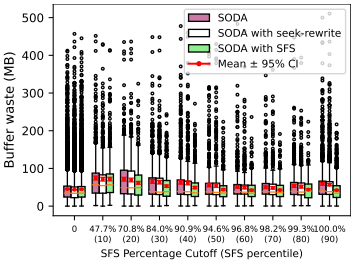


(b) Stall time of SODA with seek-rewriting vs SODA with SFS.

Fig. 9. Stall times with various ABRs



(a) SFS shows buffer waste improvement with multiple ABR algorithms.



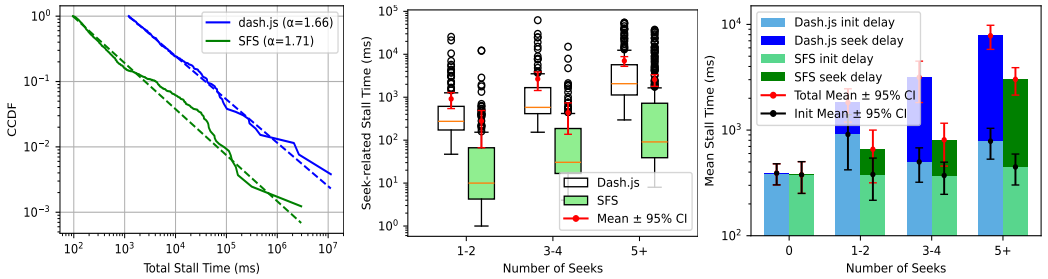
(b) Buffer waste of SODA with seek-rewriting vs SODA with SFS.

Fig. 10. Buffer waste with a variety of ABRs in the YouTube experiments.

monotonic bitrate sequences. For each of these ABRs, SFS significantly improves stall time, buffer waste, and QoE.

QoE. Figure 8b plots QoE against bandwidth for each of the four ABRs. Our results show that with sufficient bandwidth, SFS improves median QoE by 5–10% across all evaluated ABRs.

Stall time. Figure 9a plots the total stall time against bandwidth for each ABR algorithm, from which we make three observations. First, across all ABRs and bandwidth groups, SFS reduces average per-playback stall time by 341–779 ms, showing that SFS is ABR-agnostic and can improve



(a) Stall time is heavy-tailed. (b) Seek-related stall (log scale). (c) Total Delays (log scale).

Fig. 11. Stall time distribution and stall performance in the user test.

a wide variety of ABR algorithms. Second, with increasing bandwidth, total stall time decreases yet SFS stall reduction increases, demonstrating that with sufficient bandwidth, SFS is better able to reduce seek-related stalls. Finally, across different ABRs, all ABRs behave similarly except Dash.js, which performs best due to the abandon request rule, which abandons requests for chunks that have been skipped in a seek, further justifying the use of the Dash.js ABR in other evaluations.

Buffer Waste. Figure 10a shows the buffer waste for each of the 4 different ABRs. For each ABR, SFS improves buffer waste, and does so by a statistically significant amount at and above 50 Mbps.

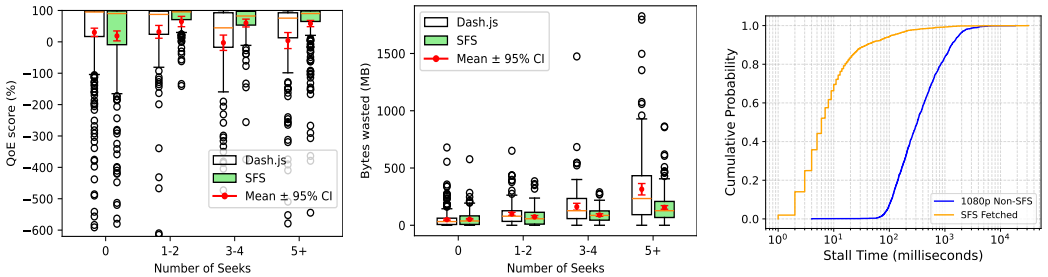
Impact of Seek-time Rewriting. Our experiments show that SFS improves SODA even when SFS is not triggered by the ABR. This is because seek-time rewriting makes better use of the downloaded chunks, decreasing stall time and buffer waste. We separately run our experiments using SODA with only the seek-time rewriting and plot stall time (Figure 9b) and buffer waste (Figure 10b). Our results show that seek-time rewriting provides some benefit, accounting for all of the benefit when SFS is not triggered, but providing only a small portition of the benefit when SFS is always triggered.

By exploring several different categories of videos, using viewing patterns derived from YouTube retention curves, these experiments show that SFS can co-exist with a variety of ABR algorithms and is beneficial across a range of video types. SFS does not degrade stall time or QoE under poor network conditions, and under better network conditions, SFS can significantly improve QoE by significantly decreasing seek-related stall times, while simultaneously reducing buffer waste.

5.7 Real-World User Test

We build a real-world user test where we serve instructional videos related to our course *Introduction to Programming*. Our protocol is approved by our institution's Institutional Review Board. We take a selection of these instructional videos, chunk them into SFS format, host them on a server, and publish them on the course website. Viewers (who we presume are mostly students in this course) who visit this server affirmatively consent to the experiment, including randomization, and have the option to opt-out of trace recording and randomization (in which case their video is delivered by Dash.js); opt-out users are not counted in our experiment. Other viewers are, on a per-playback basis, randomized into the Dash.js group and the SFS group. We collect a total of 1,908 video views, consisting of 934 in the Dash.js group and 974 in the SFS group.

Some users do not watch any video at all. This paper focuses on the sessions that play video, since our quality metrics are unimportant in sessions that don't play videos. Among the remaining 1,795 video views, 879 are in the Dash.js group and 916 are in the SFS group; the total stall times for these traces appear to follow a heavy-tailed power law distribution, as shown in Figure 11a. To improve the readability of the graphs, we take the 98% that have the lowest total stall time and plot



(a) SFS improves QoE. (b) SFS reduces buffer waste. (c) Stall time across all sessions.

Fig. 12. SFS improves QoE and reduces buffer waste in our user test.

them in this paper. (SFS is not worse in the remaining 2% of runs, but the variance makes it hard to read figures showing all runs; performance from those 2% of runs are in Appendix B.)

Of these, 1,449 video views experience one or more jumps. Some jumps happen before video replay resumes, complicating analysis of jump frequency and distance; removing playbacks that experience such jumps left us with 1,409 video views and 7,011 jumps, an average of around 4 jumps per playback. The median jump distance is 9.23 seconds, the 90th-percentile jump distance is 102.67 seconds, and about 26% of jumps go backwards in the video. The median playback interval between jumps is 2.58 seconds, with a 90th-percentile of 74.79 seconds. Thus, generally jumps are several times larger than the playback interval.

Stall Time. Figure 11b shows the seek-related stall time plotted against the number of seeks, and shows a statistically significant reduction in seek-related stall time for all seek groups. Figure 11c shows total stall time plotted against the number of seeks. The initial delay is similar across protocol (Dash.js vs. SFS) and number of seeks, but shows more variance than in previous experiments, due to the heavy-tailed nature of network performance. Average total delay is lower by a statistically significant margin with SFS than with Dash.js across all seek groups, which is intuitive given that initial delay is similar and SFS has significantly better seek-related stall time. We also measure the non-zero stall time occurring within 2 seconds of each seek for two categories of users: SFS users which have already prefetched the SFS chunk, and Dash.js users streaming at the highest video quality (1080p). The SFS users experience a median of 6 ms stall time, with 70% of users experiencing 10 ms or less, whereas Dash.js users experience a median of 320 ms stall and only 0.16% of users experiencing 10 ms or less. A CDF is shown in Figure 12c. *This plot shows the non-zero stall time occurring within 2 seconds of each seek for two categories of users: SFS users which have already prefetched the SFS chunk, and Dash.js users streaming at the highest video quality (1080p). SFS is so effective at reducing stall time that the first percentile Dash.js latency (the best 1% of latencies) falls into the 92nd percentile SFS latency (the worst 8% of latencies).* This result shows that downloading the SFS chunk significantly helps in the download of the remaining sub-chunks.

QoE. Figure 12a shows QoE plotted against the number of seeks; SFS provides better QoE by a statistically significant margin whenever there is at least one seek. The average QoE is mildly lower without seeks, since there is no improvement in stall time; however, this decrease with no seeks is much smaller than the QoE increases when seeking, which makes SFS a good tradeoff.

Buffer Waste. Figure 12b shows the buffer waste against the number of seeks. SFS reflects a slightly higher buffer waste with zero seeks, slightly lower buffer waste with 1–2 seeks, but for 3 or more seeks SFS has an average buffer waste that is statistically significantly lower than Dash.js. This demonstrates that SFS does not increase the bandwidth usage of the video server.

6 Related Work

Adaptive BitRate (ABR). There has been extensive prior work on ABR designs. Two major categories are buffer based ABR algorithms such as BOLA [38] and BBA [18], and mixed decision ABR controllers such as DYNAMIC [37], MPC [42], and HYB [1]. Additional ABR algorithms such as fast switching [37] have also been studied to improve ABR choices between when the bitrate choice is made and when the affected frame is played. SFS, building on top of Dash.js' default ABR controller [9], has the characteristics of hybrid ABR decision system, enabling prefetch within a single video. In contrast, Dashlet [22] focuses on prefetching across videos. This distinction highlights SFS's focus on improving stall times during intra-video seeking. Recent work has built ABRs using machine learning. Approaches such as Pensieve [26] purely aim to optimize bitrate selection, and do not attempt to improve seek-related QoE. Other approaches, such as HotDash [34], implement some prefetch logic that uses machine learning to prefetch parts of the video. Unlike SFS, such work focuses on hot spots rather than seamless seeking across the entire video.

Video Chunking. Previous work on video chunking [45] analyzes the relationship between chunk size variation and ABR performance. Low-latency DASH [15] uses smaller chunks, which do not all start with key frames, to reduce latency. SFS gets this benefit, but also chooses chunks of different duration and size to allow for immediate playback in many cases, thus improving seek-related stall times. Finally, low-latency DASH [14] focuses on live streaming where streamer-to-viewer latency is important, whereas SFS focuses on Video-on-Demand content.

User Behavior. Our SFS system is built on the assumption that users do not watch videos in full, and aims to mitigate the stall time and buffer waste caused by users seeking within a video. Extensive previous work validates this assumption. A study of video retention [4] shows that most users seek within a video during playback, and the amount of seeking and video retention varies by video category and country. Other work [2, 11, 21, 28] shows that rebuffering time (time spent refilling the buffer when it runs out) impacts user behavior and results in loss of user retention. View counts and user engagement is also related to seek behavior [31].

Quality of Experience. Video Quality of Experience has been studied extensively. Studies on the relative impact of QoE metrics on subjective Quality of Experience [2] show that rebuffering is very disruptive, whereas bitrate switches are less important. Other work (e.g., [11, 21]) also suggests that rebuffering reduces user engagement. Recent video delivery work, such as SODA [6], reflects these priorities by weighting rebuffering ratio much more strongly in their QoE metric as compared to bitrate switches.

7 Conclusion

This paper presents StallFreeSeek, a novel video streaming system designed to significantly reduce stall times and improve QoE scores whenever users with favorable network conditions seek within a video. SFS achieves this by combining a carefully calculated nonuniform video chunking with a video prefetching client. Our method specifically addresses stall times caused by seeking within videos, offering a seamless playback experience. Through extensive evaluations across over fifty thousand runs and nearly two thousand real-world runs, we demonstrate that our system consistently outperforms existing video delivery methods under sufficient network conditions while maintaining comparable performance in poor network conditions. For example, in our ABR experiments on real-world network performance traces, when SFS is used for the whole video, SFS improved QoE by 25 percentage points, reduced seek-related stall times by 79%, and reduces buffer waste by 44%.

References

[1] Zahaib Akhtar, Yun Seong Nam, Ramesh Govindan, Sanjay Rao, Jessica Chen, Ethan Katz-Bassett, Bruno Ribeiro, Jibin Zhan, and Hui Zhang. 2018. Oboe: auto-tuning video ABR algorithms to network conditions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, Article 3230558, 15 pages. https://doi.org/10.1145/3230543.3230558

[2] Christos George Bampis, Zhi Li, Anush Krishna Moorthy, Ioannis Katsavounidis, Anne Aaron, and Alan Conrad Bovik. 2017. Study of Temporal Effects on Subjective Video Quality of Experience. *IEEE Transactions on Image Processing* 26, 11 (2017), 5217–5231. https://doi.org/10.1109/TIP.2017.2729891

[3] Blender Foundation. 2013. Big Buck Bunny, Sunflower version. Available under Creative Commons Attribution 3.0 license. (2013). http://bbb3d.renderfarming.net/ Accessed: Jan 31, 2025.

[4] Bo-Rong Chen, Jiayu Zhu, Yanxin Jiang, and Yih-Chun Hu. 2024. vRetention: A User Viewing Dataset for Popular Video Streaming Services. In *Proceedings of the 15th ACM Multimedia Systems Conference (MMSys '24)*. Association for Computing Machinery, New York, NY, USA, 313–318. https://doi.org/10.1145/3625468.3652175

[5] Bo-Rong Chen, Jiayu Zhu, Yanxin Jiang, and Yih-Chun Hu. 2024. YouTube/Bilibili Audience Retention Dataset. (March 2024). https://github.com/flowtele/vRetention

[6] Tianyu Chen, Yiheng Lin, Nicolas Christianson, Zahaib Akhtar, Sharath Dharmaji, Mohammad Hajiesmaili, Adam Wierman, and Ramesh K. Sitaraman. 2024. SODA: An Adaptive Bitrate Controller for Consistent High-Quality Video Streaming. In *Proceedings of the ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 613–644. https://doi.org/10.1145/3651890.3672260

[7] CTIA. 2025. 5G in America. Online; accessed Jan 31, 2025. (2025). https://www.ctia.org/the-wireless-industry/5g-in-america CTIA - The Wireless Industry.

[8] DASH Industry Forum. 2023. Example of Configuring Control Bar in dash.js. github repo. (2023). https://reference.dashif.org/dash.js/v4.4.0/samples/getting-started/controlbar.html Accessed: Jan 31, 2025.

[9] DASH Industry Forum. 2024. dash.js: A Reference Client Implementation for the Playback of MPEG DASH via JavaScript and Compliant Browsers. https://github.com/Dash-Industry-Forum/dash.js. (2024). Accessed Jan 31, 2025.

[10] Jonathan Deber, Ricardo Jota, Clifton Forlines, and Daniel Wigdor. 2015. How Much Faster is Fast Enough? User Perception of Latency & Latency Improvements in Direct and Indirect Touch. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. Association for Computing Machinery, New York, NY, USA, 1827–1836. https://doi.org/10.1145/2702123.2702300

[11] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. 2011. Understanding the impact of video quality on user engagement. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*. Association for Computing Machinery, New York, NY, USA, 362–373. https://doi.org/10.1145/2018436.2018478

[12] FFmpeg Development Team. 2023. FFmpeg. Open source multimedia framework for decoding, encoding, transcoding, muxing, demuxing, streaming, filtering, and playing. (2023). https://ffmpeg.org Accessed Jan 21, 2025.

[13] Fiber Broadband Association. 2025. Fiber Broadband Association Reports Record Fiber-To-The-Home Deployment in 2024. https://fiberbroadband.org/2025/01/23/fiber-broadband-association-reports-record-fiber-to-the-home-deployment-in-2024/. (23 January 2025). Press release by the Fiber Broadband Association.

[14] DASH Industry Forum. [n. d.]. Dash.js/Usage/Low Latency Streaming. https://dashif.org/dash.js/pages/usage/low-latency.html. ([n. d.]). Accessed October 15, 2025.

[15] DASH Industry Forum. 2017. DASH-IF publishes Low-Latency DASH extensions and 2nd Community Review for DASH-IF Ad Insertion. https://dashif.org/news/low-latency-dash/. (2017). Accessed October 15, 2025.

[16] Cristos Goodrow. 2017. You know what's cool? A billion hours. https://blog.youtube/news-and-events/you-know-whats-cool-billion-hours/. (Feb. 2017). Accessed May 26, 2025.

[17] Thomas Guilbert. 2024. *HTMLVideoElement.requestVideoFrameCallback()*. Draft Community Group Report. Web Platform Incubator Community Group (WICG). https://wicg.github.io/video-rvfc/ Accessed Jan 22, 2025.

[18] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. 2014. A buffer-based approach to rate adaptation: evidence from a large video streaming service. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 187–198. https://doi.org/10.1145/2619239.2626296

[19] Bert Hubert. 2025. tc(8) - Linux man page. Online. (2025). https://man7.org/linux/man-pages/man8/tc.8.html Accessed Jan 1, 2025.

[20] Ricardo Jota, Albert Ng, Paul Dietz, and Daniel Wigdor. 2013. How fast is fast enough? a study of the effects of latency in direct-touch pointing tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. Association for Computing Machinery, New York, NY, USA, 2291–2300. https://doi.org/10.1145/2470654.2481317

[21] S. Shunmuga Krishnan and Ramesh K. Sitaraman. 2012. Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs. In *Proceedings of the 2012 Internet Measurement Conference (IMC '12)*.

Association for Computing Machinery, New York, NY, USA, 211–224. https://doi.org/10.1145/2398776.2398799

[22] Zhuqi Li, Yaxiong Xie, Ravi Netravali, and Kyle Jamieson. 2023. Dashlet: Taming Swipe Uncertainty for Robust Short Video Streaming. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1583–1599. https://www.usenix.org/conference/nsdi23/presentation/li-zhuqi

[23] Melissa Licciardello, Maximilian Grüner, and Ankit Singla. 2020. Understanding video streaming algorithms in the wild. In *International Conference on Passive and Active Network Measurement*. Springer, Online, 298–313.

[24] Axiomatic Systems LLC. 2023. Bento4 - MP4 & MPEG DASH Library and Tools. Available under GPL and commercial licenses. (2023). https://www.bento4.com Accessed Jan 21, 2025.

[25] I. Scott MacKenzie, Tatu Kauppinen, and Miika Silfverberg. 2001. Accuracy measures for evaluating computer pointing devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '01)*. Association for Computing Machinery, New York, NY, USA, 9–16. https://doi.org/10.1145/365024.365028

[26] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 197–210. https://doi.org/10.1145/3098822.3098843

[27] Baiju Muthukadan. 2011. Selenium with Python. (2011). https://selenium-python.readthedocs.io Accessed April 14, 2023.

[28] Hyunwoo Nam, Kyung-Hwa Kim, and Henning Schulzrinne. 2016. QoE matters more than QoS: Why people stop watching cat videos. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. IEEE, San Francisco, CA, 1–9. https://doi.org/10.1109/INFOCOM.2016.7524426

[29] Netflix. 2021. A Cooperative Approach to Content Delivery: A Netflix Briefing Paper. Netflix open connect; accessed Jan 31, 2025. (2021). https://openconnect.netflix.com/Open-Connect-Briefing-Paper.pdf Netflix briefing paper on Open Connect.

[30] Ookla.LLC. 2025. Speedtest Global Index: Internet Speed Around the World. Online; accessed Jan 31, 2025. (2025). https://www.speedtest.net/global-index Speedtest Global Index report on global internet speeds.

[31] Minsu Park, Mor Naaman, and Jonah Berger. 2021. A Data-Driven Study of View Duration on YouTube. *Proceedings of the International AAAI Conference on Web and Social Media* 10, 1 (Aug. 2021), 651–654. https://doi.org/10.1609/icwsm.v10i1.14781

[32] Sandvine. 2023. Global Internet Phenomena Report. https://www.prnewswire.com/news-releases/sandvines-2023-global-internet-phenomena-report-shows-24-jump-in-video-traffic-with-netflix-volume-overtaking-youtube-301723445.html. (2023). Accessed April 23, 2023.

[33] Andreas Schmid, Maria Ambros, Johanna Bogon, and Raphael Wimmer. 2024. Measuring the Just Noticeable Difference for Audio Latency. In *Proceedings of the 19th International Audio Mostly Conference: Explorations in Sonic Cultures (AM '24)*. Association for Computing Machinery, New York, NY, USA, 325–331. https://doi.org/10.1145/3678299.3678331

[34] Satadal Sengupta, Niloy Ganguly, Sandip Chakraborty, and Pradipta De. 2018. HotDASH: Hotspot Aware Adaptive Video Streaming Using Deep Reinforcement Learning. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, Cambridge, UK, 165–175. https://doi.org/10.1109/ICNP.2018.00026

[35] Daniel Silhavy, Stefan Pham, Stefan Arbanowski, Stephan Steglich, and Björn Harrer. 2022. Latest advances in the development of the open-source player dash.js. In *Proceedings of the 1st Mile-High Video Conference (MHV '22)*. Association for Computing Machinery, New York, NY, USA, 32–38. https://doi.org/10.1145/3510450.3517311

[36] Iraj Sodagar. 2011. The mpeg-dash standard for multimedia streaming over the internet. *IEEE multimedia* 18, 4 (2011), 62–67.

[37] Kevin Spiteri, Ramesh Sitaraman, and Daniel Sparacio. 2019. From Theory to Practice: Improving Bitrate Adaptation in the DASH Reference Player. *ACM Trans. Multimedia Comput. Commun. Appl.* 15, 2s, Article 67 (July 2019), 29 pages. https://doi.org/10.1145/3336497

[38] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K. Sitaraman. 2020. BOLA: Near-Optimal Bitrate Adaptation for Online Videos. *IEEE/ACM Transactions on Networking* 28, 4 (2020), 1698–1711. https://doi.org/10.1109/TNET.2020.2996964

[39] David Tuber and Vasilis Giotsas. 2021. Unboxing the Last Mile: Introducing Last Mile Insights. Cloudflare Blog; accessed Jan 31, 2025. (16 September 2021). https://blog.cloudflare.com/last-mile-insights/ Cloudflare blog post on Last Mile Insights.

[40] Kurt Wilms. 2024. Smash that replay button: A 2024 recap of YouTube on TV. https://blog.youtube/news-and-events/2024-recap-of-youtube-on-tv/. (Dec. 2024). Accessed May 26, 2025.

[41] F. Y. Yan, H. Ayers, C. Zhu, S. Fouladi, J. Hong, K. Zhang, P. Levis, and K. Winstein. 2020. Learning in Situ: A Randomized Experiment in Video Streaming. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, USENIX, Santa Clara CA, 495–511. https://www.usenix.org/conference/nsdi20/presentation/yan

[42] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. 2015. A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data*

Communication (SIGCOMM '15). Association for Computing Machinery, New York, NY, USA, 325–338. https://doi.org/10.1145/2785956.2787486

[43] YouTube Support. 2025. YouTube Recommended Upload Encoding Settings. Online; accessed Jan 31, 2025. (2025). https://support.google.com/youtube/answer/1722171 YouTube support page on video encoding recommendations.

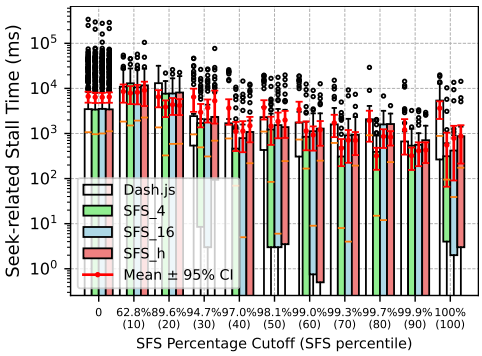
[44] yt-dlp contributors. 2023. yt-dlp: A command-line video downloader. GitHub repository. (2023). https://github.com/yt-dlp/yt-dlp Accessed Jan 22, 2025.

[45] Tong Zhang, Fengyuan Ren, Wenxue Cheng, Xiaohui Luo, Ran Shu, and Xiaolan Liu. 2017. Modeling and analyzing the influence of chunk size variation on bitrate adaptation in DASH. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. IEEE, Atlanta, GA, 1–9. https://doi.org/10.1109/INFOCOM.2017.8057057

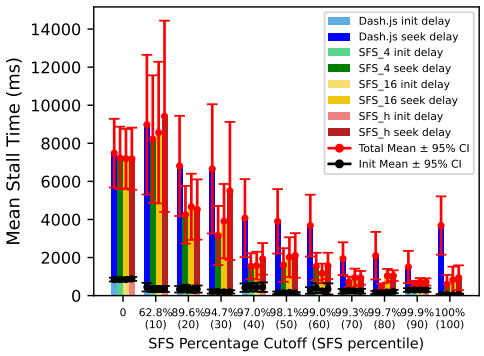
[46] Jiayu Zhu and Yifang Zhang. 2026. camera_ready_sfs. (2026). Retrieved January 18, 2026 from https://github.com/jimmyzhu12/camera_ready_sfs

A Long Video Test

Though the focus of our paper is on videos up to around 15 minutes, as such videos are most common on YouTube, in this section we evaluate three different strategies for achieving SFS on longer videos, to validate the potential of SFS in longer videos. We select one long video, with duration 1 hour and 4 seconds. We process the video in the same way as our other YouTube videos, as described in §5.5. We evaluate three different parameterizations for SFS: the first parameterization fetches *every* SFS chunk, which allows for seek with 4-second granularity (labeled SFS_4 in our results). The second parameterization fetches *every fourth* SFS chunk, which allows for seek with 16-second granularity (labeled SFS_16 in our results). The final parameterization starts by fetching *every fourth* SFS chunk, but also fetches every SFS chunk near the boundary of the linear buffer; this hybrid scheme is labeled SFS_h in our results. We rewrite seek times to the finest granularity at which SFS chunks are buffered (that is, 16 seconds for SFS_16 and 4 seconds for SFS_4 and SFS_h). As in our previous YouTube tests, we test each mode against 5 bandwidth groups. As a result, for each SFS chunk prefetch interval and the original Dash.js, we perform 1000 runs. Due to time constraints, we only analyzed one video; the reduced number of runs results in larger error bars as compared to our other experiments.

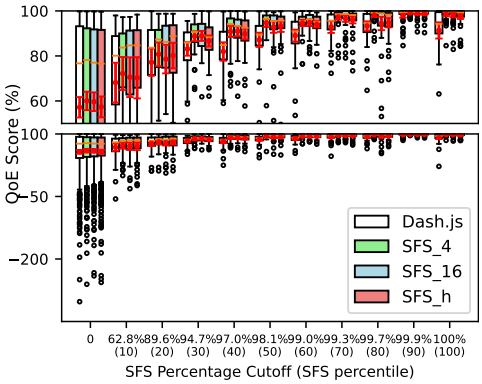


(a) SFS slightly improves seek-related stall at high SFS usage.

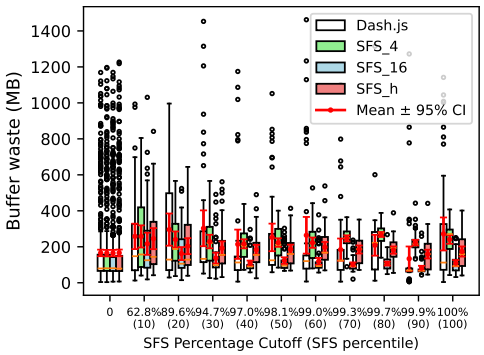


(b) SFS improves mean stall time.

Fig. 13. Seek-related stall of SFS on long YouTube video. For each SFS decile, we plot Dash.js, SFS chunk per 4s, SFS chunk per 16s, SFS hybrid.

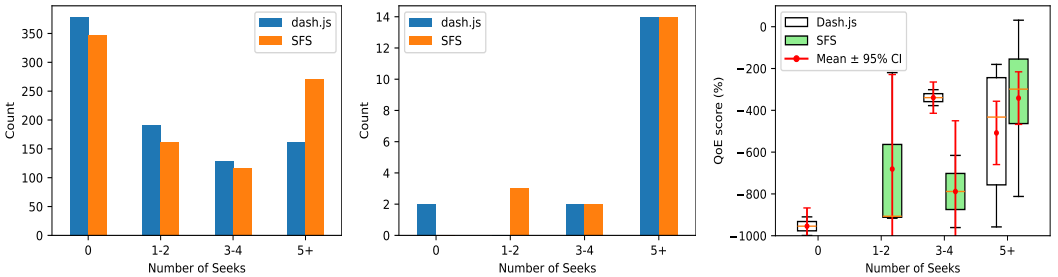


(a) SFS improves QoE.



(b) SFS variants have different buffer waste.

Fig. 14. QoE and buffer waste of SFS on long YouTube video.



(a) Number of seeks across the best 98% of total stall times.

(b) Number of seeks across the worst 2% of total stall times.

(c) QoE of the worst 2% of total stall time.

Fig. 15. User test: seek distribution and QoE.

Stall Time. Figure 13a plots seek-related stall against SFS decile from our long video test. Across all deciles, the performance of SFS variants is very similar. Furthermore, at the highest decile of SFS usage, seek-related stall time is lower for SFS than for Dash.js by a statistically significant margin. The trend of seek time improvement with increasing SFS decile mirrors the results of SFS with shorter videos, suggesting that statistically significant improvements in seek time could be demonstrated simply by running more videos. Figure 13b plots mean stall time, divided into initial stall times and seek-related delays against SFS decile. As with our experiments with shorter videos, initial delays are comparable across Dash.js and all SFS parameterizations, and SFS generally improves average delays (with the single, not statistically significant exception being that SFS_h underperforms Dash.js at the lowest decile of SFS usage), with the improvement becoming statistically significant at the highest decile of SFS usage.

QoE. Figure 14a plots QoE against SFS decile. Runs with SFS enabled, regardless of fetching strategy, and regardless of SFS usage, never provide worse QoE on average, and in some categories (such as the top decile of SFS usage) provide better QoE by a statistically significant margin.

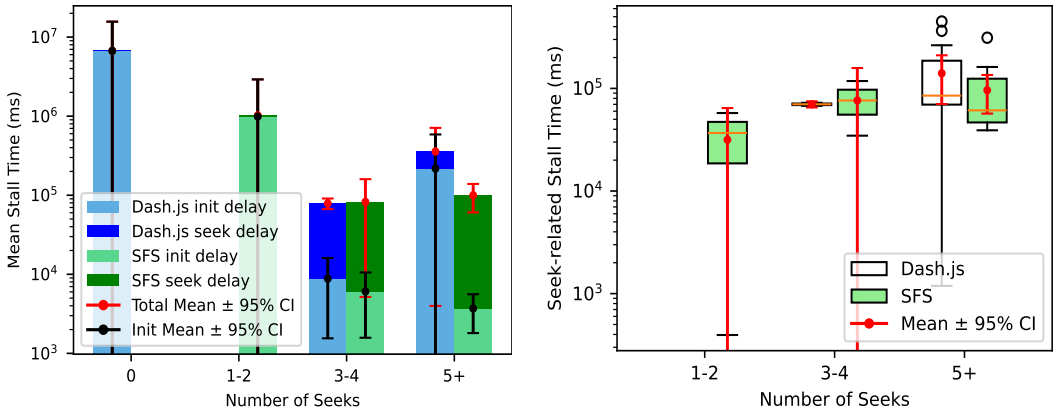
Buffer Waste. Figure 14b plots buffer waste against SFS decile. Buffer waste is the metric on which the SFS variants will display the greatest level of variation, with SFS_4 showing the largest amount of waste, SFS_16 showing the least waste, and SFS_h aiming to be the middle ground. Focusing on statistically significant results, SFS_16 has less buffer waste than Dash.js at 8 different SFS deciles, and SFS_4 has more buffer waste than Dash.js at one SFS decile. Qualitatively, the hybrid scheme appears to achieve rough parity with Dash.js across all SFS deciles. These results show that SFS can balance user experience and buffer waste across a variety of video lengths.

Overall, our experiments with long videos show that even naïve parameterizations of SFS (such as fetching all SFS chunks) can provide good QoE and stall time behavior for videos as long as one hour while not having significantly worse buffer waste than Dash.js.

B User Test

As mentioned in Section 5.7, the evaluation graphs from our main paper show the lowest (best) 98% of total stall time for Dash.js and SFS runs. In this section, we plot the performance of the remaining 2% of runs.

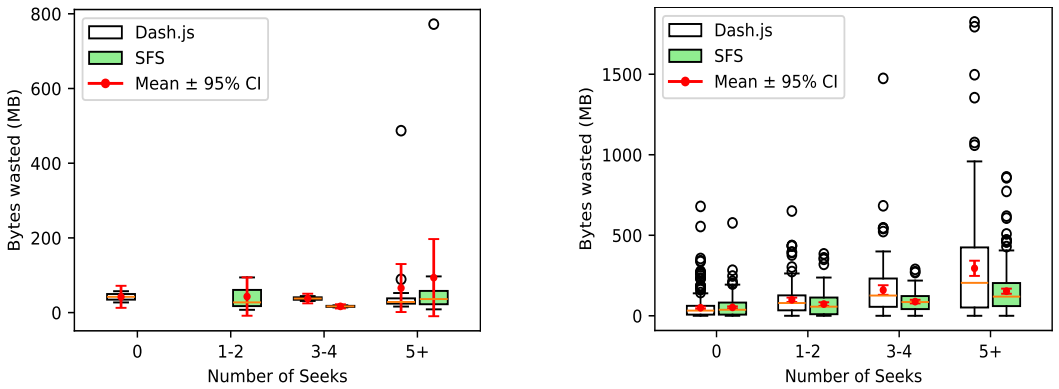
There are only 18 traces from the Dash.js group and 19 traces from the SFS group, so there are not samples for all of the "Number of Seeks" on the x-axis. In particular, there are no Dash.js traces with 1–2 seeks. Furthermore, the traces are heavily weighted towards 5+ seeks, because these graphs reflect the traces with the highest total stall time, and additional seeks increases the chances of increased stall time. The distribution of these bottom 2% of runs is shown in Figure 15b, which can be compared to the distribution from the best 98% which is shown in Figure 15a.



(a) Mean stall time.

(b) Seek-related stall time.

Fig. 16. Mean and seek-related stall times in the user test, of *the worst 2% of total stall time.*



(a) Buffer waste, worst 2% of total stall time.

(b) Buffer waste across entire user test.

Fig. 17. User test buffer waste.

Figure 15c plots the QoE of the worst 2% of total stall time results from the user test against the number of seeks. High variance makes it difficult to find statistically significant results; however, in the few runs with 3–4 seeks, Dash.js appears to have better QoE than SFS, though the sample size is small (26 runs for each of Dash.js and SFS). In particular, Figure 16a shows that for the 3–4 seek range, the total stall time of Dash.js and SFS is almost identical, and well within the 95% confidence interval. This means that a significant portion of the QoE difference comes from *delivered bandwidth*, which is a factor that is not affected by the SFS code, and is likely due to variation in the available bandwidth for the users on these two SFS traces and the two Dash.js traces.

As mentioned previously, Figure 16a shows the total stall time for the worst 2% of stall time results from the user test, plotted on a log scale. These results show that on average, even among the worst 2% of stall times, SFS continues to have comparable or better average stall time than Dash.js, though the variance is too high to reach any statistically significant conclusions.

Though it may appear from Figure 16a that the seek delay of Dash.js with 5 or more seeks is less than the seek delay of SFS, this appearance is only an illusion caused by the log-scale of the y-axis. Figure 16b isolates the seek-related stalls for the worst 2% of stall times, and SFS has comparable or better average seek-related stall time than Dash.js, though the variance is again too high to reach any statistically significant conclusions.

Figure 17a plots the buffer waste for the worst 2% of stall times. This result shows negligible difference between Dash.js and SFS. Because buffer waste is the one metric we use that does not depend on stall time, the heavy-tailed nature of stall time in our user experiment does not impact this graph. Thus, the most accurate graph for buffer waste is the one that does not separate by stall time, which is shown in Figure 17b. SFS is slightly worse with zero seeks, slightly better with 1–2 seeks, and better by a statistically significant margin with at least 3 seeks.

Received June 2025; revised November 2025; accepted December 2025