A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP

Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, Bruno Sinopoli
Carnegie Mellon University
{yinxiaoqi522, abhishekjindal93}@gmail.com, {vsekar,brunos}@andrew.cmu.edu

ABSTRACT

User-perceived quality-of-experience (QoE) is critical in Internet video applications as it impacts revenues for content providers and delivery systems. Given that there is little support in the network for optimizing such measures, bottlenecks could occur anywhere in the delivery system. Consequently, a robust bitrate adaptation algorithm in client-side players is critical to ensure good user experience. Previous studies have shown key limitations of state-of-art commercial solutions and proposed a range of heuristic fixes. Despite the emergence of several proposals, there is still a distinct lack of consensus on: (1) How best to design this client-side bitrate adaptation logic (e.g., use rate estimates vs. buffer occupancy); (2) How well specific classes of approaches will perform under diverse operating regimes (e.g., high throughput variability); or (3) How do they actually balance different QoE objectives (e.g., startup delay vs. rebuffering). To this end, this paper makes three key technical contributions. First, to bring some rigor to this space, we develop a principled control-theoretic model to reason about a broad spectrum of strategies. Second, we propose a novel *model predictive control* algorithm that can optimally combine throughput and buffer occupancy information to outperform traditional approaches. Third, we present a practical implementation in a reference video player to validate our approach using realistic trace-driven emulations.

CCS Concepts

•Information systems → Multimedia streaming; •Networks → Network protocol design; Application layer protocols;

Keywords

Internet Video; Bitrate Adaptation; DASH; Model Predictive Control

SIGCOMM '15, August 17 - 21, 2015, London, United Kingdom

DOI: http://dx.doi.org/10.1145/2785956.2787486

1 Introduction

Many recent studies have highlighted the critical role that user-perceived quality-of-experience (QoE) plays in Internet video applications, as it ultimately affects revenue streams for content providers [24, 35]. Specifically, metrics such as the duration of rebuffering (i.e., the player's playout buffer does not have content to render), startup delay (i.e., the lag between the user clicking vs. the time to begin rendering), the average playback bitrate, and the variability of the bitrate delivered have emerged as key factors.

Given the complex Internet video delivery ecosystem and presence of diverse bottlenecks, the *bitrate adaptation logic* in the client-side video player becomes critical to optimize user experience [16]. In the HTTP-based delivery model that predominates today [44], videos are typically chunked and encoded at different bitrate levels. The goal of an adaptive video player is to choose the bitrate level for future chunks to deliver the highest possible QoE; e.g., maximizing bitrate while minimizing the likelihood of rebuffering and avoiding too many bitrate switches.

Many recent efforts have pointed out key challenges in designing this adaptation logic (e.g., [46, 17, 32, 34]) and several proposals have emerged to try and address these challenges (e.g., [34, 17, 33]). Despite the proliferation of numerous algorithms, however, there appears to be a lack of clarity and consensus across these solutions on several fronts; e.g., some argue for better throughput estimation [47], while others suggest improving chunk scheduling [34]. Some researchers even argue against *rate-based* approaches that rely on throughput estimates from previous chunk downloads and make the case for *buffer-occupancy based* algorithms that make their decisions purely based on buffer occupancy [33].

In order to understand the fundamental tradeoffs between different classes of algorithms (e.g., rate- vs buffer-based) under different operating regimes (e.g., low vs. high throughput variability), we begin by formulating the video bitrate adapdation as a *stochastic optimal control* problem. We formally define the key dynamic variables involved in the video adaptation problem and a concrete objective. This framework allows us to outline the broader design space of control algorithms for this problem. We identify a key shortcoming in existing approaches that rely exclusively on pure rate- or buffer-based strategies, and that might be potentially missing out on strategies that *combine* both signals.

Building on insights from the control-theoretic formulation, we argue that *model predictive control (MPC)* [22] is

a suitable class of algorithms that can optimally combine both rate-based and buffer-based feedback signals. At a high level, MPC attempts to predict key environment variables over a moving look-ahead horizon and solve an exact optimization problem based on the prediction. MPC is the technology of choice in a multitude of real world control problems [22]. In addition to its intuitive formulation, it can explicitly handle complex control objectives and constraints, and has a set of well understood tuning parameters such as the prediction horizon. Moreover, MPC has other qualitative advantages as its development time is much shorter compared to advanced control methods and it is easier to maintain, as changing model parameters does not require complete redesign.

In our context, the MPC approach entails predicting the expected throughput for the next few chunks and using this to make optimal bitrate decisions for QoE maximization. Indeed, our simulation results confirm that if we could run an optimal MPC algorithm and the prediction error was low, then the MPC scheme can outperform traditional rate-based and buffer-based strategies.

In practice, however, running a MPC-based algorithm is challenging because it needs to solve a non-trivial discrete optimization problem at each time step. Even ignoring the computational overhead, there are practical difficulties as we might need to bundle this solver logic with every video player or require users to download and install additional software. To address these challenges, we develop a simple-yet-efficient *FastMPC* mechanism. Conceptually, FastMPC essentially follows a table enumeration approach, where we describe the problem state-space, solve the specific instances optimally offline, and store the optimal control decisions for future online use. If implemented naively, however, the size of this table can induce significant memory overhead and startup delays for video players (e.g., additional JavaScript to load). Fortunately, we show that with a simple value binning and compression strategy, we can achieve near-optimal performance with manageable table sizes.

We have prototyped our FastMPC bitrate adaptation algorithm in an open source dynamic adaptive streaming player called `dash.js` [1]. Our choice of platform is a pragmatic one—it is the reference open-source implementation for the MPEG-DASH standard based on the HTML5 specification and is actively supported by leading industry participants [7]. We show that our implementation adds negligible overhead to the baseline `dash.js` player. We also showcase the FastMPC-based player in our demo page [14].

We evaluate our algorithms and prototype implementation using realistic emulation experiments on measured [9, 10] and synthetic throughput variability traces. We also augment these results with simulation-based sensitivity analysis experiments to analyze the effect of key operating parameters on the performance of different classes of algorithms. Our key findings are:

1. Our proposed MPC approach consistently outperforms the state-of-art adaptation algorithms by 15% in broadband (FCC) dataset and 10% in cellular (HSDPA) dataset in terms of median QoE. It also achieves significant improvement (60+% median QoE) compared to the industry reference player `dash.js`;
2. Our fast and low-overhead implementation FastMPC requires similar CPU usage and only 60 kB extra memory usage comparing to other algorithms.

Contributions and roadmap: In summary, this paper makes the following key contributions:[1]

- Development of a formal control-theoretic model of the bitrate adaptation problem (Section 3);
- Design of a MPC approach that subsumes existing rate- and buffer-based strategies (Section 4);
- A practical and fast table enumeration based algorithm FastMPC that near-optimally approximates the performance of an exact MPC approach (Section 5);
- A low-overhead implementation based on the open source reference video player `dash.js` (Section 6);
- A systematic evaluation of different classes of algorithms over a wide range of operating parameters and realistic traces (Section 7)

We begin by discussing background on DASH and related work in the next section.

2 Background and Related Work

We begin with a high-level overview of how HTTP-based adaptive video streaming works, before describing the key shortcomings of today's state-of-art solutions.

Internet video technologies such as Microsoft Smooth-Streaming [13], Apple's HLS [5], and Adobe's HDS [2] rely on *HTTP-based* adaptive streaming. This class of protocols is being standardized under the umbrella of Dynamic Adaptive Streaming over HTTP or DASH [16]. In DASH systems, each video consists of multiple segments or "chunks" (corresponding to a few seconds of play time) and each chunk is encoded at multiple *discrete* bitrates. The chunks from different bitrate streams are aligned so that the video player can switch to a different bitrate if necessary at a chunk boundary. This approach has several pragmatic advantages over custom streaming protocols such as Real-Time Messaging Protocol (RTMP). The use of HTTP enables providers to seamlessly bypass middleboxes. Furthermore, it can use existing commodity CDN servers without requiring custom modifications. Finally, by making the server stateless, one can implement better application-layer resilience using multiple servers and CDNs [41, 40].

Figure 1 shows an abstract model of the adaptive video player. The player uses some inputs (e.g., buffer occupancy or estimates of the network throughput) in its decision logic to choose the bitrate level for the next chunk(s) to be downloaded. In making this decision, there are many potentially conflicting QoE considerations a player must account for: (1) minimizing rebuffering events where the playback buffer

[1] An early workshop version of the paper made the case for a MPC-based approach [50]. However, it did not provide a concrete algorithm, a practical implementation, and evaluation using real throughput traces.

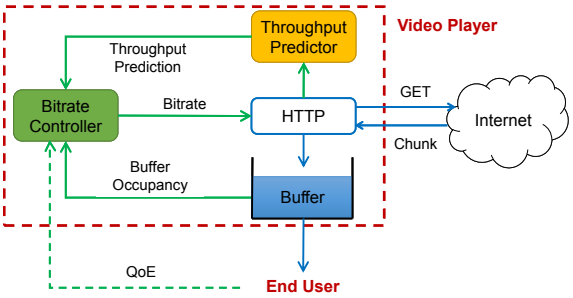


Figure 1: Abstract model of DASH players

is empty and cannot render the video; (2) delivering as high a playback bitrate as possible within the throughput constraints; (3) minimizing startup delay so that the user does not quit while waiting for the video to load; and (4) keeping the playback as smooth as possible by avoiding frequent or large bitrate jumps [24, 35].

To see why these objectives are conflicting, let us consider two extreme solutions. A trivial solution to minimize rebuffering and the startup delay would be to always pick the lowest bitrate, but it conflicts with the goal of delivering high bitrate. Conversely, picking the highest available bitrate may lead to many rebuffering events. Similarly, the goal of maintaining a smooth playback may also conflict if the optimal choice to simultaneously minimize rebuffering and maximizing average bitrate is to switch bitrates for every chunk.

The focus of this paper is on *client-side* adaptation solutions. Other complementary work includes the use of server-side bitrate switching (e.g., [37, 18]), TCP changes to avoid bursts (e.g., [27]), and in-network throughput management and caching (e.g., [31, 45, 42]). We focus on the client-side problem for two key reasons. First, client-side solutions offer the most immediately deployable alternative in contrast to solutions that require in-network support (e.g., [31, 45, 42]), server-side software changes (e.g., [37, 18]), or modifications to lower-layer transport protocols (e.g., [27, 28, 39, 29, 36]). Second, the client is often in the best position to quickly detect performance issues and respond to dynamics. That said, we believe that the formal foundations and algorithms we develop can be equally applied to these other deployment scenarios.

Many measurement studies have shown the poor performance of state-of-art video players with respect to these QoE measures (e.g., [46, 34, 32]). These studies show that most problems are not artifacts of specific players but manifest across all state-of-art players such as SmoothStreaming [13], Netflix [11], Adobe OSMF [3], and Akamai HD [4]. For brevity we do not reproduce these results here but refer interested readers to prior work (e.g., [46, 34, 32]).

To alleviate these problems, there have been several recent proposals in the research literature (e.g., [47, 34, 18, 33, 38]). At a high level, these solutions can be roughly divided into two categories: (1) *rate-based algorithms* and (2) *buffer-based algorithms*. Video players with rate-based methods essentially pick the highest possible bitrate based on the estimated available throughput. However, as shown in prior work throughput estimation on top of HTTP suffers from significant biases [32], which leads to problems with traditional rate-based approaches. Some solutions try to work around these biases by either smoothing out throughput estimates [47] or choosing better scheduling strategies [34]. On the other hand, recent work makes a case for buffer-based algorithms [33]. Rather than using throughput estimates, this class of algorithms uses buffer occupancy as the feedback signal, and designs mechanisms to keep the buffer occupancy at a desired level, essentially discarding any available throughput information.

Despite the broad interest in this topic, what is critically lacking today is a principled understanding of bitrate adaptation algorithms. Each aforementioned solution offers point heuristics that work under specific (and implicit) environmental assumptions. While each approach seen in isolation has been shown to outperform commercial players, there is little effort to systematically compare how different *classes of algorithms* stack up against each other or which of these technical components are *critical*, or how *robust* these algorithms are across different operating regimes (e.g., throughput stability, buffer size, number of bitrate levels). Furthermore, many of these algorithms even fail to formally state what *objective* they seek to optimize making it harder to conduct a meaningful comparison.

Our first-order goal in this work is to bring some clarity to this space. Rather than design yet another point solution, we start by developing a first-principles approach via control theory to develop a general framework to reason about classes of algorithms. In the next section, we use this control-theoretic "lens" to formally define the stochastic optimization that video bitrate adaptation algorithms try to solve.

3 Control-Theoretic Model

In this section, we develop a mathematical model of the HTTP video streaming process and formally define the bitrate adaptation problem. This model gives us a framework to compare and evaluate existing algorithms and serves as the foundation for potential improvements.

3.1 Video Streaming Model

We model a video as a set of consecutive *video segments* or *chunks*, $\mathcal{V} = \{1, 2, \cdots, K\}$, each of which contains L seconds of video. Each chunk is encoded at different bitrates. Let $\mathcal{R}$ be the set of all available bitrate levels. The video player can choose to download chunk k at bitrate $R_k \in \mathcal{R}$. Let $d_k(R_k)$ be the size of chunk k encoded at bitrate R_k. In constant bitrate (CBR) case, $d_k(R_k) = L \times R_k$, while in variable bitrate (VBR) case the $d_k \sim R_k$ relationship can differ across chunks.

The higher bitrate is selected, the higher video quality is perceived by the user. Let $q(\cdot) : \mathcal{R} \to \mathbb{R}_+$ be a non-decreasing function which maps selected bitrate R_k to video quality perceived by user $q(R_k)$. Note that $q(\cdot)$ may depend on the video-playing device as well as the content of the video. For example, while on HDTV 3Mbps and 1Mbps may lead to significant difference in user experience, the

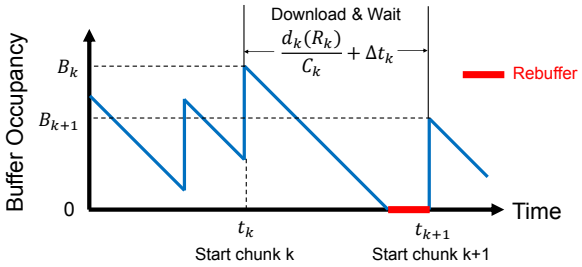


Figure 2: Illustration of buffer dynamics

video quality in 3Mbps and 1Mbps may be similar on a mobile device; Also, improving the bitrate of "dynamic" chunks will result in more QoE gain than improving "static" chunks.

The video segments are downloaded into a *playback buffer*, which contains downloaded but as yet unviewed video. Let $B(t) \in [0, B_{max}]$ be the *buffer occupancy* at time t, i.e., the play time of the video left in the buffer. The *buffer size* B_{max} depends on the policy of the service provider, as well as storage limitations on the player. A typical player buffer may hold few tens of seconds of video segments.

Figure 2 helps illustrate the conceptual operation of the video player. At time t_k, the video player starts to download chunk k. The download time for this chunk will be $d_k(R_k)/C_k$; i.e., it depends on the size of selected chunk with bitrate R_k, as well as average download speed C_k experienced during this download process. Once chunk k is completely downloaded, the video player waits for Δt_k and starts to download the next chunk $k+1$ at time t_{k+1}. We assume that the waiting time Δt_k is small and will not lead to rebuffering events. If we denote by C_t the network throughput at time t, then we have:

$$t_{k+1} = t_k + \frac{d_k(R_k)}{C_k} + \Delta t_k \tag{1}$$

$$C_k = \frac{1}{t_{k+1} - t_k - \Delta t_k} \int_{t_k}^{t_{k+1}-\Delta t_k} C_t \, dt. \tag{2}$$

The buffer occupancy $B(t)$ evolves as the chunks are being downloaded and the video is being played. Specifically, the buffer occupancy increases by L seconds after chunk k is downloaded and decreases as the user watches the video.[2] Let $B_k = B(t_k)$ denote the buffer occupancy when the player starts to download chunk k. The buffer dynamics can then be formulated as:

$$B_{k+1} = \left(\left(B_k - \frac{d_k(R_k)}{C_k} \right)_+ + L - \Delta t_k \right)_+ \tag{3}$$

Here, the notation $(x)_+ = \max\{x, 0\}$ ensures that the term can never be negative. Note that if $B_k < d_k(R_k)/C_k$, the buffer becomes empty while the video player is still downloading chunk k, leading to *rebuffer* events as shown in Figure 2.

[2]The "startup" phase will be slightly different as the player waits for some amount of buffer to build up before draining the buffer.

The determination of waiting time Δt_k, also referred as *chunk scheduling* problem, is an equally interesting and important problem in improving fairness of multi-player video streaming [34]. However, in this paper we assume that the player immediately starts to download chunk $k+1$ as soon as chunk k is downloaded. The one exception is when the buffer is full, the player waits for the buffer to reduce to a level which allows chunk k to be appended. Formally,

$$\Delta t_k = \left(\left(B_k - \frac{d_k(R_k)}{C_k} \right)_+ + L - B_{max} \right)_+ \tag{4}$$

3.2 QoE Maximization Problem

The ultimate goal of bitrate adaptation is to improve the QoE of users in order to achieve higher long-term user engagement [24]. Our goal is to provide a flexible QoE model rather than a fixed notion of QoE as this is an active area of research [19]. While users may differ in their specific QoE functions, we can enumerate the key elements of video QoE as:

1. *Average Video Quality:* The average per-chunk quality over all chunks: $\frac{1}{K}\sum_{k=1}^{K} q(R_k)$;
2. *Average Quality Variations*: This tracks the magnitude of the changes in the quality from one chunk to another: $\frac{1}{K-1}\sum_{k=1}^{K-1} |q(R_{k+1}) - q(R_k)|$;
3. *Rebuffer*: For each chunk k rebuffering occurs if the download time $d_k(R_k)/C_k$ is higher than the playout buffer level when the chunk download started (i.e., B_k). Thus the *total rebuffer time* [3] is $\sum_{k=1}^{K} \left(\frac{d_k(R_k)}{C_k} - B_k \right)_+$.
4. *Startup Delay* T_s, assuming $T_s \ll B_{max}$.

As users may have different preferences on which of the four components is more important, we define the QoE of video segment 1 through K by a weighted sum of the aforementioned components:

$$QoE_1^K = \sum_{k=1}^{K} q(R_k) - \lambda \sum_{k=1}^{K-1} |q(R_{k+1}) - q(R_k)| - \mu \sum_{k=1}^{K} \left(\frac{d_k(R_k)}{C_k} - B_k \right)_+ - \mu_s T_s \tag{5}$$

Here λ, μ, μ_s are non-negative weighting parameters corresponding to video quality variations, rebuffering time and startup delay, respectively. A relatively small λ indicates that the user is not particularly concerned about video quality variability; the large λ is, the more effort is made to achieve smoother changes of video quality. A large μ, relatively to the other parameters, indicates that a user is deeply concerned about rebuffering. In cases where users prefer low startup delay, we employ a large μ_s.

In summary, this definition of QoE is quite general as it allows us to model varying user preferences on different contributing factors.

[3]Alternatively, one can also consider the *number of rebuffering events* formulated as $\sum_{k=1}^{K} \mathbf{1}\left(\frac{d_k(R_k)}{C_k} > B_k \right)$.

$$\max_{R_1,\cdots,R_K,T_s} \quad QoE_1^K \tag{6}$$

$$s.t. \qquad t_{k+1} = t_k + \frac{d_k(R_k)}{C_k} + \Delta t_k, \tag{7}$$

$$C_k = \frac{1}{t_{k+1} - t_k - \Delta t_k} \int_{t_k}^{t_{k+1}-\Delta t_k} C_t \, dt, \tag{8}$$

$$B_{k+1} = \left(\left(B_k - \frac{d_k(R_k)}{C_k} \right)_+ + L - \Delta t_k \right)_+, \tag{9}$$

$$B_1 = T_s, \quad B_k \in [0, B_{max}] \tag{10}$$

$$R_k \in \mathcal{R}, \quad \forall k = 1, \cdots, K. \tag{11}$$

Figure 3: Formulation for QoE maximization ($QOE\_MAX_1^K$) subject to buffer and throughput dynamics.

QoE maximization problem: We are now ready to formulate the problem of bitrate adaptation for QoE maximization as in Figure 3, denoted as $QOE\_MAX_1^K$. Given throughput trace $\{C_t, t \in [t_1, t_{K+1}]\}$ as input, the optimization provides the following as output: 1) bitrate decisions $R_1, \cdots, R_K$, and 2) startup time T_s.

Note that the problem $QOE\_MAX_1^K$ is formulated assuming the video playback has not started at the time of this optimization so the start-up delay T_s is a decision variable. However, this QoE maximization can also take place during video playback at time t_{k_0} when the next chunk to download is k_0 and the current buffer occupancy is B_{k_0}. In this case, we can drop the variable T_s and denote the corresponding *steady state* problem as $QOE\_MAX\_STEADY_{k_0}^K$.

3.3 Classes of Algorithms

In this section we characterize problem $QOE\_MAX_1^K$ and describe existing bitrate adaptation algorithms within this framework to understand how they relate to one another.

The problem in Figure 3 is a finite-horizon stochastic optimal control problem. The source of randomness is in the available throughput C_t. At time t_k when the player chooses bitrate R_k, only the past throughput $\{C_t, t \le t_k\}$ is available, while the future values $\{C_t, t > t_k\}$ are not known.

However, a *throughput predictor* can be used to obtain predictions defined as $\{\hat{C}_t, t > t_k\}$. Based on such prediction and on buffer occupancy information (which is instead known precisely), the *bitrate controller* selects bitrate of the next chunk k:

$$R_k = f\left(B_k, \{\hat{C}_t, t > t_k\}, \{R_i, i < k\}\right). \tag{12}$$

The design of effective throughput predictors is an interesting research direction in its own right. In this paper, we focus on bitrate adaptation algorithms only and assume that predictors are given to us and are characterized in terms of their expected prediction errors. Namely, we focus on the design of $f(\cdot)$ and on the effect of the prediction error on the performance of the compared control algorithms. In the following sections, we will systematically evaluate how different algorithms perform with a state-of-art predictor under a variety of variability conditions.

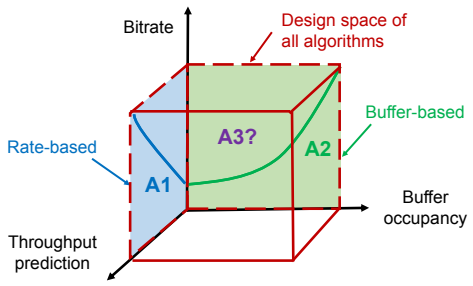


Figure 4: Design space of algorithms for the video adaptation problem: Most current approaches choose the bitrate as a function of only one variable; e.g., A1 is rate-based (RB) while A2 is buffer-based (BB).

Now, different adaptation algorithms essentially adopt different functions $f(\cdot)$. Specifically, two main categories of algorithms appear in the literature: *rate-based (RB)* and *buffer-based (BB)* algorithms. RB strategies essentially choose bitrate only based on throughput prediction, i.e.,

$$R_k = f\left(\{\hat{C}_t, t > t_k\}, \{R_i, i < k\}\right). \tag{13}$$

For example, a typical RB strategy is to choose the maximum possible bitrate below the predicted throughput.

On the other hand, BB strategies advocate decision making based only on buffer occupancy, namely:

$$R_k = f\left(B_k, \{R_i, i < k\}\right), \tag{14}$$

while regarding throughput variations as unmodeled disturbances. For example, Huang et al., illustrate one roadmap for designing BB algorithms [33].

Note, however, that both classes of algorithms are discarding possibly useful information as shown in Figure 4. Consequently, both are in principle suboptimal. Ideally, we want to use both buffer occupancy and throughput prediction, thereby considering a broader design space of bitrate adaptation strategies, as shown in Eq (12) and algorithm A3 depicted in Figure 4.

4 Model Predictive Control Approach for Optimal Bitrate Adaptation

In this section, we make a case for a Model Predictive Control (MPC) approach for bitrate adaptation and describe a concrete MPC-based workflow that can optimally combine throughput prediction and buffer occupancy. We also develop a robust MPC approach that can better handle errors in throughput prediction under highly variable network conditions.

4.1 Why MPC?

First, we provide the intuition behind the choice of MPC in our setting. Note that we cannot claim that MPC is necessary or the optimal choice in the space of all possible control algorithms. Our goal is merely to argue that MPC is a natural fit for the bitrate adaptation problem.

Strawman solutions: As we saw before, bitrate adaptation is essentially a stochastic optimal control problem. In this respect, there are two candidate well-known control algorithms: (1) Proportional-integral-derivative (PID) control [25] and (2) Markov Decision Process (MDP) based control [21]. While PID is computationally simpler compared to MPC, it can only serve to stabilize the system and cannot explicitly optimize our QoE objective. In addition, PID control is designed to work in continuous time and state space and using it in a highly discrete system such as ours may result in performance degradation or instability [25]. Alternatively, with MDP we could consider formulating the throughput and buffer state transition as Markov processes, and find the optimal control policy using standard algorithms such as value iteration or policy iteration [21]. However, this has a strong assumption that throughput dynamics follow Markov processes and it is unclear if this holds in practice. We regard the potential use of MDP and analysis of the throughput dynamics as future work (see Section 8).

Case for MPC: Ideally, given perfect knowledge of future throughput over the entire horizon of a video $[t_1, t_{K+1}]$, the optimal bitrate $R_1, \cdots, R_K$ and startup delay T_s can be calculated in one shot by solving the optimization problem for the entire video $QOE\_MAX_1^K$. In practice, such perfect information is not available, making it difficult to find such optimal solutions using offline optimization.

While perfect information may not be available for the entire future, it is possible that reasonably accurate throughput prediction can be instead obtained for a short horizon to the future $[t_k, t_{k+N}]$. The intuition here is that network conditions are reasonably stable on short timescales and usually do not change drastically during a short horizon (tens of seconds) [51]. Based on this insight, we can run a QoE optimization using the prediction in this horizon, apply the first bitrate R_k, and move the horizon forward to $[t_{k+1}, t_{k+N+1}]$. This scheme is known as model predictive control (MPC) or receding horizon control [22]. MPC algorithms are widely used in different domains, ranging from industrial control to navigation. The general benefits of MPC are in that MPC can utilize predictions to optimize a complex control objective online in a dynamical system under constraints.

4.2 Basic MPC Algorithm

Algorithm 1 shows a high-level overview of the workflow of MPC for bitrate adaptation. In our context, the algorithm essentially chooses bitrate R_k by looking N steps ahead (i.e., the moving horizon), and solves a specific QoE maximization problem (this depends on whether the player is in steady or startup phase) with throughput predictions $\{\hat{C}_t, t \in [t_k, t_{k+N}]\}$, or $\hat{C}_{[t_k,t_{k+N}]}$. The first bitrate R_k is applied by using feedback information and the optimization process is iterated at each step k.

At iteration k, the player maintains a moving horizon from chunk k to $k+N-1$ and carries out the following three key steps, as shown in Algorithm 1.

1. *Predict*: Predict throughput $\hat{C}_{[t_k,t_{k+N}]}$ for the next N chunks using some throughput predictor. Our goal in this paper is not to design a prediction mechanism but to rely on existing approaches. Naturally, improving the accuracy of this prediction will improve the gains achieved via MPC. That said, MPC can be extended to be robust to errors as we discuss below.
2. *Optimize*: This is the core of the MPC algorithm: Given the current buffer occupancy B_k, previous bitrate R_{k-1} and throughput prediction $\hat{C}_{[t_k,t_{k+N}]}$, find optimal bitrate R_k. In *steady state*, $R_k = f_{mpc}\left(R_{k-1}, B_k, \hat{C}_{[t_k,t_{k+N}]}\right)$, implemented by solving $QOE\_MAX\_STEADY_k^{k+N-1}$. In the *start-up phase*, it also optimizes start-up time T_s as $[R_k, T_s] = f_{mpc}^{st}\left(R_{k-1}, B_k, \hat{C}_{[t_k,t_{k+N}]}\right)$, implemented by solving $QOE\_MAX_k^{k+N-1}$. If we ignore practical details about computational overhead, we can simply use off-the-shelf solvers such as `CPLEX` to solve these discrete optimization problems. As we will see in Section 5, we do not need to explicitly solve the optimization problem within the video player in practice.
3. *Apply*: Start to download chunk k with R_k and move the horizon forward. If the player is in *start-up phase*, wait for T_s before starting playback.

Algorithm 1 Video adaptation workflow using MPC

```
Initialize
for k = 1 to K do
    if player is in startup phase then
        Ĉ_[t_k,t_{k+N}] = ThroughputPred(C_[t_1,t_k])
        [R_k, T_s] = f^st_mpc(R_{k-1}, B_k, Ĉ_[t_k,t_{k+N}])
        Start playback after T_s seconds
    else if playback has started then
        Ĉ_[t_k,t_{k+N}] = ThroughputPred(C_[t_1,t_k])
        R_k = f_mpc(R_{k-1}, B_k, Ĉ_[t_k,t_{k+N}])
    end if
    Download chunk k with bitrate R_k, wait till finished
end for
```

This workflow has several qualitative advantages compared with buffer-based (BB), rate-based (RB) as we discuss below. First, this MPC algorithm uses both throughput prediction and buffer information in a principled way. Second, compared to pure RB approaches, MPC smooths out prediction error at each step and is more robust to prediction errors. Specifically, by optimizing several chunks over a moving horizon, large prediction errors for one particular chunk will have lower impact on the performance. Third, MPC directly optimizes a formally defined QoE objective, while in RB and BB the tradeoff between different QoE factors is not clearly defined and therefore can only be addressed in an ad hoc qualitative manner.

4.3 Robust MPC

The basic MPC algorithm assumes the existence of an accurate throughput predictor. However, in certain severe net-

work conditions, e.g., in cellular networks or in prime time when the Internet is congested, such accurate predictors may not be available. For example, if the predictor consistently overestimates the throughput, it may induce high rebuffering. To counteract the prediction error, we develop a *robust MPC* algorithm.

Robust MPC essentially optimizes the worst-case QoE assuming that the actual throughput can take any value in a *range* $[\underline{\hat{C}_t}, \overline{\hat{C}_t}]$ in contrast to a point estimate $\hat{C}_t$. Robust MPC entails solving the following optimization problem at time t_k to get bitrate $R_k = f_{robustmpc}(R_{k-1}, B_k, [\underline{\hat{C}_t}, \overline{\hat{C}_t}])$:

$$\max_{R_k,\cdots,R_{k+N-1}} \min_{C_t \in [\underline{\hat{C}_t}, \overline{\hat{C}_t}]} QoE_k^{k+N-1} \quad (15)$$

$$s.t. \quad \text{Constraints (7) to (11)} \quad (16)$$

In general, it may be non-trivial to solve such a max-min robust optimization problem. In our specific case, however, we can prove that the worst case scenario takes place when the throughput is at its lower bound $C_t = \underline{\hat{C}_t}$. Thus, the implementation of robust MPC is straightforward. Instead of $\hat{C}_t$, we use the lowest possible $\underline{\hat{C}_t}$ as the input to the regular MPC QoE maximization problem. Formally,

THEOREM 1. *The robust MPC controller is equivalent to the regular MPC taking the lower bound of throughput as input, namely,*

$$\begin{aligned} R_k &= f_{robustmpc}(R_{k-1}, B_k, [\underline{\hat{C}_t}, \overline{\hat{C}_t}]) \\ &= f_{mpc}(R_{k-1}, B_k, \underline{\hat{C}_t}) \end{aligned}$$

PROOF SKETCH. Conceptually, QoE function $QoE(R, C)$ can be written as the sum of 3 terms (g_1: total video quality, g_2: total quality change, g_3: rebuffer time), in which only the rebuffer time term depends on throughput C. Thus,

$$\begin{aligned} &\max_R \min_{C \in [\underline{C}, \overline{C}]} QoE(R, C) \\ \equiv &\max_R \left(g_1(R) - \lambda \times g_2(R) - \max_{C \in [\underline{C}, \overline{C}]} \mu \times g_3(R, C) \right) \\ \equiv &\max_R QoE(R, \underline{C}) \end{aligned}$$

As any decrease of throughput C will lead to longer rebuffer time, the minimum QoE is achieved at $C = \underline{C}$. □

The one potential downside is that robust MPC is more conservative than regular MPC by always assuming the lowest throughput. The degree of conservativeness here naturally depends on how loose/tight the lower bound is. In practice, we use maximum prediction error over the past several chunks as bounds in our implementation and find that it works well in practice (discussed in Section 7).

5 Using MPC in Practice — FastMPC

While rate-based and buffer-based algorithms need relatively minor computations, the challenge with MPC is that we need to solve a discrete optimization problem at each time step. There are two practical concerns here:

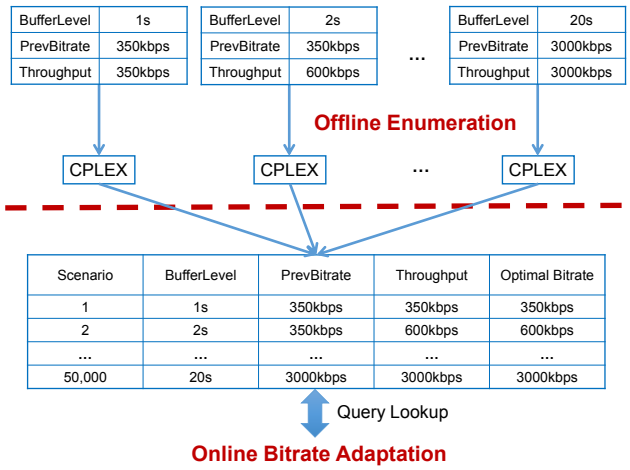


Figure 5: "FastMPC" idea: We enumerate possible scenarios and create a table indexing the optimal decision for each scenario.

- *Computational overhead:* First, the high computational overhead of MPC is especially problematic for low-end mobile devices, which are projected to be the dominant video consumers going forward. Since the bitrate adaptation decision logic is called before the player starts to download each chunk, excessive delay in the bitrate adaptation logic will negatively affect the QoE of the player.
- *Deployment:* Since we do not have a closed-form or combinatorial solution for the QoE maximization problem, we will need to use a solver (e.g., `CPLEX` or `Gurobi`). However, it may not be possible for video players to be bundled with such solver capabilities; e.g., licensing issues may preclude distributing such software or it may require additional plugin or software installations which poses significant barriers to adoption [26].

From the above discussion, it is evident that the solution we develop should be *lightweight* and *combinatorial* (i.e., not solving a LP or ILP online). As such, in this section, we address these two key practical issues by developing a fast and low-overhead FastMPC design that does not require any explicit solver capabilities in the video player [48].

5.1 High-Level Idea of FastMPC

At a high level, FastMPC algorithms essentially follow a table enumeration approach. Here, we do an offline step of enumerating the state-space and solve each specific instance. Then, in the online step we just use these stored optimal control decisions mapped to the current operation conditions. That is, the algorithm will be reduced to a simple table lookup indexed by the key value closest to the current state and the output of the lookup is the optimal solution for the selected configuration.

In our setting (Figure 5), the state-space is determined by the following dimensions: (1) current buffer level, (2) previous bitrates chosen, and (3) the predicted throughput for the next N chunks (i.e., the planning horizon). Thus, FastMPC will entail enumerating potential scenarios capturing differ-

ent values for each dimension and solving the optimization problems offline.

Unfortunately, directly using this idea will be very inefficient as we have a high dimensional state space. For instance, if we have 100 possible values for the buffer level, 10 possible bitrates, a horizon of size 5, and 1000 possible throughput values, there will be 10^{18} rows in the table![4] There are two obvious consequences of this large state space. First, it may not be practical to explicitly store the full table in the memory. Note that this is not just a hypothetical concern. If we need a practical implementation of this table lookup in the `dash.js` player [1] it will mean very high memory footprint along with large startup delay as the table needs to be downloaded to the player module. Second, it will incur a non-trivial offline computation cost that may need to be rerun as the operating conditions change.

5.2 Optimizing FastMPC Performance

Next, we present two key optimizations to make the table enumeration approach tractable.

Compaction via binning: First, to address the offline exploration cost, our insight is that we may not need very fine-grained values for the buffer and the throughput levels. As a consequence these values may be suitably coarsened into aggregate bins. Moreover, with binning we do not need to explicitly store the row keys as these are directly computed from the bin row indices. The challenge here is to balance the granularity of binning and the loss of optimality in practice. In practice, we find that using 100 bins for buffer level and 100 bins for throughput predictions works well and yields near-optimal performance.

Table compression: Our second insight is that the decision table learned by the offline computation will have significant structure. Specifically, the optimal solutions for several similar scenarios will likely be the same. Thus, we can exploit this structure in conjunction with the binning strategy to explore a simple lossless compression strategy using a run-length encoding to store the decision vector. The optimal decision can then be retrieved online using binary search. In practice, we see that with compression the table occupies less than 60 kB with 100 bins for buffer levels, 100 bins for throughput predictions and 5 bitrate levels.

6 Implementation

In this section, we describe our implementation of the MPC approach in the `dash.js` framework. Our implementation is based on the `dash.js` master branch (v1.2.0 release) as it was the stable version at the time of development. We believe that our implementation can be easily adapted to future versions as we require minimal modifications ($\approx$ 800 lines of JavaScript). For more information on the source code and demo please visit our demo page [14].

Choice of player: Many prior adaptive bitrate players were prototyped using the Adobe OSMF framework [34, 3, 12] and this seemed a natural choice. However, our conversations with industry personnel revealed that almost all content providers are switching to HTML5-based players based on the MPEG-DASH standard [16] and thus OSMF (based on Flash and with decreasing market share) is unlikely to be a platform with real-world impact. Having chosen a DASH player, we qualitatively evaluated several implementations of the DASH standard (e.g., [23, 8, 43]). Unfortunately, these rely either on custom clients or niche video player platforms. Given these considerations, we chose the `dash.js` framework as it is the reference open-source implementation for the MPEG-DASH standard and is actively supported by leading industry participants [7]. We believe our prototype efforts will also inform the evolution of these standardization efforts. For instance, a key requirement for any control algorithms is to know the size (in bytes) of each video chunk, but the standard does not mandate the *manifest* to report chunk sizes, which may be a key shortcoming of the current specification.

`dash.js` **overview:** To understand our implementation and modifications, we begin with some brief background on the architecture of the `dash.js` player. The key components are highlighted in Figure 6.

At a high level, the `dash.js` implementation separates high-level video streaming functionalities from low-level specific DASH standard related components. As we are not particularly interested in standard-specific implementation, we leave the code unmodified and only focus on the adaptive streaming related functions.

The classes and functions that are key to bitrate adaptation and video streaming logic are as follows:

- `BufferController`: This class provides functions to manage buffer levels of the player by requesting new segments and making bitrate change decisions. Specifically, function `validate` is periodically invoked and calls `getPlaybackQuality` function in `AbrController` class to find optimal bitrate. It also maintains a variable `bufferLevel` to record the current buffer occupancy of the player, which can be used for bitrate decisions.
- `AbrController`: This class contains the core bitrate adaptation logic. In the original `dash.js` implementation, a rule-based decision logic is employed to find the bitrate. Specifically, `DownloadRatioRule` selects bitrate based on the "download ratio" (play time of last chunk divided by its download time); On the other hand, `InsufficientBufferRule` chooses bitrate depending on whether the buffer level has reached a lower limit recently to avoid rebuffers. Priorities are assigned to each rule to resolve conflicts and make final bitrate decisions.

Modifications and extensions: We observed two implementation details in `dash.js` that were problematic. First, the code periodically calls the `validate` function to check the status of the buffer and call functions in `AbrController` to decide if the current bitrate should be changed. Note that this implies the bitrate decisions are not always made at chunk boundaries, which may lead to delay of execution of bitrate decisions, or even redownloading previous chunks.

[4] 100 buffer levels $\times$ 10 bitrates $\times$ 1000 throughput 1 values $\times \cdots \times$ 1000 throughput 5 values $= 10^{18}$ entries.

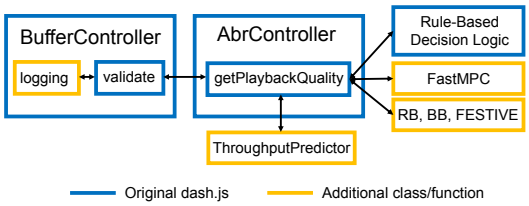


Figure 6: `dash.js` code structure and our modifications

Second, the `dash.js` downloads multiple chunks in parallel even though chunks that are earlier in the video stream should ideally be prioritized.

To address these concerns, we changed the bitrate decision and chunk download process in `dash.js` code by making two key changes to `BufferController` class: 1) bitrate decisions are made at the start of each chunk, 2) chunk download is completely sequential, i.e., no concurrent downloads of multiple chunks are allowed. This allows a basic implementation framework which is consistent with our model and other proposed algorithms.

With these fixes, we implemented different bitrate adaptation algorithms (e.g., FastMPC, BB, RB) by replacing the original rule-based bitrate adaptation logic by our own implementation. The FastMPC implementation has a static table that is used to index control decisions. We also implemented a harmonic mean based throughput prediction scheme based on prior work [34], as well as additional logging functions in the `BufferController` class to record a complete log of the state of the player, including buffer level, bitrates, rebuffer time, predicted/actual throughput.

7 Evaluation

In this section, we compare our approach against existing rate- and buffer-based approaches using a combination of real player and simulation experiments. We also present microbenchmarks on the CPU and memory overhead of our FastMPC implementation.

7.1 Setup

We begin by describing key parameters: (1) throughput variability traces; (2) video-specific parameters; (3) configurations for various adaptation algorithms; and (4) definition of a normalized QoE metric that we use throughout this section.

7.1.1 Input Parameters

Throughput traces: Our goal is to evaluate various bitrate adaptation approaches using realistic network variability conditions. Given the paucity of large-scale sustained throughput measurements over several tens of seconds, however, we use a combination of existing datasets and synthetic models:

1. *Broadband dataset (FCC) [9]:* The FCC dataset consists of more than 1 million sets of throughput measurements, where each set contains six data points each representing average throughput during a 5s interval. We extract throughput traces of the same server and client IP address and concatenate these to match the length of the video. For experiments we randomly pick 1000 of the concatenated traces whose average throughput is between 0 to 3Mbps, to avoid trivial cases where picking the maximum bitrate is always the optimal solution.
2. *Mobile dataset (HSDPA) [10]:* The HSDPA dataset consists of 30min of continuous 1s measurement of video streaming throughput of a moving device in Telenor's 3G/HSDPA mobile wireless network in Norway. We randomly pick 1000 throughput traces from the full dataset.
3. *Synthetic dataset:* Finally we also use a synthetic dataset to supplement the aforementioned datasets. The throughput is based on some hidden state $S_t \in \mathcal{S}$ modeling the number of users sharing a bottleneck link. The actual throughput C_t follows a Gaussian distribution with mean m_s and variance σ_s^2, given the value of hidden state $S_t = s$. We vary both the state transition probability matrix as well as the parameters m_s, σ_s^2 to generate traces.

Figure 7 shows the throughput characteristics of all three datasets. Among three datasets, throughput is the most stable in broadband network and the most variable in mobile network. In other words, the HSPDA dataset is a good stress test for our MPC approach that assumes the throughput is predictable on short timescales.

Video parameters: We use the "Envivio" video from DASH-264 JavaScript reference client test page [6] which is 260s long, consisting of 65 4s chunks. The video is encoded by H.264/MPEG-4 AVC codec in the following bitrate levels: $\mathcal{R} = \{350\text{kbps}, 600\text{kbps}, 1000\text{kbps}, 2000\text{kbps}, 3000\text{kbps}\}$. This is consistent with the requirement for YouTube video bitrate levels for 240p, 360p, 480p, 720p and 1080p respectively [15]. We set the buffer size to $B_{max} = 30s$. We assume $q(\cdot)$ is an identity function. As a default QoE function, we use the weights $\lambda = 1$, $\mu = \mu_s = 3000$, meaning 1-sec rebuffer/start-up time receives the same penalty as reducing the bitrate of a chunk by 3000 kbps. We also run sensitivity experiments that vary the QoE weights.

7.1.2 Algorithms and Metrics

Adaptation algorithms: Determining the optimal algorithm within each class is difficult as it involves optimizing over an infinite-dimensional functional space. To this end, we choose a widely adopted function form for each class of algorithms from prior work, and optimize the free parameters by empirical simulations based on a training dataset containing 100 throughput traces randomly picked across all datasets. We evaluate the following algorithms:

1. *RB:* The bitrate is picked as the maximum available bitrate which is less than $p = 1$ times throughput prediction using harmonic mean of past 5 chunks;
2. *BB:* We employ the function suggested by Huang et al [33], where bitrate R_k is chosen to be the maximum available bitrate which is less than $r_k = f(B_k)$ with reservoir $r = 5s$ and cushion $c = 10s$.
3. *FastMPC:* We use a look-ahead horizon $h = 5$ with throughput predictions using harmonic mean of past 5 chunks; We use 100 bins for throughput prediction and 100 bins for buffer level. We also evaluate the exact MPC with perfect throughput prediction for the next 5 chunks in simulations (denoted as *MPC-OPT*).

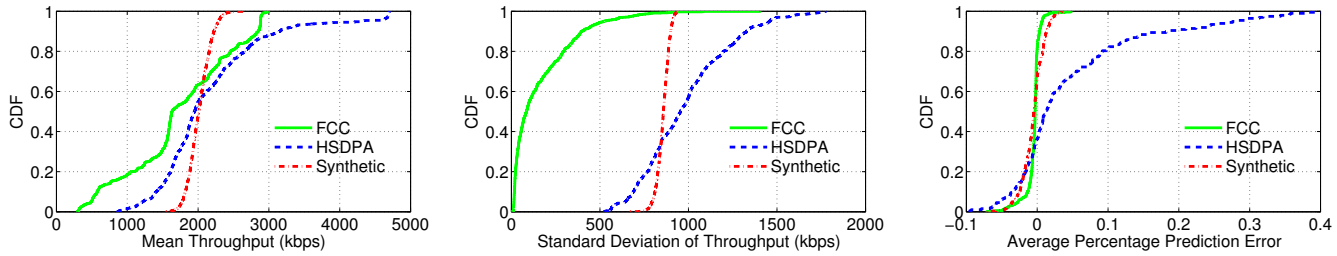
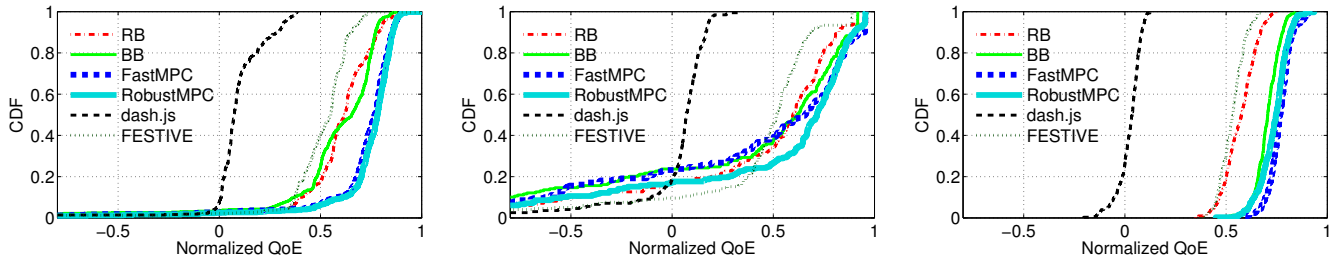


Figure 7: Characteristics of datasets



(a) FCC dataset (b) HSDPA dataset (c) Synthetic dataset

Figure 8: Real experiment results with different throughput traces

4. *RobustMPC:* We assume that the throughput lower bound is $\hat{C}_t = \hat{C}_t/(1+err)$, where $\hat{C}_t$ is obtained using harmonic mean of past 5 chunks, while prediction error err is the maximum absolute percentage error of the past 5 chunks.
5. `dash.js`*:* The original implementation adopts a rule-based bitrate decision logic as shown in Section 6. We keep the original bitrate adaptation logic unmodified, but disable the multi-chunk downloading and allow the bitrate to switch only at chunk boundaries.[5]
6. *FESTIVE [34]:* This rate-based algorithm balances both efficiency and stability, and incorporates fairness across players but that is not a concern in this paper. We assume there is no wait time between consecutive chunk downloads, and implement FESTIVE without the randomized chunk scheduling. Note that this does not negatively impact the player QoE in single player case. Specifically, FESTIVE calculates the efficiency score depending on $p = 1$ times throughput predictions using harmonic mean of past 5 chunks, as well as a stability score as a function of the bitrate switches in the past 5 chunks. The bitrate is chosen to minimize stability score plus $\alpha = 12$ times efficiency score.

Throughput predictor: Note that RB, *-MPC, and FESTIVE need a good throughput predictor. Developing good predictors for different scenarios is outside the scope of the paper. Building on insights from prior work, we use the harmonic mean of the observed throughput of the last 5 chunks because it is robust to outliers in per-chunk estimates [34]. We revisit this issue in Section 8.

[5]This enables a consistent comparison of the algorithms rather than conflate it with other artifacts because of parallel downloads. We also tested the original `dash.js` without any modification, but its performance is worse than our modified version (not shown).

Normalized QoE metric: We define a normalized QoE metric as follows. For a given throughput trace $\{C_t, t \in [t_1, t_{K+1}]\}$, the *offline optimal QoE*, denoted by $QoE(OPT)$, is the maximum QoE that can be achieved with perfect knowledge of future throughputs over the entire horizon. It can be calculated by solving problem $QOE\_MAX_1^K$ [6] and provides a theoretical upper bound of achievable QoE. On the other hand, a real *online* algorithm A selects bitrate R_k based on current throughput predictions $\{\hat{C}_t, t > t_k\}$ without knowing the entire future. We denote the online QoE achieved by algorithm A by $QoE(A)$ and define *normalized QoE of A (n-QoE(A))* for an algorithm A as: $n\text{-}QoE(A) = \frac{QoE(A)}{QoE(OPT)}$.

7.2 Real Player Evaluation

First, we present emulations with the real player setup comparing our FastMPC approach against several prior approaches. Our basic experiment setup consists of two computers (Ubuntu 12.04 LTS) with a 100Mbps direct network connection emulating a video client and server. The video client is a Google-Chrome web browser for linux (version 39) with V8 JavaScript engine while the video server is a simple HTTP server based on `node.js` (version 0.10.32). We use the linux `tc` tool to throttle the throughput of the link between two computers according to the throughput traces employed. We use Emulab [49] to carry out several such experiments in parallel.

Figure 8 show the CDF of normalized QoE over the three sets of throughput traces. First, we see that existing algorithms achieve only 60-70% of optimal QoE confirming that there is still large room to improve video QoE. Second, RobustMPC outperforms non-MPC algorithms in all datasets with an improvement in median normalized QoE of 15%, 10%, and 5% in the FCC, HSDPA, and Synthetic datasets

[6]To make it tractable to compute this offline optimal, we assume it can pick bitrates from a continuous range $[R_{min}, R_{max}]$.

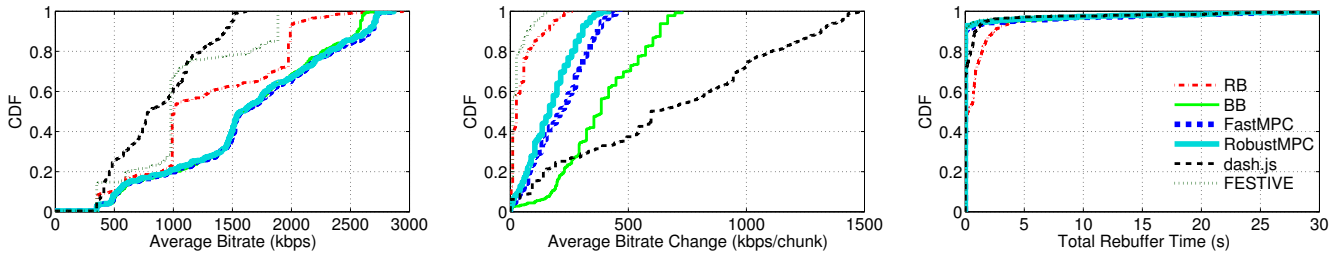


Figure 9: Detailed performance for FCC dataset

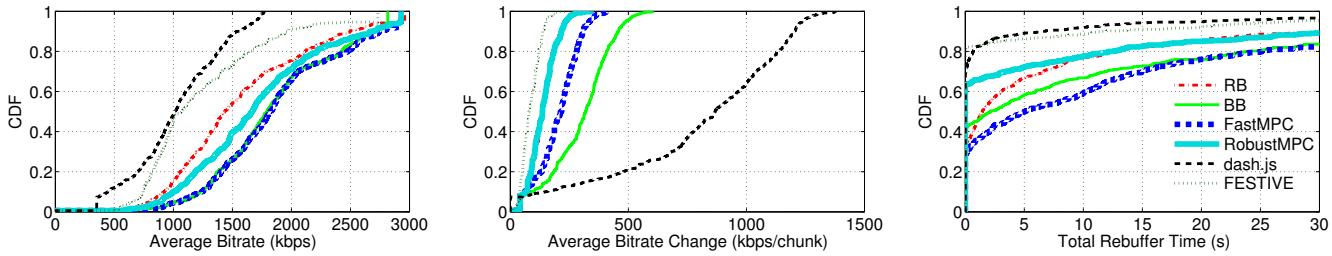


Figure 10: Detailed performance for HSDPA dataset

respectively. Third, we see significant improvement (60+% median normalized QoE) compared with the original `dash.js` player. Finally, we see that the basic FastMPC is more sensitive to prediction errors than RobustMPC. While there is no difference between Fast- and RobustMPC on FCC and Synthetic results, the difference is especially visible in the HSPDA result where regular FastMPC suffers and presents no gains versus RB and BB.

To better understand the impact of prediction error, Figure 7 shows the CDF of per-session average percentage prediction errors for the datasets. In FCC dataset, the average error of our harmonic mean throughput predictor is less than 5%, while in HSPDA dataset, the worst-case prediction error can be as high as 40%. We also observe that the predictor over-estimates the true throughput for more than 20% of the time in HSPDA dataset which leads to significant rebuffering. As such, inaccurate prediction can ruin the decision making of regular FastMPC, while RobustMPC is less affected as it incorporates prediction error to avoid choosing bitrate too aggressively when predictions are inaccurate.

The earlier normalized QoE result shows the aggregate combination of different QoE factors. Next, we zoom in on the individual quality factors to explain the QoE improvements in Figures 9 and 10. In the FCC dataset, all algorithms achieve similarly low rebuffer time as throughput is predictable. The performance difference essentially stems from reducing unnecessary bitrate switches. RobustMPC, FastMPC and BB achieve similar average bitrates, but RobustMPC uses fewer bitrate switches. In the HSPDA result, rebuffer time becomes a more important issue. While FastMPC achieves similar average bitrate and fewer switches comparing to BB, it suffers from large rebuffer time. On the other hand, RobustMPC achieves significant less rebuffer time but at a slightly lower average bitrate: Zero rebuffer in 65% of all cases, versus 40% for BB and FastMPC. As a result, RobustMPC still outperforms other algorithms in overall QoE.

Doing a cross-dataset analysis, we see that the tail distributions of the overall QoE show different characteristics. In the FCC result, only 1% users experience normalized QoE <0 while in HSPDA this occurs in 10% of all cases.[7] Again, the main reason is that the high variability of mobile network induces long rebuffering which affects the overall QoE.

Finally, even though FESTIVE is a rate-based algorithm, it performs slightly worse than regular RB in our datasets because it puts a higher weight on stability and switches up bitrate slowly even when the available throughput is increasing.[8] On the other hand, the `dash.js` heuristic rule-based adaptation achieves low rebuffer time, but incurs many unnecessary switches. Thus, its overall QoE is significantly worse than all algorithms.

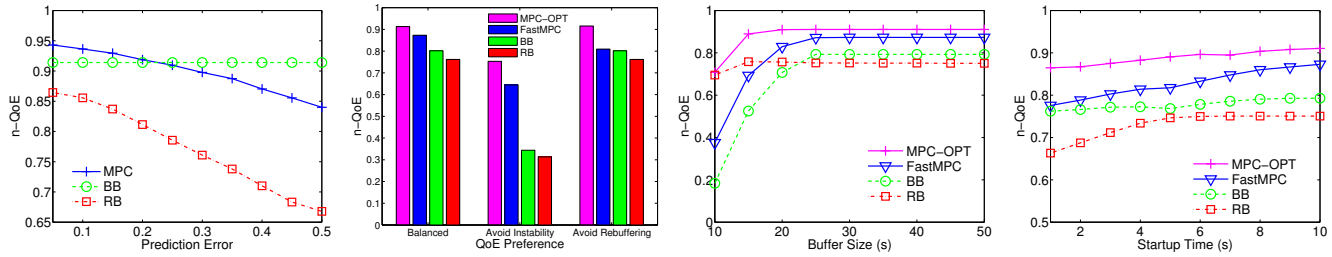
7.3 Sensitivity Analysis

For sensitivity analysis we evaluate different algorithms using a custom simulation framework. As before, the simulation takes as input a throughput trace and models the video download/playback process and the buffer dynamics. At time t_k when the bitrate of chunk k is needed, the simulation calls the bitrate controller embedded with different algorithms to get R_k. Using this framework, we study the sensitivity of the approaches to key factors such as: (1) prediction error, (2) choice of QoE function, (3) playout buffer size, (4) number of bitrate levels, and (5) startup delay.

Throughput prediction: Here, we want to study the impact of prediction error of general predictors rather than analyze a particular one (e.g., harmonic mean). To this end, we use the average error level to characterize the performance of a throughput predictor and model the prediction output as being a combination of the true throughput with added random noise according to the average error level. Figure 11a shows

[7]The QoE can be negative when rebuffer time is too long or there are too many switches.

[8]This is not a flaw, but a deliberate choice for achieving multi-player fairness [34].



(a) Prediction error (b) QoE preferences (c) Buffer size (d) Startup time

Figure 11: Sensitivity analysis vs. operating conditions

how the throughput prediction errors influence the performance of bitrate adaptation algorithms. As expected, BB is unaffected as it does not use any throughput information. When throughput predictions are accurate, MPC has larger advantage over BB algorithms. As prediction error grows beyond 25%, MPC can be even worse than BB. This suggests that if the actual prediction error is very large, then the video player should drop RB or MPC and use pure BB algorithms. In contrast with regular MPC, robust MPC is less affected by prediction error as it takes into possible error into account and maximizes the worst case QoE.

Users' QoE preferences: We compared the performance of the algorithms under 3 sets of QoE weights, "Balanced" ($\lambda = 1, \mu = \mu_s = 3000$), "Avoid Instability" ($\lambda = 3, \mu = \mu_s = 3000$), "Avoid Rebuffering" ($\lambda = 1, \mu = \mu_s = 6000$). As shown in Figure 11b, as users put more penalty weights to bitrate instability, the MPC algorithms show more advantage over RB and BB. This is because MPC algorithms explicitly model the bitrate vs. bitrate instability tradeoff in the QoE function, while RB and BB do so in ad-hoc ways. However, when rebuffering time is a more important factor, BB algorithms perform similarly with FastMPC algorithms because of two key reasons. First, BB algorithms keep a minimum buffer level so that the player has a better chance surviving low throughput with less/no rebuffering time. Second, while MPC algorithms do a good job with perfect throughput prediction, they can suffer from long rebuffering time since harmonic mean predictor is imperfect. As such, MPC can be improved by maintaining a minimum buffer level and employing a more accurate predictor.

Buffer size and startup delay: Figure 11c analyzes the impact of playout buffer size. First, when buffer size is small (<25s in play time), increasing buffer size improves the performance of all algorithms. A larger buffer protects the player against rebuffering events and also provides more degrees of freedom to optimize performance. As buffer size reaches a certain level (25s of play time), the performances of all algorithms stay constant even buffer size is further increased. Finally, RB is the least affected by buffer size because it does not consider buffer level in its decision logic.

While our approach optimizes startup delay automatically, we analyze how overall QoE (except the startup delay term) is affected if the startup delay is fixed. As shown in Figure 11d, as startup time increases, the performance of all algorithms improves, as the player accumulates more video in the buffer at the start-up phase making it easier to manage rebuffering events.

Bitrate levels: We also study how number of bitrate levels influences the performance (not shown). With BB and MPC, we can achieve better performance using finer-grained set of bitrate levels. With RB, however, the performance of RB first improves as we add more bitrate levels, but decreases when there are too many bitrate levels. The reason is that RB starts changing bitrate more frequently, leading to increased bitrate instability. One caveat with MPC is that finer-grained bitrate levels also require more discretization levels for the FastMPC implementation. Understanding this tradeoff is an interesting direction for future work.

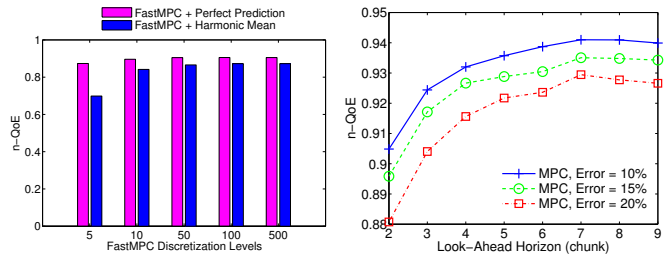
7.4 MPC Configuration and Overhead

Overhead: As discussed earlier, FastMPC might increase player overhead relative to BB and RB style algorithms. We compare the CPU and memory usage of our implementation of FastMPC, BB, and RB algorithm with the default `dash.js` player. We find that FastMPC, BB, and RB all consume similar amount of CPU, while FastMPC uses only 60 kB more memory (not shown).

FastMPC discretization: Recall that the number of discretization levels is an important design parameter for FastMPC. More discretization levels increase FastMPC performance but require more player memory and may also increase startup delay. We study this performance vs. overhead tradeoff in Figure 12a and Table 1. From Figure 12a, we see that more discretization levels imply larger performance gains for FastMPC but the improvement shows diminishing return; e.g., FastMPC achieves 90% of optimal QoE with 100 levels while this drops to 70% if there are only 5 levels. Second, the gain vs. discretization level also has some dependency on the throughput predictor especially with very coarse discretization. Table 1 shows that while the memory overhead

Discretization levels	Extra JavaScript code size	
	Full table	Run length coding
50	25.0 kB	19.1 kB
100	100 kB	56.4 kB
200	400 kB	141 kB
500	2.50 MB	451 kB

Table 1: FastMPC table size



(a) Discretization (b) Look-ahead horizon

Figure 12: MPC configuration parameters

increases with more levels, the simple compression scheme we discussed earlier can reduce the memory overhead especially when number of levels is large. For instance, with 100 levels the compression rate is 0.5 while with 500 levels it can reduce the table size by 82%. Even with 500 levels, the table size is quite reasonably low.

Look-ahead horizon: Figure 12b shows how planning horizon impacts the performance of MPC algorithms. As the look-ahead horizon increases, MPC performances grow and stay stable since more information of future throughput is taken into account. However, as we look further into the future, prediction accuracy can reduce. The performance of MPC can even drop if the horizon is too large.

7.5 Summary of Results

Our main findings are summarized as follows:

1. RobustMPC outperforms existing algorithms in both broadband (FCC) and cellular (HSDPA) datasets, while regular FastMPC does not show advantage in cellular network due to high throughput instability;
2. Our implementation of FastMPC algorithm incurs very low overhead: near-zero CPU overhead and 60 kB increase in memory usage compared to original `dash.js`;
3. Sensitivity analysis shows that FastMPC has advantages over BB and RB in wide parameter ranges. However, there is still room for improvement by increasing FastMPC discretization granularity and employing more accurate throughput predictors.

8 Discussion

Before concluding, we revisit two outstanding issues.

Multi-player effects: In this paper, we focused purely on improving the design of a single video player. A natural question is to extend these insights to multiple players and interaction with cross traffic [34, 32]. To fully consider multi-player interaction and fairness, we can extend our control-theoretic model to explicitly consider a fairness term in the QoE function and model the effects of TCP on throughput allocation. For instance, we might be able to reason about fairness from the perspective of game theory or distributed control theory in this context. This is an interesting direction for future research.

Throughput prediction: As observed by other researchers, better throughput prediction can improve video performance in cellular networks [52]. A key limitation of our work is that we do not have accurate algorithms for throughput prediction and the literature is surprisingly scarce and dated [30, 51, 20]. Two interesting directions of future work are in using crowdsourced approaches based on measurements from other clients [41] and developing a better understanding of throughput predictability and stability in the wild.

9 Conclusions

Our paper was motivated by recent debates surrounding the design of dynamic adaptive streaming over HTTP (DASH) algorithms. To bring some rigor to this space, we developed a control-theoretic problem formulation that allowed us to explore the design space systematically and evaluate quantitatively different classes of solutions through well-defined QoE metrics. With the key insights that a broader design space is available compared to existing solutions, we designed and implemented a model predictive control approach to optimally combine buffer occupancy and throughput predictions in order to maximize the user's QoE. We demonstrated a practical implementation of MPC using the `dash.js` reference video player. Our trace-driven emulations using realistic throughput variability traces confirmed the advantages over state of the art solutions in a wide range of operating conditions with negligible increase in computation and memory requirements. As future work, we plan to incorporate more accurate throughput predictions and explicitly capture multi-player interactions.

Acknowledgments

This work was supported in part by the National Science Foundation under awards ECCS-0925964 and CNS-1345305. We thank our shepherd Keith Winstein for helping us improve the final version. We thank Aditya Ganjam and David Oran for useful discussions regarding industry player platforms that informed our implementation.

10 References

[1] Dash-Industry-Forum, dash.js . https://github.com/Dash-Industry-Forum/dash.js/wiki.
[2] Adobe HTTP Dynamic Streaming. www.adobe.com/products/hds-dynamic-streaming.html.
[3] Adobe OSMF player. http://www.osmf.org.
[4] Akamai HD network. www.akamai.com/hdnetwork.
[5] Apple's HTTP Live Streaming. https://developer.apple.com/streaming/.
[6] DASH-264 JavaScript reference client landing page 1.4.0. http://dashif.org/reference/players/javascript/1.4.0/samples/dash-if-reference-player/index.html.
[7] DASH Industry Forum members. http://dashif.org/members/.
[8] DASH VLC plugin. http://www-itec.uni-klu.ac.at/dash/?page_id=10.
[9] FCC dataset. https://www.fcc.gov/measuring-broadband-america. Accessed: 2014-12-01.

[10] HSDPA dataset. http://home.ifi.uio.no/paalh/dataset/hsdpa-tcp-logs. Accessed: 2014-12-01.

[11] Netflix. http://www.netflix.com/.

[12] OSMF 2.0 release code. http://sourceforge.net/projects/osmf.adobe/files/latest/download.

[13] Smooth Streaming protocol. http://go.microsoft.com/?linkid=9682896.

[14] The demo page for our MPC-based bitrate adaptation. http://users.ece.cmu.edu/~vsekar/mpcdash.html.

[15] YouTube live encoder settings, bitrates and resolutions. https://support.google.com/youtube/answer/2853702?hl=en.

[16] I. Sodagar. The MPEG-DASH Standard for Multimedia Streaming Over the Internet. *IEEE Multimedia*, 2011.

[17] S. Akhshabi, L. Anantakrishnan, C. Dovrolis, and A. C. Begen. What Happens when HTTP Adaptive Streaming Players Compete for Bandwidth? In *Proc. NOSSDAV*, 2012.

[18] S. Akhshabi, L. Ananthakrishnan, A. Begen, and C. Dovrolis. Server-Based Traffic Shaping for Stabilizing Oscillating Adaptive Streaming Players. In *Proc. ACM SIGMM NOSSDAV*, 2013.

[19] A. Balachandran, V. Sekar, A. Akella, S. Seshan, I. Stoica, and H. Zhang. Developing a Predictive Model of Quality of Experience for Internet Video. In *Proc. ACM SIGCOMM*, 2013.

[20] H. Balakrishnan, M. Stemm, S. Seshan, and R. H. Katz. Analyzing Stability in WideArea Network Performance. In *Proc. ACM SIGMETRICS*, 1997.

[21] D. P. Bertsekas, D. P. Bertsekas, D. P. Bertsekas, and D. P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 1. Athena Scientific Belmont, MA, 1995.

[22] E. F. Camacho and C. B. Alba. *Model Predictive Control*. Springer, 2013.

[23] L. D. Cicco, V. Caldaralo, V. Palmisano, and S. Mascolo. TAPAS: a Tool for rApid Prototyping of Adaptive Streaming algorithms. In *Proc. CoNext VideoNext workshop*, 2014.

[24] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. A. Joseph, A. Ganjam, J. Zhan, and H. Zhang. Understanding the Impact of Video Quality on User Engagement. In *Proc. ACM SIGCOMM*, 2011.

[25] G. F. Franklin, J. D. Powell, and M. L. Workman. *Digital Control of Dynamic Systems*, volume 3. Addison-wesley Menlo Park, 1998.

[26] A. Ganjam, F. Siddiqui, J. Zhan, X. Liu, I. Stoica, J. Jiang, V. Sekar, and H. Zhang. C3: Internet-Scale Control Plane for Video Quality Optimization. In *Proc. NSDI*, 2015.

[27] M. Ghobadi, Y. Cheng, A. Jain, and M. Mathis. Trickle: Rate Limiting YouTube Video Streaming. In *Proc. USENIX ATC*, 2012.

[28] S. Gouache, G. Bichot, A. Bsila, and C. Howson. Distributed and Adaptive HTTP Streaming. In *Proc. ICME*, 2011.

[29] D. Havey, R. Chertov, and K. Almeroth. Receiver Driven Rate Adaptation for Wireless Multimedia Applications. In *Proc. MMSys*, 2012.

[30] Q. He, C. Dovrolis, and M. Ammar. On the Predictability of Large Transfer TCP Throughput. In *Proc. ACM SIGCOMM*, 2005.

[31] R. Houdaille and S. Gouache. Shaping HTTP Adaptive Streams for a Better User Experience. In *Proc. MMSys*, 2012.

[32] T.-Y. Huang, N. Handigol, B. Heller, N. McKeown, and R. Johari. Confused, Timid, and Unstable: Picking a Video Streaming Rate is Hard. In *Proc. IMC*, 2012.

[33] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson. A Buffer-Based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *Proc. ACM SIGCOMM*, 2014.

[34] J. Jiang, V. Sekar, and H. Zhang. Improving Fairness, Efficiency, and Stability in HTTP-based Adaptive Video Streaming with FESTIVE. In *Proc. CoNext*, 2012.

[35] S. S. Krishnan and R. K. Sitaraman. Video Stream Quality Impacts Viewer Behavior: Inferring Causality using Quasi-Experimental Designs. In *Proc. IMC*, 2012.

[36] R. Kuschnig, I. Kofler, and H. Hellwagner. Evaluation of HTTP-based Request-Response Streams for Internet Video Streaming. *Multimedia Systems*, pages 245–256, 2011.

[37] L. De Cicco, S. Mascolo, and V. Palmisano. Feedback Control for Adaptive Live Video Streaming. In *Proc. of ACM Multimedia Systems Conference*, 2011.

[38] Z. Li, X. Zhu, J. Gahm, R. Pan, H. Hu, A. Begen, and D. Oran. Probe and Adapt: Rate Adaptation for HTTP Video Streaming at Scale. *Selected Areas in Communications, IEEE Journal on*, 32(4):719–733, 2014.

[39] C. Liu, I. Bouazizi, and M. Gabbouj. Parallel Adaptive HTTP Media Streaming. In *Proc. ICCCN*, 2011.

[40] H. Liu, Y. Wang, Y. R. Yang, A. Tian, and H. Wang. Optimizing Cost and Performance for Content Multihoming. In *Proc. ACM SIGCOMM*, 2012.

[41] X. Liu, F. Dobrian, H. Milner, J. Jiang, V. Sekar, I. Stoica, and H. Zhang. A Case for a Coordinated Internet Video Control Plane. In *Proc. ACM SIGCOMM*, 2012.

[42] R. K. P. Mok, X. Luo, E. W. W. Chan, and R. K. C. Chang. QDASH: A QoE-aware DASH system. In *Proc. MMSys*, 2012.

[43] C. Mueller, S. Lederer, J. Poecher, and C. Timmerer. Libdash - An Open Source Software Library for the MPEG-DASH Standard. In *Proc. ICME*, 2013.

[44] L. Popa, A. Ghodsi, and I. Stoica. HTTP as the Narrow Waist of the Future Internet. In *Proc. HotNets*, 2010.

[45] R. Rejaie and J. Kangasharju. Mocha: A Quality Adaptive Multimedia Proxy Cache for Internet Streaming. In *Proc. NOSSDAV*, 2001.

[46] S. Akhshabi, A. Begen, C. Dovrolis. An Experimental Evaluation of Rate Adaptation Algorithms in Adaptive Streaming over HTTP. In *Proc. MMSys*, 2011.

[47] G. Tian and Y. Li. Towards Agile and Smooth Video Adaption in Dynamic HTTP Streaming . In *Proc. CoNext*, 2012.

[48] Y. Wang and S. Boyd. Fast Model Predictive Control using Online Optimization. *Control Systems Technology, IEEE Transactions on*, 18(2):267–278, 2010.

[49] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. *Proc. OSDI*, 2002.

[50] X. Yin, V. Sekar, and B. Sinopoli. Toward a Principled Framework to Design Dynamic Adaptive Streaming Algorithms over HTTP. In *Proc. ACM SIGCOMM HotNets*, 2014.

[51] Y. Zhang and N. Duffield. On the Constancy of Internet Path Properties. In *IMW*, 2001.

[52] X. K. Zou, J. Erman, V. Gopalakrishnan, E. Halepovic, R. Jana, X. Jin, J. Rexford, and R. K. Sinha. Can Accurate Predictions Improve Video Streaming in Cellular Networks? In *Proc. ACM HotMobile*, 2015.