2018 IEEE 26th International Conference on Network Protocols

HotDASH: Hotspot Aware Adaptive Video Streaming using Deep Reinforcement Learning

Satadal Sengupta*, Niloy Ganguly†, Sandip Chakraborty‡
Dept. of Computer Science and Engineering
IIT Kharagpur, Kharagpur, India
*satadal.sengupta@iitkgp.ac.in, {†niloy, ‡sandipc}@cse.iitkgp.ernet.in

Pradipta De
College of Engineering and Computing
Georgia Southern University, Georgia, USA
pde@georgiasouthern.edu

***Abstract*—A large fraction of video content providers have adopted adaptive bitrate streaming over HTTP. The client player typically runs an adaptive bitrate (ABR) algorithm to decide upon the most optimal quality for the next few seconds of video playback. State-of-the-art ABR algorithms attempt to achieve an optimal trade-off among the competing objectives of high bitrate, less rebuffering, and high smoothness, in the face of unpredictable bandwidth variability. However, optimal bandwidth utilization does not necessarily ensure high quality of experience (QoE). Different users have different content preferences even within the same video, due to differences in team loyalties (in sport), character preferences (in movies and soaps), and so on. In this work, we present HotDASH, a system which enables opportune prefetching of user-preferred temporal video segments (called *hotspots*). HotDASH is powered by an optimal prefetch and bitrate decision engine, and is implemented using a prefetch module in the open source DASH player *dash.js*. The decision engine is designed as a cascaded reinforcement learning (RL) model, implemented using the state-of-the-art actor-critic RL algorithm A3C over a neural network. We train the neural network using trace-driven simulations over a large variety of bandwidth conditions. HotDASH outperforms all baseline algorithms, with a 16.2% QoE improvement over the best-performing baseline, and achieves 14.31% better average bitrate due to its ability to prefetch opportunistically.**

***Index Terms*—hotspot aware video streaming, deep reinforcement learning**

I. Introduction

Traffic from HTTP-based video streaming applications accounts for the largest share of the global Internet traffic in recent years [1]. Simultaneously, the user demand for video content with high Quality of Experience (QoE) has grown rapidly. The inability to provide satisfactory QoE to users translates into heavy losses for the service providers. However, complications arise from the fact that different users experience the same video differently. Numerous studies have shown that the quality of video experience for a user may depend on a variety of different factors, including gender, age, community, content, and so on [2]–[5]. Naturally, there arises a need for personalized video experience.

Video service providers primarily use *Dynamic Adaptive Streaming over HTTP* (DASH), to serve content to their subscribers. The target video is typically broken into *chunks* of fixed playing time, with each chunk stored at different qualities[1]. The client-side video player attempts to estimate the available bandwidth, based on the current playing conditions (in terms of throughput, buffer state, and so on) and employs adaptive bitrate (ABR) algorithms to choose an optimal quality for the next chunk. However, problems arise when the estimation is inaccurate; if a high-quality chunk is requested by the client in the face of insufficient bandwidth, the playback stalls for a few seconds (until the chunk is downloaded completely), a phenomenon known as *rebuffering*. Besides rebuffering, abrupt changes in the quality of consecutive chunks also hinder QoE; streaming algorithms, therefore, try to diminish the extent of abrupt changes, and thereby increase playback *smoothness*. Obviously, the primary requirement still remains to stream the video at the highest quality possible. State-of-the-art ABR algorithms, e.g. MPC [6] and Pensieve [7], therefore optimize for an objective function (or reward), which comprises three competing components: (1) high *bitrate*, (2) less *rebuffering*, and (3) high *smoothness*. The result is a session of video playback, where some temporal segments (chunks) are streamed at a higher quality than others, with the variation in quality closely following that of the available bandwidth.

While the aforementioned scenario is the most optimal in terms of bandwidth usage, to an unsuspecting user, the quality variability may seem random, and less than satisfactory. For example, let us consider a graduate student Bob, who is also an avid sports fan, and supports the team Panthers. Unfortunately, he misses out on watching the tournament final between Panthers and Eagles (another team), due to an impending deadline. Finally getting an opportunity, he arrives at the nearest café, connects his smartphone to the WiFi network, and starts watching the much anticipated highlights. The WiFi connection being unreliable, he ends up with a video experience where the quality is lower during scoring opportunities of Panthers, even though the streaming algorithm did its best in utilizing the available WiFi bandwidth.

The requirement, therefore, is for a video streaming strategy which takes into account content preferences of users, in addition to optimal use of bandwidth. More specifically, the streaming algorithm must be aware of high-priority temporal

[1]The terms *quality*, *bitrate*, and *resolution* are often synonymously used in this context, even though they are technically different (albeit related). In this paper, we use these terms synonymously, unless otherwise specified.

978-1-5386-6043-0/18/$31.00 ©2018 IEEE
DOI 10.1109/ICNP.2018.00026

IEEE computer society

segments in the video – which we call *hotspots* in this paper – and optimize video delivery accordingly. Existing works in the literature [8]–[10] have focused on identifying and extracting important video segments based on user personalization. However, merely forcing the player to download hotspots in higher quality does not suffice; the rebuffering time can increase significantly if high quality chunks are downloaded in the face of declining bandwidth (§V). Only a streaming algorithm, which makes opportune use of higher bandwidth, and a buffer with sufficient content waiting to play, will be able to serve hotspots at a satisfactory quality, without disregarding the competing objectives of low rebuffering time and high smoothness. Note that this requirement is different from *foveated rendering* [11], where spatially important parts of a video (more specifically, images combining to form a video) are rendered in a higher resolution.

In this paper, we propose HotDASH (abbreviation for hotspot-aware DASH), a system which delivers video content in a hotspot-aware manner. HotDASH circumnavigates around the requirement of forceful high-quality downloads, by implementing *opportune prefetching* of hotspot video chunks. *Prefetching* refers to the out-of-order download of a particular video chunk. Prefetching is *opportune* when the bandwidth is sufficiently high to download a high bitrate version of the chunk, and the buffer has sufficient content to not get completely depleted (thus causing rebuffering) during the additional delay caused due to prefetching. Additionally, it is important to prefetch in a manner such that other (regular/non-hotspot) video chunks are still downloaded at satisfactory bitrates, and playback smoothness does not diminish excessively. In a nutshell, *opportune prefetching* is ideally equivalent to a reordering of optimal bitrates (as yielded by state-of-the-art ABR algorithms) among hotspot chunks and regular chunks, such that the hotspot chunks end up with the higher bitrates, without causing rebuffering or lack of smoothness (as far as possible). Since state-of-the-art bitrate adaptation algorithms treat every video chunk with equal importance, they lack the ability to prioritize download of certain chunks over others. Even if prefetching is enabled, these algorithms would be unable to optimally decide when to prefetch, such that the aforementioned objectives are addressed.

We address three important challenges through HotDASH:

1) **On-demand prefetch in DASH:** We solve the technical challenge of enabling a DASH client to request out-of-order hotspot chunks, by implementing a custom prefetch module in the open source DASH player *dash.js* [12].
2) **Identifying hotspot prefetch opportunities:** We address the research challenge of intelligently detecting opportune moments for hotspot chunk downloads, by implementing a cascaded neural network model which learns through *experience* (i.e., performs *reinforcement learning (RL)*). HotDASH ensures highest possible quality for hotspot chunks under current bandwidth conditions.
3) **Achieving a balance between the hotspot quality and other QoE objectives:** The RL model further ensures that prefetching hotspot chunks does not lead to rebuffering, or inordinate decline in quality of regular chunks and in playback smoothness, as far as possible.

The prefetch decision engine, which is at the core of HotDASH, does away with the requirement of pre-programmed control rules, or carefully supervised strategies, which seldom generalize well. Instead, during the training phase, it starts with zero assumptions regarding the operating environment of the video player. As it takes decisions regarding when (if at all) to prefetch hotspot chunks, and at what quality, it receives a reward for every such decision. The reward is same as the objective function mentioned before (optimizing for high *bitrate*, less *rebuffering*, and high *smoothness*), with the adjustment that high bitrate decisions for hotspot chunks are awarded more than those for regular chunks (§III-D1). The obtained reward reinforces the model to make decisions which maximize the cumulative reward, thus ensuring user satisfaction without compromising on bandwidth utilization. While using deep reinforcement learning (RL) for video streaming has been proposed before by Pensieve [7], our novelty lies in the unique cascaded RL model, whereby we leverage Pensieve's model for bitrate decisions, and train our own neural network for prefetching decisions (§II).

The HotDASH decision engine is trained using A3C, which is a state-of-the-art actor-critic RL algorithm [13]. We perform training over a large corpus of bandwidth traces, available online. The training is hastened with the use of a trace-driven simulation environment, which faithfully models live video playback sessions. Once trained, the model is put up as a service, running on a separate server. The DASH client periodically sends the playback state (in terms of throughput, buffer, sizes of upcoming regular and hotspot chunks, etc.) to this service, and receives a prefetch decision, as well as the optimal bitrate for the next chunk. We evaluate HotDASH by comparing against 6 state-of-the-art ABR algorithms, under a wide variety of network traces. HotDASH outperforms all the baseline algorithms, achieving an average improvement in QoE of 16.2% over the best performing baseline, Buffer Based [14]. It also outperforms Pensieve [7] by 30%, and two versions of MPC [6] (robustMPC and fastMPC) by 32.6% and 67.4% in terms of average QoE, respectively. In terms of average bitrate, HotDASH achieves 14.31% improvement over Pensieve [7].

II. System Overview

In this section, we first provide a background of video streaming using DASH, and introduce our system HotDASH in that context.

A. Background

HTTP-based adaptive video streaming (standardized as DASH) is the delivery solution of choice for most video service providers in recent times. Fig. 1 illustrates the working of a client DASH player: the ABR controller in the player receives playing conditions in terms of throughput (from the throughput estimator) and buffer size (from the buffer controller). It then determines an optimal bitrate for the next

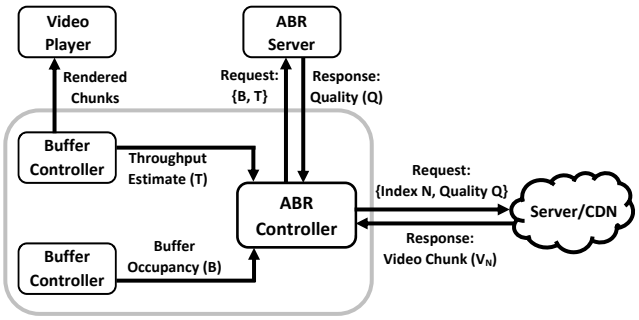


Fig. 1: Adaptive bitrate streaming using DASH in state-of-the-art approaches: The ABR controller in the client player relays playback conditions (thoughput, buffer occupancy, etc.) to an external ABR server, which responds with an optimal bitrate for the next chunk.

chunk, and initiates download from the corresponding CDN (or content server).

Certain state-of-the-art ABR algorithms require running expensive computations for optimal bitrate selection (e.g., Pensieve [7], which trains a neural network for bitrate decisions). While an in-player implementation is possible, device resource limitations (e.g., in smartphones and smart TVs) may hinder performance of the client player. The complex computation may be pushed to an additional stateless ABR server in such cases, as shown in Fig. 1. Being stateless, the ABR server is able to serve requests from multiple clients with different playing conditions simultaneously, with no additional overhead. Furthermore, the performance of ABR algorithms has been found to remain largely unaffected by the latency introduced due to player-ABR server packet exchanges [7]. Keeping these advantages in mind, our proposed system HotDASH employs an ABR server to handle bitrate selection and prefetch decisions, using an RL model as outlined in §I.

B. HotDASH: System Overview

HotDASH consists of two major components (as shown in Fig. 2): (1) *hotdash.js*, which is a prefetch-enabled version of the open source DASH player dash.js [12], and (2) the *HotDASH decision engine*, which acts as the ABR server (as mentioned in §II-A). After a video chunk download is complete, hotdash.js prepares a set of parameters (e.g., last chunk bitrate, throughput, buffer occupancy, next hotspot and regular chunk size, etc.), which combine together to form the *playback state information*. This information is then sent to the external ABR server *HotDASH decision engine*, in the form of a HTTP request. The decision engine responds with two outputs, i.e., a prefetch decision (yes/no), and the optimal bitrate for the next chunk. hotdash.js initiates download of the corresponding chunk accordingly. In the following section, we delve deeper into the design of the HotDASH decision engine.

III. HotDASH Decision Engine

In this section, we describe the design considerations which influenced the design of the *HotDASH decision engine*, and the reinforcement learning (RL) model which is employed to enable optimal prefetch and bitrate decisions. We recall from

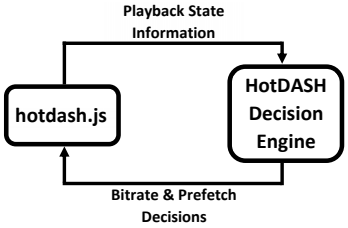


Fig. 2: HotDASH overview: Our system consists of 2 components, i.e., the prefetch-enabled player *hotdash.js*, and the *HotDASH decision engine*, which computes optimal bitrate and prefetch decisions.

§II-B that the decision engine is entrusted with the following responsibilities during a chunk download opportunity: (i) decide whether or not to prefetch the next hotspot chunk, (ii) decide the bitrate of the next hotspot chunk if prefetch decision is *yes*, and (iii) decide the bitrate of the next regular chunk if prefetch decision is *no*.

A. Prefetch Decision Considerations

The decision involving prefetch is non-trivial, since indiscriminate prefetching may lead to abnormally high rebuffering time. A mistimed prefetch may make the player wait for two chunk intervals – the prefetched chunk and the next in-sequence chunk – before resuming play. Again, if prefetch happens when the bandwidth is low, hotspot chunks end up getting lower bitrates, which defeats the purpose of our system. The bitrate at which prefetch is possible at the current instance is also important (if bitrate would be low, prefetching would result in no additional benefit). In a nutshell, the decision-making at a particular instance of time depends on the following factors: (1) current environment under which the player is operating, which involves bandwidth and buffer conditions, (2) detailed knowledge regarding the upcoming video chunks (in terms of chunk size and position in the video), for both hotspot and regular chunks, and (3) the bitrate at which prefetching would occur (if it were to occur), and the bitrate at which the next regular chunk would be downloaded (if prefetching now is deemed as a poor choice).

Therefore, with the objective of enabling scrupulous prefetch decisions, we identify the following states, which capture all the information required to generate an optimal prefetch decision:

1) *Player environment* (say, S1), which captures information related to the bandwidth and buffer conditions currently experienced by the client player;
2) *Regular chunk state* (say, S2), which captures details regarding the next regular video chunk;
3) *Hotspot chunk state* (say, S3), which captures details related to the next hotspot video chunk;
4) *Bitrate decisions* (say, S4), which captures the bitrates for the next hotspot chunk as well as the next regular chunk (both are required since prefetch decision may be yes or no).

The above discussion leads to an interesting observation: it is not sufficient to generate prefetch and bitrate decisions

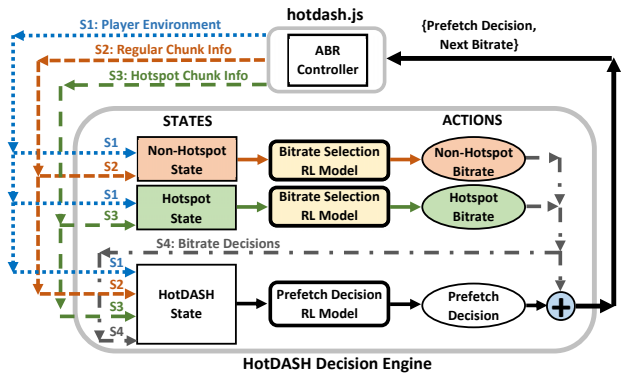


Fig. 3: HotDASH decision engine: The client player hotdash.js initiates a ABR request by sending across details of the player operating environment, and details of next hotspot and regular chunk to our cascaded model, which generates prefetch and bitrate decisions as the final output.

for a download opportunity simultaneously since the prefetch decision internally depends on the *hypothetical* bitrate decisions (for a hotspot if prefetch were to take place, and for a regular chunk, if not). This dependency leads us towards a cascaded design for the HotDASH decision engine, where the hypothetical bitrates are first generated using a *bitrate selection model*, and are fed as part of the input to the *prefetch decision model*.

B. Cascaded Design

The cascaded design of HotDASH is illustrated in Fig. 3. During a typical chunk download opportunity, the *ABRController* in hotdash.js generates the following state parameter sets: (S1) Player environment, (S2) Regular chunk information, and (S3) Hotspot chunk information. The states S1 (player environment) and S2 (regular chunk information) are combined to form the input to the *bitrate selection model*, which generates the optimal bitrate for the next regular chunk, if it were to be downloaded next. Similarly, the states S1 (player environment) and S3 (hotspot chunk information) are combined to form the input for another execution of the *bitrate selection model*, thus generating the optimal bitrate for the next hotspot chunk this time around. The bitrate decisions obtained from the two executions of the *bitrate selection model* are combined to form S4 (bitrate decisions).

Now that all the four required state-sets have been generated, a combination of these act as input to our *prefetch decision model* (the exact parameters used have been summarized in Table I). The output is a binary (yes/no) prefetch decision; if *no* is chosen, then the former bitrate is selected, and if *yes* is chosen, then the latter bitrate is selected. The prefetch decision, and the corresponding bitrate, jointly form the output of the HotDASH decision engine, and are sent back to the *ABRController* in hotdash.js, which initiates the corresponding download, thus completing the round of execution.

C. Reinforcement Learning for Optimal Decisions

The discussion above on a cascaded design for HotDASH led us to two different decision models, the *bitrate selection model* and the *prefetch decision model*. Owing to a large number of highly variable input parameters influencing prefetch and bitrate decisions, we realize that devising a straightforward optimization function, taking all these factors into consideration, is particularly difficult. Furthermore, the unpredictability in bandwidth when connected to a WiFi network in the wild, renders any fixed control rule sub-optimal. Consequently, we rely on the ability of a *reinforcement learning* algorithm, to experience a large variety of network conditions, and to learn from decisions made in the past.

TABLE I: HotDASH Decision Engine State Parameters

Category	Param	Significance
Player Environment (S1)	x_t	Throughput measurement for the last chunk
	τ_t	Download duration for the last chunk
	b_t	Play buffer occupancy
	l_t	Bitrate of last downloaded chunk
Regular Chunk State (S2)	$\vec{n_t}$	m available sizes for the next regular chunk
	c_t	No. of chunks remaining in video
Hotspot Chunk State (S3)	B_t	Total buffer occupancy (includes prefetched)
	$\vec{N_t}$	m available sizes for the next hotspot chunk
	C_t	No. of hotspot chunks remaining in video
	P_t	No. of chunks till playback of next hotspot chunk
	$\vec{V_t}$	Distance (in chunks) from every hotspot chunk, $\lvert\vec{V_t}\rvert = h$
	L_t	Bitrate of last downloaded hotspot chunk
Bitrate Decisions (S4)	A_t^r	Bitrate returned if next chunk is regular
	A_t^h	Bitrate returned if next chunk is hotspot

An RL algorithm operates in an environment E, takes as input a state S, takes an action A, and receives a reward R against its action. As the algorithm iterates over a large corpus of states and keeps receiving reward against all actions it performs, it learns the most optimal action for a particular state, such that the reward is maximized. In case of the *bitrate selection model*, the video player is the environment E, S1+S2 and S1+S3 form the input state S in two different executions, the QoE metric used to optimize the model is the reward R (described in §V-A4), and the corresponding bitrate decision is the action A. In case of the *HotDASH decision engine*, the environment E remains the same, S1 through S4 combine to form the input state S, the QoE metric designed specifically to train it is the reward R (described in §V-A5), and the binary prefetch decision is the action A.

D. RL Methodology

In this subsection, we describe the methodology adopted to train each of the reinforcement learning models *bitrate selection model* and *prefetch decision model*.

1) Training Mechanism: In the training phase, the HotDASH RL agent explores the video streaming environment E. In the most realistic case, the learning should take place using a real video playback implementation. However, such an implementation demands that for each training data point, the agent waits till a video chunk is downloaded from the content server over the Internet, before moving on to the next data point. This would result in unnecessary wastage of valuable training time.

In order to circumvent this problem, we implement a simulated playback environment for training our RL model. The simulator closely replicates a real video streaming scenario, by maintaining a buffer, which is similar to a real video client's playback buffer. In this context, we define two buffer parameters: (1) *Total buffer*, which maintains the total occupancy of the client buffer (including the out-of-order prefetched chunks), and (2) *Play buffer*, which maintains the duration for which playback is still possible without stalling (i.e., in-sequence buffer occupancy). These buffer parameters have been implemented in our version of the client player, i.e., hotdash.js; the exact implementation details have been explained in §IV-B2. Our simulation environment also maintains these two buffer parameters and behaves in exactly the same way as the system implementation in hotdash.js. When prefetch and bitrate decisions have been made for a chunk download, the simulator increments the total buffer with a new chunk. If the decision was to download a regular chunk, the play buffer is also incremented by one video chunk. However, while this download is taking place, the player continues to drain the play buffer by rendering downloaded chunks; this phenomenon is simulated by first computing the download time of the last chunk, and then draining the play buffer by the same amount. The download time is simply computed as the quotient of the size of the corresponding video chunk (which depends on the bitrate decided), and the bandwidth as obtained from the trace. The simulator also keeps track of rebuffering instances, i.e., when the playback buffer has been completely drained before the download of a new in-sequence chunk is complete, and records the exact time for which rebuffering has taken place. In case the total buffer is full, the simulator sleeps for a certain duration before downloading the next designated chunk, as is the case in a real DASH client implementation.

After the completion of each video chunk download, the simulator prepares the parameters which define the state S for our RL agent (exact parameters provided in Table I). Processing of state S by the RL agent is described in §III-D2.

2) Training Algorithm: HotDASH uses A3C [13], a state-of-the-art actor-critic method, which involves training two neural networks, i.e., an actor network, and a critic network. Fig. 4 illustrates the RL implementation for the *prefetch decision model* using A3C. Each parameter of state S_t is passed on to the *actor network* as well as the *critic network*, enabling the RL model to learn a decision *policy*. The *bitrate selection model* employs a similar implementation, with the state S consisting of a subset of the states (S1 and S2 for regular chunk bitrate decisions, and S1 and S3 for hotspot chunk bitrate decisions).

Inputs: After the download of each chunk t, the RL learning agent takes state inputs $S_t = [x_{t-1}, \tau_{t-1}, b_t, l_t, \vec{n_t}, c_t, B_t, \vec{N_t}, P_t, \vec{V_t}, L_t, A_t^r, A_t^h]$ to its neural networks (significance of notations has been provided in Table I).

Training: Given S_t, the RL agent needs to take an action A_t that corresponds to the prefetch decision for the next video chunk download opportunity. The agent selects actions based on a policy, defined as a probability distribution over actions

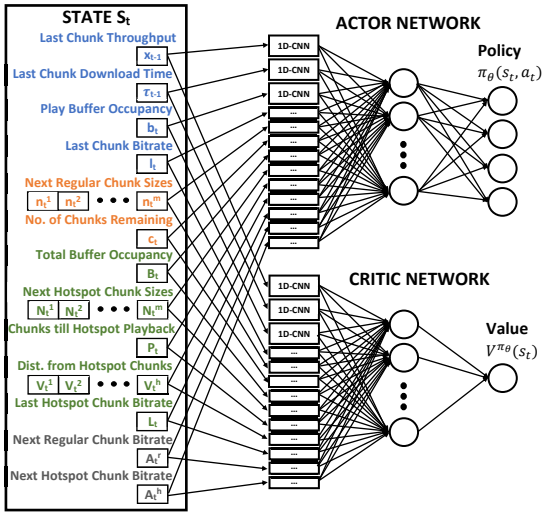


Fig. 4: Prefetch decision RL model: The prefetch decision model is implemented using A3C [13], an actor-critic network. The state S_t consists of 13 parameters, which is a combination of sub-states S1 through S4 (as listed in Table I).

$\pi : \pi(S_t, A_t) \rightarrow [0, 1]$. $\pi(S_t, A_t)$ is the probability that action A_t is taken in state S_t. The *prefetch decision model* and *bitrate selection model* use neural networks (NNs) to learn this policy. After applying each action, the simulated environment provides the learning agent with a reward R_t for that chunk. The primary goal of the RL agent is to maximize the expected cumulative (discounted) reward that it receives from the environment.

Parallel training: We spawn multiple learning agents (default is 16) in parallel to hasten training, as suggested in [13]. Each agent is configured to experience a different set of input parameters, based on the provided bandwidth trace. The central agent collects all [state, action, reward] tuples from these agents, and aggregates them to generate a single model.

IV. HotDASH Implementation

We have now looked at the design of the HotDASH decision engine. In this section, we delve in-depth into the implementation specifics of each component of HotDASH.

A. HotDASH Decision Engine Implementation

Following up from our discussions in §III, the HotDASH decision engine employs two RL-based models, i.e., the *prefetch decision model* and the *bitrate selection model*. We use the state-of-the-art ABR algorithm Pensieve [7], which also uses A3C [13] to learn optimal bitrates, as the *bitrate selection model* (we leverage a pretrained model provided by Pensieve, for this purpose). The *prefetch decision model*, on the other hand, is implemented in TensorFlow [15], by passing 8 past values of vector inputs ($\vec{n_t}$ and $\vec{N_t}$) to a 1D convolution layer (CNN) with 128 filters, each of size 4 with stride 1. The remaining scalar parameters are passed to another 1D-CNN having the same shape. The results obtained from these layers are then aggregated in a hidden layer, which uses 128 neurons

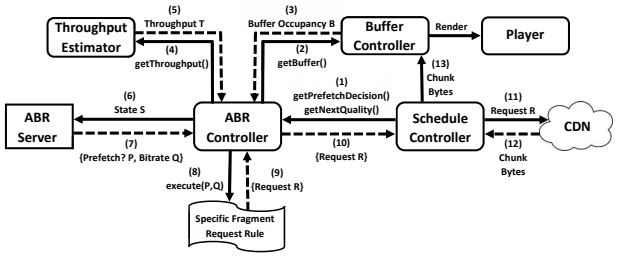


Fig. 5: Control Flow in hotdash.js: The schematic shows the flow of control in hotdash.js, from the end of one video chunk download, till the end of the next video chunk download.

and applies a softmax function. The critic network uses the same neural structure; however, its final output is a linear neuron (devoid of activation function). During the training phase, we set the value of discount factor $\gamma = 0.99$, thus allowing current actions to be influenced by 100 future steps. The learning rates for the actor network is set at 0.0001, while that for the critic network is set at 0.001. The entropy factor β is configured such that it gradually decays from 1 to 0.1 over $100,000$ iterations.

B. hotdash.js Implementation

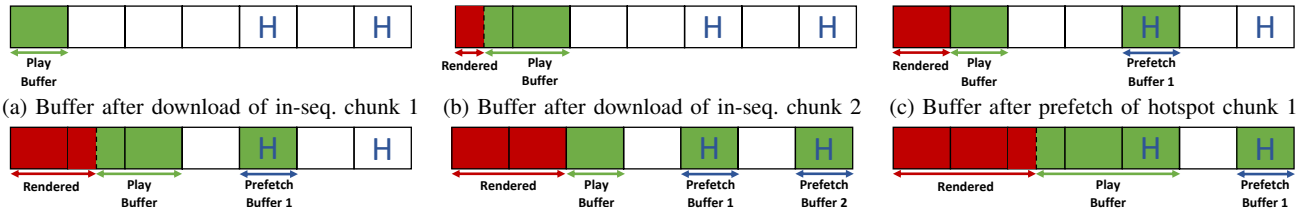
We recall that the client video player in our system, i.e. hotdash.js, is implemented on top of the reference DASH player implementation dash.js [12]. By default, dash.js progresses in a sequential stop-and-wait manner: once the download of a chunk starts, the scheduler repeats through sleep cycles, until the download is complete, and the chunk data is added to the buffer. Then it initiates the download of the next chunk in the pipeline, i.e., the next in-sequence chunk. However, prefetching a chunk requires two specific additional abilities: (1) requesting a video chunk by playback time, and (2) managing buffer such that out-of-order prefetched chunks can be accommodated, and appropriately rendered for playback. We describe in detail how we implement each of the above requirements in our version of the DASH player, i.e., hotdash.js.

1) Control Flow for Prefetch-enabled Player hotdash.js: The control flow in our implementation has been illustrated in detail in Fig. 5. We implemented the rule *SpecificFragmentRequestRule*, which enables downloading a video chunk by time (this time is the playback time of the chunk, when played offline/without rebuffering). We further equipped the *ABRController* and *ScheduleController* to implement prefetch decisions and bitrate decisions.

The flow of control in hotdash.js is as follows. Once the download of a chunk ends, the *ScheduleController* initiates the process for downloading the next chunk. It requests the *ABRController* for the prefetch decision (whether next download will be a hotspot prefetch or a regular chunk), and the corresponding bitrate (1). The *ABRController* requests the *BufferController* for buffer occupancy status (2), and the *ThroughputEstimator* for estimated throughput (4); the corresponding responses are sent as buffer occupancy B (3), and throughput estimate T (5), respectively. Hereafter, the *ABRController* combines B and T along with other relevant parameters (discussed in §III), and sends this combination across to the *ABRServer* (HotDASH decision engine) as state S (6). The *ABRServer* returns the prefetch decision P, and the corresponding bitrate Q, based upon S (7). The *ABRController* now sends P and Q to a rule (8), called the *SpecificFragmentRequestRule*, which has the ability to build a video chunk request based upon specified playback time; the chunk request R is returned to the *ABRController* (9), and then back to the *ScheduleController* (10). The *ScheduleController* executes this request (11), to fetch the actual video chunk bytes from the corresponding CDN (12). The chunk bytes are then handed over to the *BufferController* (13), which adds these chunks bytes to the player buffer. The *ScheduleController* is now prepared to download the next chunk. While this entire flow of control takes place, the buffer gets continually depleted by the video player, by rendering chunks for playback.

2) Buffer Management: dash.js is equipped with an internal buffer which is capable of storing out-of-sync chunks, in addition to regular in-sequence chunks. We exploit this capability to implement the buffer in hotdash.js. We account for two logical buffer parameters: (1) *Play buffer*, and (2) *Total buffer* (introduced in §III-D1). The *play buffer* refers to the part of the internal buffer, which stores in-sequence video bytes, which have been downloaded, but are yet to be rendered (played on the video player). Note that in order to be played seamlessly, the buffer necessarily requires video bytes in sequence. However, when a prefetch decision is executed, hotdash.js ends up downloading a video chunk which is completely out-of-sequence. Such prefetched hotspot chunks are also stored in the internal buffer, and these buffer segments combined with the play buffer, form the *total buffer*. Total buffer is therefore the actual measure of buffer occupancy, while play buffer is a measure of how long the video player can go on playing without stalling (rebuffering). The total buffer gets boosted whenever a video chunk is downloaded; the play buffer, however, gets boosted only when a regular chunk is downloaded. Video bytes get rendered continuously, thus depleting both the play buffer and total buffer (except in the case of rebuffering, when there is no play buffer left to deplete).

Fig. 6 illustrates this concept using an example scenario, for playback of a video segregated into 7 chunks. The chunk numbers 5 and 7 are the hotspot chunks, and have been labeled with the letter 'H'. We represent downloaded content, yet to be played with the *green* colour, and already rendered content with the *red* colour. For simplicity of explanation, we assume that the buffer is depleted by one-half of a chunk by the player, in the time required to download a chunk. First, chunk numbers 1 and 2 are downloaded in-sequence ((a) and (b)). Then the hotspot chunk number 5 is downloaded (c); the play buffer still ends at chunk number 2. Then chunk number 3 is downloaded in-sequence (d), followed by a prefetch of hotspot chunk number 7 (e). Now, as chunk number 4 is downloaded in-sequence (f), *buffer reconciliation* occurs, which means that



(a) Buffer after download of in-seq. chunk 1 (b) Buffer after download of in-seq. chunk 2 (c) Buffer after prefetch of hotspot chunk 1

(d) Buffer after download of in-seq. chunk 3 (e) Buffer after prefetch of hotspot chunk 2 (f) Buffer after download of in-seq. chunk 4

Fig. 6: Buffer management in hotdash.js: The buffer in hotdash.js is equipped to store both regular chunks, and prefetched hotspots chunks.

a prefetched chunk (number 5) gets added to the play buffer, and strengthens it by two chunks at-a-time.

V. Evaluation

We evaluate our proposed system HotDASH both by illustrating the bitrate improvement obtained due to the ability to prefetch, and by analyzing the impact of prefetch decisions on overall QoE. In this section, first we discuss the methodology used to evaluate HotDASH, and then we present the corresponding results.

A. Methodology

The methodology used for HotDASH is as follows.

1) Network Trace Datasets: In order to evaluate the performance of HotDASH on realistic network environments, we leverage two publicly available datasets, and create a corpus of 2000 bandwidth traces of 320 secs, similar to Pensieve [7].

Source Traces: The datasets include a broadband dataset provided by the US Federal Communications Commission [16], and another from Telenor's HSDPA connections in Norway [17]. The US dataset consists of more than 1M traces, with each trace containing average throughput values for every 5 secs, logged over an interval of 35 mins. The Norway dataset contains 30 mins traces of scenarios when users were in transit.

Dataset Generation: For our experiments, we construct a trace corpus using 1000 randomly selected intervals (spanning 320 secs each) from each dataset. The throughput bounds of 0.2 Mbps (lower) and 6 Mbps (higher) are enforced, so that trivial cases may be avoided (above 6 Mbps can sustain maximum bitrate throughout, whereas below 0.2 Mbps cannot sustain any available bitrate). All traces are formatted according to the requirements of the Mahimahi [18] network emulation tool, which we use for our live experiments.

Train-Test Segregation: In order to train HotDASH, we randomly sample 80% of the traces; the remaining 20% is used for testing HotDASH and the baseline ABR algorithms.

2) Hotspot Selection and Generalization: We train HotDASH with a fixed combination of hotspot chunks,and test it (and baseline algorithms) over 100 carefully selected hotspot combinations, to determine whether it generalizes well.

Number of Hotspot Chunks: In all our experiments, we designate 10% of the video chunks as hotspot chunks (say, C chunks). This is based on the assumption that very high user interest can sustain for around 10% of playback time (e.g., 12 mins in a 2 hour movie).

Hotspot Combinations for Training and Testing: During training, we randomly select a combination of C hotspot chunks, and keep this combination fixed for all training epochs. During testing, however, our aim is to ascertain whether HotDASH generalizes well across a wide selection of hotspot combinations. In order to achieve this, we employ the WSP space-filling algorithm [19]. WSP enables selection of a set of points in an n-dimensional space, such that the points are at least d (input parameter) distance apart from each other, and are distributed such that the entire space is well-explored. We input the number of dimensions as C (number of hotspot chunks), and allow WSP to generate a set of 100 hotspot combinations. No repetition of values are allowed in a single combination.

3) Baseline ABR Algorithms: We present performance comparisons between HotDASH and six baseline algorithms, which are representative of the state-of-art ABR techniques.

1) **Buffer Based:** Huang et al. [14] proposed a buffer-based algorithm, which attempts to select bitrates such that the buffer occupancy is always maintained above 5 seconds. The highest available bitrate is selected if there is a 15 seconds buffer cushion.
2) **Rate Based:** Bandwidth experienced during last chunk download is naïvely estimated to be the current throughput. The highest bitrate supported by this throughput is selected.
3) **Festive:** Festive [20] is a more sophisticated rate-based algorithm, where throughput prediction is done using the harmonic mean of the throughput values observed during the last 5 chunk downloads. The highest bitrate below this throughput estimate is selected.
4) **FastMPC:** FastMPC [6] performs optimization for a QoE metric (discussed in §V-A4), over the next 5 video chunks, based on past throughput estimation and buffer occupancy observations.
5) **RobustMPC:** RobustMPC [6] is a variation of FastMPC, which further improves it by considering and accounting for errors observed between predicted and observed throughput values during last 5 chunk download instances.
6) **Pensieve:** Pensieve [7] selects the next optimal bitrate by using reinforcement learning to optimize over a QoE metric (discussed in §V-A4).

4) Quality of Experience (QoE) Metrics: Various studies over the last decade have identified different metrics for accurately measuring the quality of experience (QoE) associated with video streaming [4], [21]–[24]. The general QoE metric

used by MPC [6] is defined as:

$$QoE = \sum_{i=1}^{n} q(R_i) - \mu . \sum_{i=1}^{i} T_i - \sum_{i=1}^{n-1} |q(R_{i+1} - q(R_i)| \quad (1)$$

QoE is defined for a video with n chunks; R_i is the bitrate of $chunk_i$, and $q(R_i)$ maps this bitrate to a utility value, which represents the quality perceived by the user. The rebuffering time resulting from downloading $chunk_i$ at bitrate R_i is given by T_i; μ is a weight which determines how heavily rebuffering is penalized. The final term accounts for playback smoothness, and penalizes QoE when consecutive bitrates change abruptly. In a nutshell, QoE increases with high bitrates, but diminishes with rebuffering, and lack of smoothness.

We consider three different variations of QoE for our performance evaluation, each of which defines a different $q(R_i)$, and a different value of μ.

1) QoE_{lin}: In the linear case,

$$q(R_i) = q_{lin}(R_i) = R_i;\ \mu = \mu_{lin} = 4.3$$

MPC [6] used this metric for optimization.

2) QoE_{log}: In the logarithmic case,

$$q(R_i) = q_{log}(R_i) = log(R/R_{min});\ \mu = \mu_{log} = 2.66$$

This metric is based on the rationale that some users cannot perceive the improvement in quality at higher bitrates as prominently, as they do when the bitrates are lower. BOLA [25] used this metric in their implementation.

3) QoE_{hd}: In the high definition (HD) case, $q(R_i)$ assigns static scores for different bitrate values:

$$q(R_i) = q_{hd}(R_i) = \left\{ \begin{array}{ll} 1, & \text{if } R_i = 0.30 \\ 2, & \text{if } R_i = 0.75 \\ 3, & \text{if } R_i = 1.20 \\ 12, & \text{if } R_i = 1.85 \\ 15, & \text{if } R_i = 2.85 \\ 20, & \text{if } R_i = 4.30 \end{array} \right\};\ \mu = \mu_{hd} = 8$$

This metric favors HD video rendering by assigning increasingly higher values to higher bitrates. Pensieve [7] used this metric alongside QoE_{lin} and QoE_{log}.

5) QoE Metric used for Training HotDASH: We train our RL model in the HotDASH decision engine, using a variant of QoE_{lin} as the RL reward metric. Let a target video consisting of n chunks be represented by the ordered set $V = \{v_1, v_2, ..., v_n\}$. We define an ordered set $H \subset V$, which consists of all m hotspot chunks present in the video: $H = \{h_1, h_2, ..., h_m\}$, such that $m \leq n$. We recall that the objective of our model is to determine prefetch opportunities, such that the hotspot bitrates are maximized, but at the same time, other objectives (maximum regular chunk bitrates, minimum rebuffering, and maximum smoothness) are not compromised. We incorporate this notion in the QoE metric by defining $QoE_{hotspot}$, a variant of QoE_{lin}, such that:

$$QoE_{hotspot} = \sum_{h \in H} q_{hd}(R_h) + \sum_{v \in V-H} q_{lin}(R_v) - \mu . \sum_{i=1}^{i} T_i - \sum_{i=1}^{n-1} |q_{lin}(R_{i+1}) - q_{lin}(R_i)| \quad (2)$$

The RL model is encouraged to download hotspot chunks at higher bitrates, by using the HD reward function $q_{hd}(.)$ to determine bitrate utility of hotspot chunks. In contrast, the bitrate utility for regular chunks is determined using the linear function $q_{lin}(.)$. The rest of the terms remain the same as in QoE_{lin}. This modification in the bitrate reward for hotspot chunks reflects the user's preference for hotspot chunks over regular chunks, while keeping other QoE indicators intact.

6) Experimental Setup: We implemented all the baseline algorithms in hotdash.js, which is based on dash.js (version 2.4) [26] [2]. The baseline algorithms *Buffer Based*, *Rate Based*, and *Festive* were implemented entirely in hotdash.js. Since HotDASH, Pensieve [7], and both variants of MPC [6] rely on an external ABR server for bitrate selection decisions (and also prefetch decisions in case of HotDASH), we implemented the servers (with corresponding algorithm implementations) using Python `HTTPBaseServer`. Hooks were provided inside *hotdash.js* for interaction (sending request with playback state, and receiving response with decisions) with the ABR server, using `XMLHttpRequests`. In our setup, the ABR server and the video player were run on different client machines in the same local network. The player hotdash.js was configured with a (total) buffer capacity of 60 seconds. We used the "Envivio-dash3" video for all our experiments [27]; this video has been encoded using the H.264/MPEG-4 codec, and is available at 6 qualities – 240p (bitrate: 300 Kbps), 360p (bitrate: 750 Kbps), 480p (bitrate: 1200 Kbps), 720p (bitrate: 1850 Kbps), 1080p (bitrate: 2850 Kbps), and 1440p (bitrate: 4300 Kbps). The video consists of 48 chunks of approx. 4 seconds each, and has a total playback time of 193 seconds. The video chunk server was implemented in the same workstation (Ubuntu 16.04) where the video player was run. The chunks were served using Apache2 (version 2.4.18), while the client player used a Google Chrome browser (version 60). The network emulation tool Mahimahi [18] was used to emulate network conditions represented by the throughput traces generated in §V-A1.

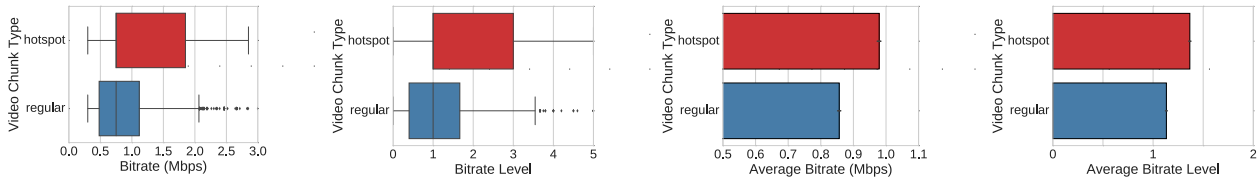
B. Results

In this subsection, we evaluate our approach HotDASH, from three different perspectives. First, we explore the impact of prefetch decisions taken by HotDASH. Second, we observe how HotDASH fares against baseline algorithms. Third, we dive deeper into HotDASH performance, and illustrate how different components contribute to the QoE metrics considered.

1) Impact of Prefetch Decisions: We note that the primary aim of HotDASH is to prefetch hotspot chunks, so that they can be played at a higher bitrate, than when left to the vagaries of unpredictable bandwidth.

Bitrate Improvement for Hotspot Chunks: We compare the bitrate values (in Mbps) and the bitrate levels, between prefetched hotspot chunks and regular chunks. The bitrate levels 0 through 5 correspond to the bitrates 0.3, 0.75, 1.2, 1.85, 2.85, and 4.3 Mbps respectively. The comparison is performed by identifying prefetch opportunity windows: a particular hotspot chunk can only be prefetched in the time interval between the time instance of last prefetch, and the

[2]The code is available at https://github.com/SatadalSengupta/hotdash



(a) Bitrate values for prefetched hotspot chunks & regular chunks (b) Bitrate levels for prefetched hotspot chunks & regular chunks (c) Avg. bitrate values for prefetched & regular chunks (d) Avg. bitrate levels for prefetched & regular chunks

Fig. 7: Bitrate improvement: HotDASH prefetched hotspot chunks at 14.31% better bitrate on average, than regular chunks.

time instance when all regular chunks before it have already been downloaded. We consider the bitrate of the prefetched hotspot chunk as the hotspot bitrate (and the corresponding bitrate level as the hotspot bitrate level), while we compute the mean bitrate (and level) of all regular chunks in the prefetch opportunity window, and regard it as the regular chunk bitrate (and the corresponding level as the regular chunk bitrate level).

Fig. 7 illustrates the improvement in bitrates for hotspot chunks due to prefetch decisions taken by HotDASH. Fig. 7a shows that the bitrates at which hotspots were downloaded, were generally higher than the bitrates for regular chunks. Similarly, Fig. 7b demonstrates a similar observation for the bitrate levels. The improvement in average bitrate between prefetched hotspot chunks and regular chunks, can be observed in Fig. 7c, while the corresponding improvement in bitrate level is presented in Fig. 7d. The mean bitrate improvement due to prefetching is computed to be 14.31%.

Prefetch Decisions: The frequency of prefetch decisions per video playback session is shown in Fig. 8. We observe that HotDASH utilizes prefetch opportunities fairly aggressively, by deciding to prefetch all 5 hotspots in the video around 40% of the time. This is followed by prefetching 4 out of 5 ($\sim$36%), 3 out of 5 ($\sim$18%), and then the remaining (less than 10%). While the prefetching is aggressive, the observed bitrate improvements demonstrate that HotDASH uses its prefetch opportunities wisely.

2) QoE Improvement over Baseline Algorithms: In order to compare HotDASH with the baseline ABR algorithms (described in §V-A3), we perform live experiments on our corpus of network traces using hotdash.js. The QoE metric considered for these experiments is $QoE_{hotspot}$, as defined in §V-A5. Since the baseline algorithms do not have the ability to prefetch, if the throughput goes down during download of hotspot chunks, then their bitrate may fall drastically. Since hotspot chunks obtain QoE according to $q_{hd}(.)$, a high difference in hotspot bitrate would result in a much larger QoE difference, than in the case of regular chunk bitrate. In order to facilitate a fair comparison, we force baseline algorithms to always download hotspot chunks at the highest bitrate available for the chunk. This elevates the bitrate component of the QoE metric to the largest extent possible.

The performance of HotDASH in comparison to baseline ABR algorithms, is depicted in Fig. 9 and 10. Fig. 9 illustrates that HotDASH nearly always obtains more than 80% QoE (w.r.t. minimum observed QoE). The raw values of QoE (in the figure legend) show that HotDASH is the only ABR algorithm which ends up with a positive average QoE. In Fig. 10, the average normalized (w.r.t. worst performing baseline Festive) QoE shows that HotDASH outperforms Buffer Based, which is the best performing baseline algorithm, by 16.2%. HotDASH outperforms Pensieve, RobustMPC, Rate Based, and FastMPC by 30%, 32.6%, 44.3%, and 67.4%, respectively.

3) Overall QoE Comparison Between HotDASH and Pensieve: We compare the average value of individual QoE components, as well as total QoE, for HotDASH (where prefetch is enabled), and for Pensieve [7] (where prefetch is not allowed). We consider all three QoE metrics discussed in §V-A4, i.e., QoE_{lin}, QoE_{log}, and QoE_{hd}. The comparisons are presented in Fig. 11. We observe that the average bitrate QoE is always higher in case of HotDASH, irrespective of the QoE metric chosen. This may be attributed to the fact that HotDASH focuses on providing higher bitrates to chunks by making better utilization of available throughput. However, the focus on higher bitrates results in higher rebuffering penalty across all three QoE metrics. The smoothness penalty, however, is again better in case of HotDASH across all the three scenarios. Overall, HotDASH obtains better QoE in two cases (QoE_{lin} and QoE_{hd}) out of three, with the exception of QoE_{log}. Since QoE_{log} disregards higher bitrates by awarding increasingly lower QoE as bitrates go up, the advantage of using a prefetch-enabled system is lost on it.

VI. Related Work

Research on adaptive bitrate video streaming over the years can be broadly classified into 3 categories: (1) determining Quality-of-Experience (QoE) for users, (2) video preparation methods, and (3) bitrate adaptation strategies for better QoE.

Determining Quality-of-Experience: Researchers realized early on that objective technical parameters, such as the Peak-Signal-to-Noise-Ratio (PSNR), do not necessarily correlate with the viewing experience of an user. A hybrid mechanism, which combined the goodness of objective scores (e.g., PSNR), and subjective scores (e.g., MOS), was proposed in 2009 [21]. Song *et al.* attempted to understand how user preferences and prior playing conditions impacted QoE; they concluded that user gender and frequency of video watching impacts QoE significantly, as does the bitrate range [2]. Gender was concluded to be a major factor in QoE prediction in a more recent study [4]. Mok *et al.* studied the relationship among network quality, application QoS and user QoE, and

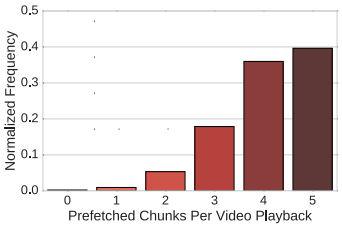


Fig. 8: Frequency (normalized) of prefetch decisions taken during each video playback session. HotDASH utilizes prefetch opportunities well, and decides to prefetch all 5 hotspots around 40% of the time.

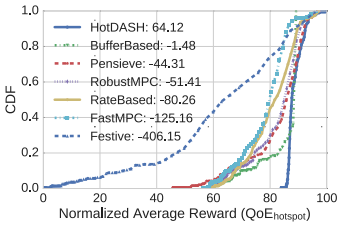


Fig. 9: CDF of $QoE_{hotspot}$ obtained for each algorithm is shown. HotDASH nearly always obtains positive QoE (above 80% normalized QoE), which is in sharp contrast with the baselines algorithms.

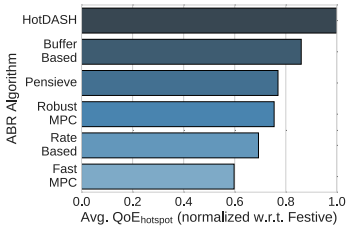
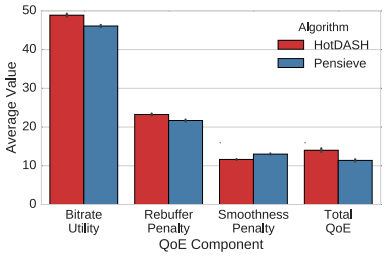
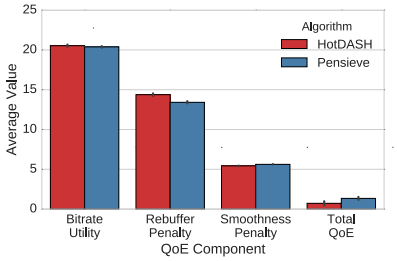


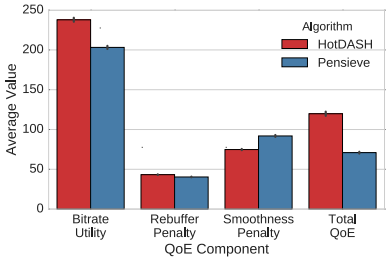
Fig. 10: Normalized $QoE_{hotspot}$ (w.r.t. FESTIVE) obtained for each algorithm. HotDASH outperforms the best performing baseline BufferBased by 16.2%, followed by Pensieve, RobustMPC, RateBased, and FastMPC.



(a) QoE components for QoE_{lin}



(b) QoE components for QoE_{log}



(c) QoE components for QoE_{hd}

Fig. 11: Comparison of individual QoE components and total QoE between HotDASH (where prefetch is enabled), and Pensieve (where prefetch is not possible). HotDASH always obtains better bitrate QoE, and results in better total QoE in 2 out of 3 cases (QoE_{lin} & QoE_{hd}).

how it manifests into user activities on the player (e.g., pauses, reducing viewing area, etc.) [23], [24]. Studies also revealed that rebuffering is the most detrimental QoE factor for most users, although the extent of QoE decline depends on the type of content (e.g., live content is more affected) [3], [28]. Balachandran *et al.* concluded that the type of video (live vs. streaming), connectivity (cable/DSL vs. wireless), and device type (PC vs. mobile devices vs. TV), are the 3 most important factors which affect user engagement [29]. Song *et al.* proposed an adaptive mobile video streaming strategy based on QoE factors such as video content and encoding, bandwidth limitations, device features, and viewing context; they concluded that focusing on these factors maximized overall user engagement and mimimized resource cost [30]. Recently, a comparative study of 3 DASH players revealed that the preferred adaptation algorithm varies with players and their configuration [31]. In this work, we focused on the QoE variability due to difference in content preferences of users, and proposed a system which accounts for it.

QoE-aware Video Preparation: In the space of video preparation with QoE considerations, Zhang *et al.* proposed detection of points-of-interest for video generation, based on sensor metadata and multiple camera views [32]. Choi *et al.* defined a simple and fast metric for identifying blockiness in frames of videos encoded at different bitrates, and optimized video generation for least perceptual blockiness [33]. Wu *et al.* identified that mobile users seldom perceive bitrate differences as prominently as PC users, and tried to minimize bitrate overhead [34]. In [8], the authors propose a neural model for determining temporal regions-of-interest in a video, with the objective of video captioning. This work can serve as a potential precursor to our system HotDASH, by identifying the hotspot segments in a video.

Bitrate Adaptation Strategies: We already discussed the bitrate adaptation strategies Buffer Based [14], Festive [20], MPC [6], and Pensieve [7] in §V-A3. Additionally, Sun *et al.* proposed CS2P, a system which optimizes bitrate adaptation based on large-scale data-driven throughput prediction [35]. An online learning bitrate adaptation strategy was proposed in [36]. QUETRA proposes a simple rate adaptation algorithm by modeling a DASH client as a M/D/1/K queue [37]. Koo *et al.* proposed REQUEST, a system which selects bitrate adaptively over both WiFi and LTE based on constraints on battery consumption and data usage [38]. Wang *et al.* formulates the adaptive video streaming problem as a multi-step predictive control problem, and optimizes for it [39]. In this work, we do not work on bitrate adaptation, rather we aim on redistribution of optimal bitrate selections, by taking into account the content preferences of users.

VII. Conclusion

In this paper, we proposed HotDASH, a system which takes into consideration content preferences of users during adaptive video streaming over HTTP. The system enables prefetching of hotspot chunks, which are video chunks with higher (than other chunks) user interest associated with them. The prefetch and bitrate decisions are taken in an opportune manner, such that higher bitrates are selected for hotspot chunks, but other

QoE considerations, such as bitrate of regular chunks, rebuffering, and smoothness, are not heavily compromised. This is made possible by the application of reinforcement learning to dynamically learn optimal prefetch decisions and bitrate decisions through experience, over a large corpus of network traces. HotDASH achieves an improvement of 16.2% over the best performing baseline in terms of average QoE, and is able to improve hotspot bitrates by 14.31%, as compared to regular chunk bitrates.

REFERENCES

[1] V. N. I. Cisco, "Global mobile data traffic forecast update, 2015–2020 white paper," *Document ID*, vol. 958959758, 2016.

[2] W. Song, D. Tjondronegoro, and M. Docherty, "Saving bitrate vs. pleasing users: where is the break-even point in mobile video quality?" in *Proc. of the 2011 ACM Multimedia Conference*. ACM, 2011, pp. 403–412.

[3] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. Joseph, A. Ganjam, J. Zhan, and H. Zhang, "Understanding the impact of video quality on user engagement," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 362–373.

[4] Y. Zhu, A. Hanjalic, and J. A. Redi, "Qoe prediction for enriched assessment of individual video viewing experience," in *Proc. of the 2016 ACM Multimedia Conference*. ACM, 2016, pp. 801–810.

[5] Z. Duanmu, K. Ma, and Z. Wang, "Quality-of-experience of adaptive video streaming: Exploring the space of adaptations," in *Proc. of the 2017 ACM Multimedia Conference*, 2017.

[6] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli, "A control-theoretic approach for dynamic adaptive video streaming over http," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 325–338, 2015.

[7] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 197–210.

[8] Z. Yang, Y. Han, and Z. Wang, "Catching the temporal regions-of-interest for video captioning," in *Proc. of the 2017 ACM Multimedia Conference*, 2017.

[9] H. Jin, Y. Song, and K. Yatani, "Elasticplay: Interactive video summarization with dynamic time budgets," in *Proc. of the 2017 ACM Multimedia Conference*, 2017.

[10] A. Volokitin, M. Gygli, A. Vasudevan, and L. Van Gool, "Query-adaptive video summarization via quality-aware relevance estimation," in *Proc. of the 2017 ACM Multimedia Conference*, 2017.

[11] A. Patney, M. Salvi, J. Kim, A. Kaplanyan, C. Wyman, N. Benty, D. Luebke, and A. Lefohn, "Towards foveated rendering for gaze-tracked virtual reality," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 6, p. 179, 2016.

[12] D. I. Forum, "dash.js," https://github.com/Dash-Industry-Forum/dash.js, 2017.

[13] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, 2016, pp. 1928–1937.

[14] 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," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 187–198, 2015.

[15] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[16] US Federal Communications Commission (FCC). [Online]. Available: http://data.fcc.gov/download/measuring-broadband-america/2016/data-raw-2015-jun.tar.gz

[17] University of Oslo. [Online]. Available: http://home.ifi.uio.no/paalh/dataset/hsdpa-tcp-logs/

[18] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan, "Mahimahi: Accurate record-and-replay for http." in *USENIX Annual Technical Conference*, 2015, pp. 417–429.

[19] J. Santiago, M. Claeys-Bruno, and M. Sergent, "Construction of space-filling designs using wsp algorithm for high dimensional spaces," *Chemometrics and Intelligent Laboratory Systems*, vol. 113, pp. 26–31, 2012.

[20] J. Jiang, V. Sekar, and H. Zhang, "Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012, pp. 97–108.

[21] K. Piamrat, C. Viho, J.-M. Bonnin, and A. Ksentini, "Quality of experience measurements for video streaming over wireless networks," in *Information Technology: New Generations, 2009. ITNG'09. Sixth International Conference on*. IEEE, 2009, pp. 1184–1189.

[22] I. Ketykó, K. De Moor, T. De Pessemier, A. J. Verdejo, K. Vanhecke, W. Joseph, L. Martens, and L. De Marez, "Qoe measurement of mobile youtube video streaming," in *Proceedings of the 3rd workshop on Mobile video delivery*. ACM, 2010, pp. 27–32.

[23] R. K. Mok, E. W. Chan, X. Luo, and R. K. Chang, "Inferring the qoe of http video streaming from user-viewing activities," in *Proceedings of the first ACM SIGCOMM workshop on Measurements up the stack*. ACM, 2011, pp. 31–36.

[24] R. K. Mok, E. W. Chan, and R. K. Chang, "Measuring the quality of experience of http video streaming," in *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*. IEEE, 2011, pp. 485–492.

[25] K. Spiteri, R. Urgaonkar, and R. K. Sitaraman, "Bola: near-optimal bitrate adaptation for online videos," in *Proc. of the 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[26] DASH Industry Forum. [Online]. Available: http://reference.dashif.org/dash.js/v2.4.0/samples/dash-if-reference-player/index.html

[27] Envivio-dash3. [Online]. Available: http://dash.edgesuite.net/envivio/Envivio-dash2

[28] S. S. Krishnan and R. K. Sitaraman, "Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs," *IEEE/ACM Transactions on Networking*, vol. 21, no. 6, pp. 2001–2014, 2013.

[29] 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 *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 339–350.

[30] W. Song, D. Tjondronegoro, and I. Himawan, "Acceptability-based qoe management for user-centric mobile video delivery: A field study evaluation," in *Proc. of the 2014 ACM Multimedia Conference*. ACM, 2014, pp. 267–276.

[31] D. Stohr, A. Frömmgen, and A. Rizk, "Where are the sweet spots? a systematic approach to reproducible dash player comparisons," in *Proc. of the 2017 ACM Multimedia Conference*, 2017.

[32] Y. Zhang, R. Zimmermann, L. Zhang, and D. A. Shamma, "Points of interest detection from multiple sensor-rich videos in geo-space," in *Proc. of the 2014 ACM Multimedia Conference*. ACM, 2014, pp. 861–864.

[33] M.-K. Choi, H.-G. Lee, M. Song, and S.-C. Lee, "Adaptive bitrate selection for video encoding with reduced block artifacts," in *Proc. of the 2016 ACM Multimedia Conference*. ACM, 2016, pp. 282–286.

[34] C. Wu, W. Zhu, Q. Li, and Y. Zhang, "Rethinking http adaptive streaming with the mobile user perception," in *Proc. of the 2017 ACM Multimedia Conference*, 2017.

[35] Y. Sun, X. Yin, J. Jiang, V. Sekar, F. Lin, N. Wang, T. Liu, and B. Sinopoli, "Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction," in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016, pp. 272–285.

[36] F. Chiariotti, S. D'Aronco, L. Toni, and P. Frossard, "Online learning adaptation strategy for dash clients," in *Proceedings of the 7th International Conference on Multimedia Systems*. ACM, 2016, p. 8.

[37] P. K. Yadav, A. Shafiei, and W. T. Ooi, "Quetra: A queuing theory approach to dash rate adaptation," in *Proc. of the 2017 ACM Multimedia Conference*, 2017.

[38] J. Koo, J. Yi, J. Kim, M. A. Hoque, and S. Choi, "Request: Seamless dynamic adaptive streaming over http for multi-homed smartphone under resource constraints," in *Proc. of the 2017 ACM Multimedia Conference*, 2017.

[39] B. Wang and F. Ren, "Towards forward-looking online bitrate adaptation for dash," in *Proc. of the 2017 ACM Multimedia Conference*, 2017.