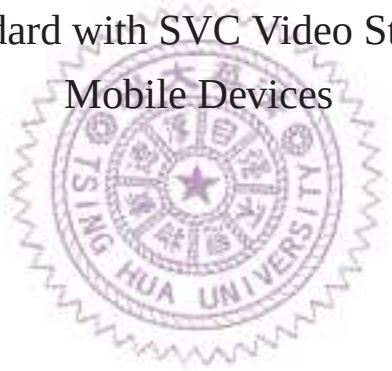
國立清華大學電機資訊學院資訊工程研究所
碩士論文
Department of Computer Science
College of Electrical Engineering and Computer Science
National Tsing Hua University
Master Thesis

實現 HTTP 動態調整串流特性整合可伸縮影像編碼器串流系統於 Android 智慧型裝置
MPEG-DASH Standard with SVC Video Streaming on Android Mobile Devices



陳建彰
Chien-Chang Chen

指導教授：徐正炘 博士
Advisor: Cheng-Hsin Hsu, Ph.D.

中華民國 104 年 06 月
June, 2015

國立清華大學
資訊工程研究所

碩士論文

實現 HTTP 動態調整串流特性整合可伸縮影像編碼器串流系統於 Android 智慧型裝置

陳建彰 撰

104
06

中文摘要

本篇我們著重在設計、實作並測量 H.264/SVC 解碼器，將此解碼器移植到多核心行動裝置上，並支援 HTTP 影像串流功能。 我們實作的解碼器使用多執行緒，充份發揮多核心的特性，除此之外，影像串流伺服器及用戶端皆支援可動態調整的 HTTP 影像串流。 我們為了測量解碼器的效能，在 Android 的平版電腦及智慧型手機上進行實驗，播放 H.264/SVC 影片來獲得數據。 依照實驗結果顯示， 在多核心的 Android 行動裝置上進行即時解碼是可行的。 舉例來說，當播放 960x544 的影片時， 解碼器的畫面更新率（FPS，畫面數／秒）可以達到20.72 FPS；當播放 480x272 的影片時，畫面更新率可 42.03 FPS。 除了對解碼器進行測量外，我們另外做了影像串流的實驗，分別在 WiFi 跟 3G 網路下使用 HTTP 影像串流。 使用 HTTP 影像串流時， 我們的解碼器最高可達到 42 FPS，以及短暫的初始延遲（約 2.5 秒）。 我們進一步擴充我們的測試平台，支援 MPEG-DASH 標準以及導入 SDL 函式庫。 MPEG-DASH 套用 SVC 可帶來下列益處：(1) 可以降低串流伺服器的儲存需求；(2) 跟伺服器要求的檔案可以重複使用達到減少伺服器的負擔；以及 (3) 可以更快速地回應調整影片畫質的請求。 我們對 MPEG-DASH 以及 SDL 進行實驗。 實驗結果顯示資料產出量每秒最高可達到 15 Mbits，播放 360x180 的影片，解碼器可達到至少 50 FPS。 最後，我們也將已建置的測試平台開放給研究社群使用。

Abstract

We design, implement, and evaluate an H.264/SVC decoder and an HTTP video streaming client on multi-core mobile devices. The decoder employs multiple decoder threads to leverage multi-core CPUs, and the streaming server/client support adaptive HTTP video streaming. To evaluate the decoder performance, we conduct experiments using real H.264/SVC videos on a tablet and a smart phone running Android 4.0. Our experimental results demonstrate that real-time H.264/SVC decoding is feasible on multi-core mobile devices. For example, for 960x544 and 480x272 videos, our decoder achieves up to 20.72 and 42.03 Frame-Per-Second (FPS), respectively. We also conduct extensive HTTP video streaming experiments over live WiFi and 3G cellular networks. Our system achieves high frame rate (up to $\sim$42 FPS), and short initial delay (as small as $\sim$2.5 secs). We extend our testbed to support MPEG DASH (Dynamic Adaptive Streaming over HTTP) standard and SDL (Simple DirectMedia Layer) library. The benefits of MPEG-DASH with SVC are that (i) the storage space requirement is reduced, (ii) segments can be reused to reduce server overhead, and (iii) switching events are performed faster. Last, we conduct experiments using MPEG-DASH standard and SDL library. The results show that the throughput of MPEG-DASH achieves up to $\sim$15 Mbits/s, and our decoder achieves at least 50 FPS for 360x180 videos. We have made our testbed publicly available to the research communities.

Contents

List of Figures

List of Tables


TSING HUA UNIVERSITY

Chapter 1

Introduction

Mobile video streaming is getting increasingly popular. Video streaming will account for more than 70 percent of mobile data traffic in 2019 according to the Cisco's report [10]. The mobile data traffic will achieve 24.3 ExaBytes per month in 2019, which is 10 times the traffic in 2014. It means a majority of users are using their mobile device to watch videos. It is important to provide better user experience for mobile streaming services. The traditional *nonscalable* video coders encode video into a stream, which can only decode the video stream at one single quality. Nonscalable coders are less suitable for mobile video streaming since they can not timely respond to heterogeneous mobile devices and adapt to dynamic wireless network conditions. In contrast, *scalable* video coders encode each video into one based layer and a number of enhancement layers. The based layer provides the basic video quality, and the enhancement layers provide the incremental quality enhancements. H.264/SVC [19] is the novel scalable coding standard, but has not been widely used for mobile video streaming because there is no efficient H.264/SVC decoder on resource-constrained mobile devices. For illustrations, we consider five different types of H.264/SVC videos, which are encoded in multiple resolutions at 24 FPS. Fig. 1.2 to 1.6 are the screenshots of the videos which are contents courtesy of CBC/Radio-Canada. All of these videos are originally the full High-Definition (HD) videos and we compute the Peak Signal-to-Noise Ratio (PSNR) in Chapter 6. Table 1.2 lists the descriptions of these five videos. We plot the *Spatial Information* (SI) and *Temporal Information* (TI) [14] of each video in Fig. 1.7. The SI means the complexity of frames. The frame with more components has higher complexity. We calculate the complexity of each frame and set the maximum one as the SI. We also calculate the TI for all videos. TI is the complexity of two continuous frames. The higher TI means the more difference between two continuous frames. In Fig. 1.7, if the point is close to the top right, it means that video has higher complexity. For example, `sport` has a lot of fast moves so that it has the highest TI. Most frames of talk are the human face and there are a few

Table 1.1: SVC Decoders are not real-time on a laptop.

Decoder	doc	jeux	soap	sport	talk
JSVM	17.75 (FPS)	20.36	19.44	17.44	19.11
OpenSVC	18.79	27.03	20.79	18.71	24.26

Table 1.2: Videos Descriptions

Video	**Description**
doc	a documentary video talking about a woman who lost her house
jeux	a live show video about the guessing games
soap	an action style soap video
sport	a sports news video including volleyball, basketball, swimming, etc.
talk	a talk show video

fast moves in the video, which leads to the lowest SI and TI. We decode the videos using two single-threaded decoders, JSVM and OpenSVC, on a laptop with an Intel i5 2.3 GHz CPU running OS X. Table 1.1 presents the average FPS, which shows that the existing H.264/SVC decoders may not run in real-time on laptops, let alone on mobile devices.

Recently, many multi-core mobile devices have been released, which may allow true parallelism for real-time applications such as H.264/SVC decoders. In this work, we develop an SVC decoder for multi-core mobile devices. Fig. 1.1 gives a screenshot of our streaming client, and the decoder running on an Android 4.0 mobile phone which is developed in our previous work [15]. To provide better event handler and larger display space, we also design the new User Interface (UI) which is detailed in Sec. 5.3. We conduct real experiments using HD videos with diverse characteristics on various mobile devices. The experimental results are very encouraging. For example, our decoder achieves up to 20.72 FPS for 960x544 videos, and 42.03 FPS for 480x272 videos on commodity multi-core mobile devices. When streaming scalable videos over HTTP to a quad-core Android phone, we also achieve high frame rate, as high as $\sim$42 FPS, and short initial delay, as small as $\sim$2.5 secs.

To the best of our knowledge, that software-based video decoders are inherently more power-hungry than hardware-based solutions, as we have observed in our evaluations. Unfortunately, there exists no massively-produced SVC decoder chip at the time of writing. We believe this is because the benefits of SVC have not been evaluated in the wild. Our end-to-end scalable video streaming testbed can be used by the mobile multimedia community for setting up complete SVC-based testbeds. We firmly believe that this will stimulate more research studies on SVC and the production of SVC decoder chips. There is a technology has similar idea to SVC, which is dividing the video into a lots of small

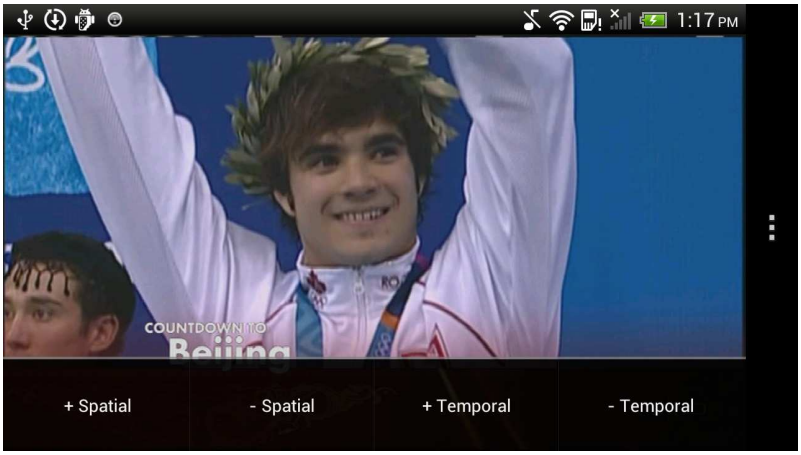


Figure 1.1: Our client and decoder running on an Android phone.

Figure 1.2: Screenshot of `doc` video.

Figure 1.3: Screenshot of `jeux` video.

Figure 1.4: Screenshot of `soap` video.

Figure 1.5: Screenshot of `sport` video.

Figure 1.6: Screenshot of `talk` video.

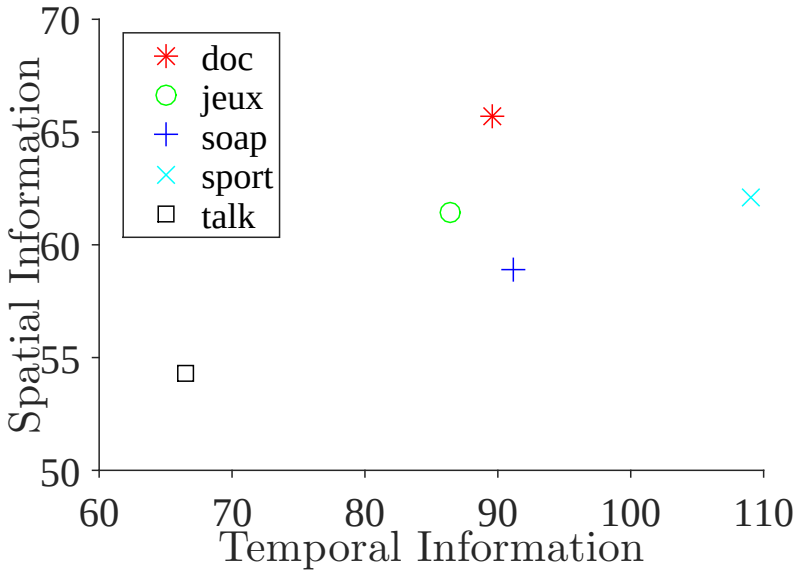


Figure 1.7: The SI and TI for each video.

chunks, called adaptation streaming.

In the recent years, the HTTP protocol streaming has become the most popular approach for delivering multimedia contents. In particular, adaptation HTTP streaming allows clients to switch to different streams on the fly. In 2012, MPEG published the adaptation HTTP streaming standard, called MPEG-DASH standard. We extend our system to support MPEG-DASH standard (see Sec. 5.2) and conduct the experiments in Chapter 6.

Chapter 2

Related Work

Although there are commercial H.264/SVC decoders [4, 5], their implementations are proprietary and thus are less suitable for research projects. There are two publicly available H.264/SVC decoders: (i) Joint Scalable Video Model (JSVM), which is the reference software of the H.264/SVC standard, and (ii) OpenSVC decoder [11], which is an open-source project, but has not been ported to modern mobile OS's. For this project, we build a multi-core decoder on Android using the library offered by the OpenSVC project.

Mueller and Timmerer propose a session mobility testbed streaming based on Dynamic Adaptive Streaming over HTTP (DASH) [17], which employs nonscalable videos. In a more recent work, Mueller et al. conduct DASH streaming experiments [16] using scalable videos. Their work focus on adaption processing and compares the performance among the MPEG-DASH and proprietary solutions. The concerned metrics are average bit-rate, number of quality switches, buffer level, and unsmooth seconds, between AVC and SVC with MPEG-DASH. Their traces are recorded by cellular network by car drives on three different highways. The results show that MPEG-DASH with H.264/SVC outperformances than MPEG-DASH with AVC. Sieber et al. [20] proposed new adaption algorithm compares with other three existing adaption algorithms on H.264/SVC MPEG-DASH streaming. Their algorithm achieves high playback quality, high bandwidth utilization, low switching frequency, low memory consumption compared with other three algorithms. Sanchez et al. [13] work focus on proxy cache and compares MPEG-DASH with H.264/SVC and MPEG-DASH with H.264/AVC video streaming on VoD service. The results show that the SVC streaming can reduced the server overhead, because of SVC layered structure can fully utilize the proxy cache. SVC streaming can provide more number of clients than AVC streaming, because of proxy hit-rate is higher.

All of above works are focus on Networks performance, such as how to increase the Quality of Experiment (QoE) to provide highest quality, and how to reduce stalling

frequency. Our work is complementary to [16] in the sense that we develop a real-time MPEG-DASH with H.264/SVC decoder on Android OS and build an end-to-end HTTP streaming testbed. Our work not only evaluation the performance of Networks but also evaluate the H.264/SVC decoder. Since we make our code publicly available [9] for research community. We want to let more people to understand the benefits of MPEG-DASH with SVC and apply this system in the real world.

Chapter 3

Background

In this section, we will introduce the idea of H.264/SVC and MPEG-DASH standards.

3.1 H.264/SVC Standard

The traditional video coders are nonscalable which is not suitable for heterogeneous mobile devices, such as H.264/AVC. When we encode the video, we need to configure encoding parameters, like frame-rate, resolution, and coder. For nonsacalable video coding, we encode one stream per configuration, therefore we need a lot of spaces to store multiple versions of videos. The client needs to choose the proper version of video to display. If the client chooses the wrong version, it will stall while playback or will spend too much resources for decoding. The scalable video coding is a better solution to solve this problem.

H.264/SVC standard [19] extends H.264/AVC standard [22] to support scalable video coding. The idea of SVC is a layered structure which divides the video into multiple layers. H.264/SVC provides three scalable features: temporal, spatial, and quality: (i) T temporal layers, where each layer leads to a different frame rate, (ii) S spatial layers, where each layer leads to a different resolution, and (iii) Q quality layers, where each layer leads to a different fidelity level controlled mostly by quantization parameters. When decoding an H.264/SVC stream, user selects a tuple $<t, s, q>$ and decodes the corresponding sub-stream for the target representation of frame rate $0 \leq t < T$, resolution $0 \leq s < S$, and fidelity level $0 \leq q < Q$. These sub-streams allow multimedia systems to conserve resources by not, e.g., storing, transmitting, buffering, uncompressing, or rendering some layers.

For each scalable feature, the layers are split into two types, based layer and enhancement layer. There are only one based layer and one or more enhancement layers for SVC video stream. Based layer contains necessary data for decoding. Client displays the worst

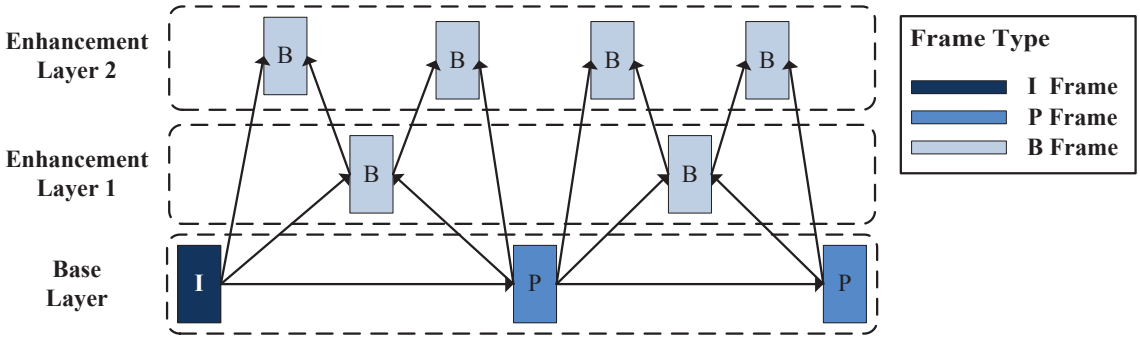


Figure 3.1: Example of temporal scalability.

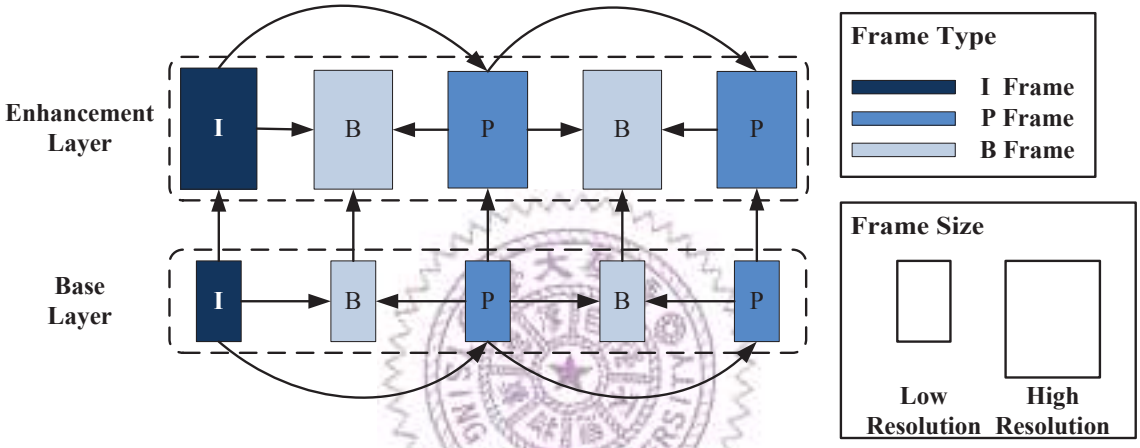


Figure 3.2: Example of spatial scalability.

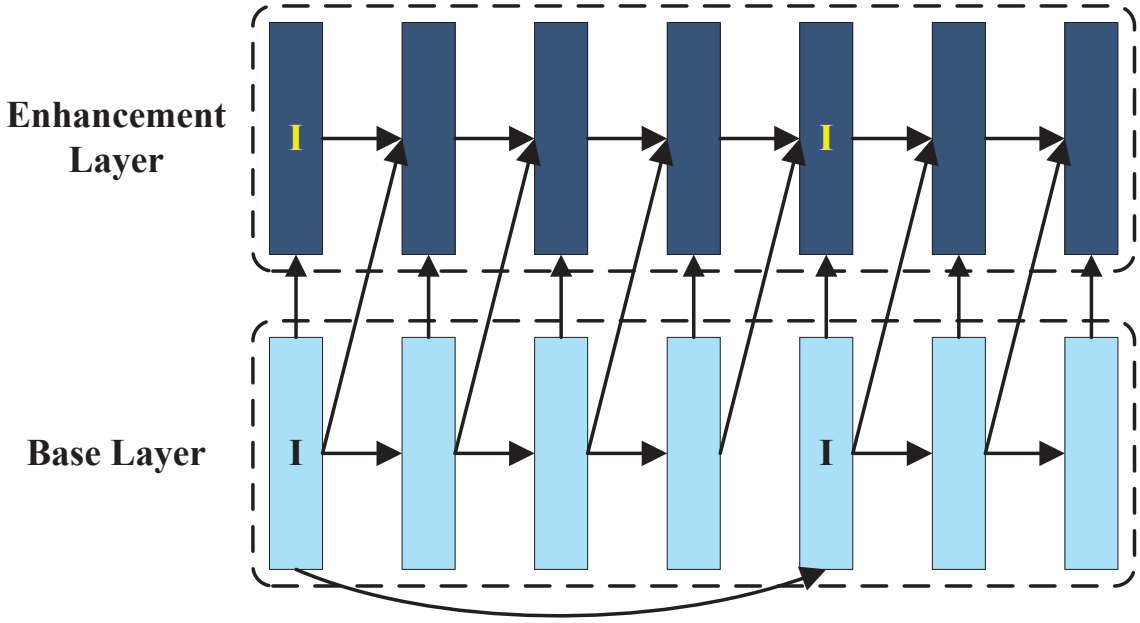


Figure 3.3: Example of quality scalability.

quality video, if decoder only decodes based layer. The worst quality can be defined by lowest frame-rate, smallest frame size, or worst fidelity level, according to encoding configuration. Enhancement layers provide extra information for enhancing fidelity level, frame-rate, or resolution. The SVC is a layer dependent structure. For instance, the higher enhancement layer depends on the lower enhancement layer and the lowest enhancement layer depends on the based layer. In other words, if the decoder is missing the based layer, decoder cannot decode any frames. If decoder is missing enhancement layers, the video stream can't be enhanced any more. The following are description of examples for scalable features.

- **Temporal Scalability.** Fig. 3.1 is an example of temporal scalability. Temporal layer number is $0 \leq t < 3$. Temporal layer 0 means based layer, temporal layer 1 means enhancement layer 1, and so on. The GOP size for this example is 9. This figure shows that the based layer includes 3 frames. Enhancement layer 1 provides 4 more B frames for enhancing. There are total 7 frames when decoding based layer and enhancement layer 1. We assume the frame-rate is 27 when decoding all temporal layers ($t = 2$). The frame-rate is decreased to 9 when decoding only based layer. Therefore, more temporal layers leads to higher frame-rate.

- **Spatial Scalability.** The example of spatial scalability is shown in Fig. 3.2. There are two layers, based layer and enhancement layer ($0 \leq s < 2$). If number of spatial layer is increasing, the resolution of decoded frames are higher. Each enhancement layer frame references correspond to based layer frame data. The enhancement layer reconstructs the higher resolution frame according to the based layer frame data and its frame data. So, the enhancement layer data only store the difference between higher and lower resolution data.

- **Quality Scalability.** Fig. 3.3 is an example of quality scalability. The reference structure of quality scalability is like the combination of temporal and spatial scalability. The lower quality layer has a worse fidelity level. The enhancement layer not only reference the lower layer data but also the previous frame data in the same layer.

User can select the appropriate number of layers to decode while using scalable video coding. We know that the android mobile devices are heterogeneous, such as different screen size and computation power. If screen size of mobile device is 1280x720, the user does not want to decode 4K video. Likewise, if computation power of android mobile device is not powerful to decode the full-quality SVC stream, decoder can ignore some enhancement layers to meet the device computation power. That is why scalable video coding is more suitable for nonscalable video coders for heterogeneous devices.

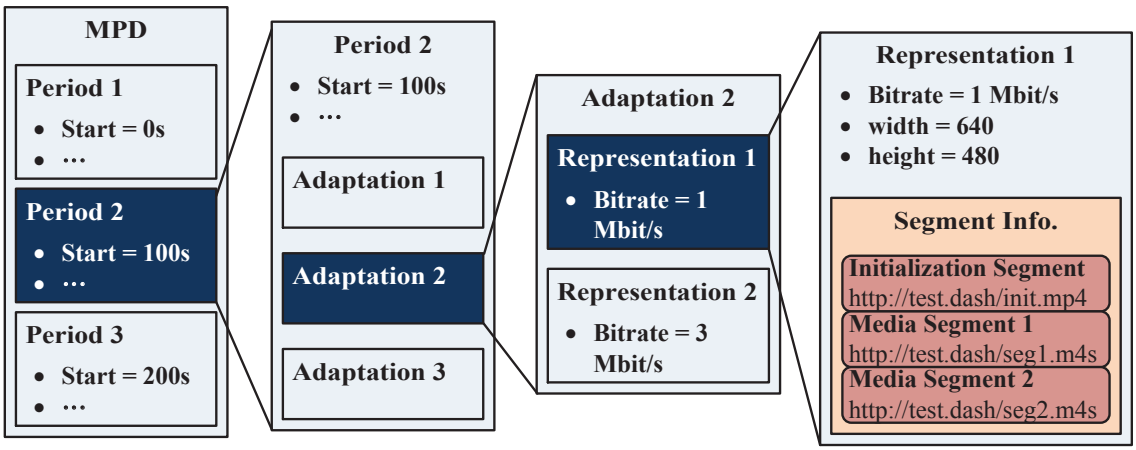


Figure 3.4: MPD hierarchical structure.

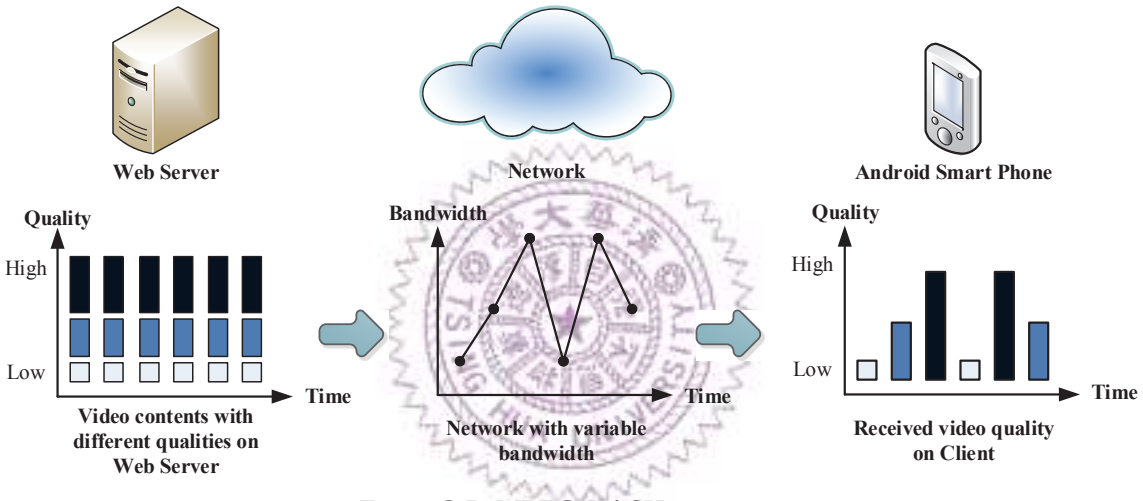


Figure 3.5: MPEG-DASH overview.

3.2 MPEG-DASH Standard

For recent years, HTTP protocol becomes a popular approach to delivery multimedia contents. Traditional streaming protocol, like *Real-Time Streaming Protocol* (RTSP), is a stateful protocol. The drawback of stateful protocol is that server needs to maintain the connection state after setting up the connection between client and server. Server and client need to send and respond the messages to keep tracking the connection state. The better approach to deliver multimedia content is using HTTP-based approach. HTTP protocol is stateless protocol so that server does not need to maintain the connection. The server responds to the media streams when it receives the HTTP request from client. It's easier to scale up number of streaming clients with slight cost. Other benefit of HTTP protocol is that HTTP can reuse the infrastructure, like CDNs, proxies, and caches. To compare with traditional streaming approach, HTTP protocol is easy to go through the firewall and without NAT traversal issues.

For the past few years, progressive download is used for transferring the multimedia contents over HTTP. By using progressive download, client requests byte range of multimedia contents from server. If client stops to watch video while multimedia contents are downloading, the downloaded media data becomes useless which leads to waste the network resources. Progressive download has another drawback which does not support switching to other streams during the downloading. The adaptive HTTP streaming can solve the drawbacks of progressive download.

The adaptive HTTP streaming consists of two components, manifest file and segments. There are some existing solutions of adaptive HTTP streaming, such as Apple HTTP Live Streaming, Adobe HTTP Dynamic Streaming, and Microsoft Smooth Streaming. But the mentioned solutions are the proprietary solutions. The problem of proprietary solutions is that they are incompatible to each other. Hence, MPEG published the adaptive HTTP streaming standard with Third Generation Partnership Project (3GPP), companies, and experts, and called it MPEG-DASH [2]. The idea of MPEG-DASH is chopping the video into multiple segments with the same segment length. The segment length is defined by display time. The MPEG-DASH segments are divided into two types, initial segment and media segment. The manifest file of MPEG-DASH is called *Media Presentation Description* (MPD).

Fig. 3.4 is a simplified hierarchical structure of MPD which is written by *eXtensible Markup Language* (XML). In this example, the outermost layer is called Period and the innermost layer is called Segment List. Each MPD includes a number of Period layers. The Period defines the start time and period of time for partial video. Period can be considered as a set of continuous segments. For each Period, it contains multiple Adaptation Sets. The Adaptation Set is a video or an audio track. If we have two different language voice and three different videos, there are total 5 Adaptation Sets inside the Period. Each Adaptation Set includes multiple Representations. The Representation is one version of stream. In other words, if we encode the video into four different resolutions, there are 4 Representations inside the Adaptation Set. Representation defines the average bit-rate, resolution for video stream, and sample rate for audio stream. Client side switches to different Representations according to the measured information, like current available bandwidth, state of buffer which is used for storing the downloaded segments, or user preference. Client can only request the segments inside the selected Representation. Fig. 3.5 is the overview of MPEG-DASH with H.264/AVC. The Web Sever stores the different versions of H.264/AVC media contents. The medium of figure is measured bandwidth. The client side based on measured bandwidth to select the appropriate segment and requests it.

The MPD structure in Fig. 3.5 is nonscalabe video streaming. The difference between

nonscalable and scalable streams are: (i) Representation is equal to specific layer for SVC. For example, if video encoded into 5 layers, there are 5 Representations. (ii) Client requests the segments from multiple Representations for SVC streaming rather than from one of Representations. If user want to display the lowest quality video stream, client requests the based layer segments from the Representation. For the MPEG-DASH with H.264/AVC, client focus on how to choose the most suitable stream to request. For the nonscalable screams, how to obtain the sufficient accuracy for measured information to choose the stream is the most important point. If measuring result is not precise enough, the segment can't be downloaded completely before deadline. In contrast, the point of MPEG-DASH with SVC video streaming is to download as more enhancement layers as possible for client. If bandwidth is decreasing rapidly while requesting the segment of enhancement layer, client can cancel downloading segment and display video without stalling.

As we mentioned, we need to encode the multiple versions of videos for nonscalable video coders. For scalable video coding, we only need to encode one version and provide multiple features of videos. Hence, using SVC can reduce the space requirement on the streaming server side. For the SVC streaming, the requested segment can be used for all of clients, like based layer. If some clients want to enhance the quality level, clients request more enhancement layers. The segments are requested once and used for many times so that reduce the overhead of the streaming server. For the traditional MPEG-DASH streaming, client only can switch to the different streams at the boundary of segments. If we use MPEG-DASH with SVC streaming, the client can switch the layer number not only at the boundary of segments but also switch the layer number during playing the segments. Using MPEG-DASH with SVC provides more flexible switch points than traditional MPEG-DAHS streaming. Therefore, MPEG-DASH with SVC streaming provides the better user experience than nonscalabe video streaming.

Chapter 4

System Architecture

Fig. 4.1 is our system architecture which contains 9 components on both server side and client side. The components *MPD Generator* and *DASH Content* are located at server side, Web HTTP server. The other components, such as *DASH Client*, *SVC Decoder*, *Renderer*, *Switch Event Handler*, and *Data Recorder*, are on the android client. DASH Client consists of *MPD Parser*, *Segment Requester*, and *Extractor*. The following is simplified step for each component.

1. Download the HD videos from [6], encode into H.264/SVC format, and generate the segments and MPD.
2. Place the segments and MPD to somewhere that can be requested by client.
3. Download MPD and parse it to obtain video information.
4. Download corresponding video segments according to MPD information and selected quality level.
5. Extract the non-decodable segments (i.e. different layer segments) and reconstruct into a decodable segment for SVC Decoder.
6. Decode the SVC segment to obtain frames for rendering.
7. Display the decoded frames on the screen.
8. When switching event occurs, Segment Requester, Extractor, and SVC Decoder are notified.
9. Data Recorder is recording the throughput, delay, and FPS while displaying.

Step 4 to step 7 is a loop, DASH Client keeps downloading and extracting, SVC Decoder keeps decoding, and Renderer keeps rendering. The following are the detailed description of each component.

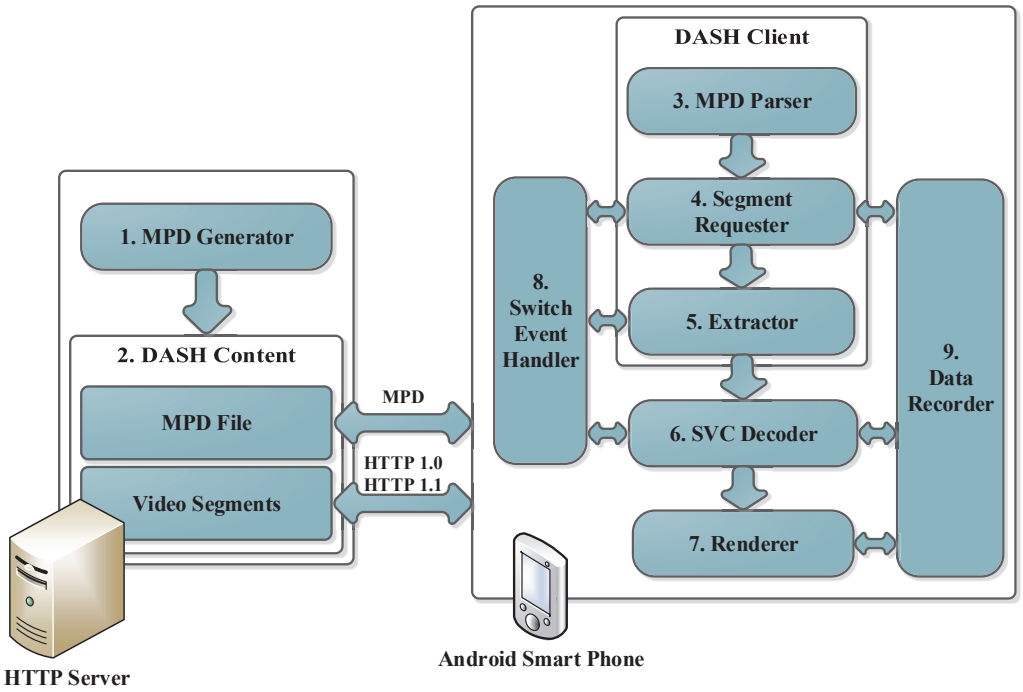


Figure 4.1: System Architecture.

First, we describe the components on the server side. All components on the server side are preprocess works. At the beginning, we download the five HD videos, `jeux`, `doc`, `soap`, `sport`, and `talk`, which are available from [6]. The video contents are courtesy of CBC/Radio-Canada. We encode these five HD videos into raw format (i.e. yuv420p) videos with multiple resolutions by FFmpeg. The number of resolutions are the same as the number of spatial layers of the SVC video. For example, if SVC video has 3 spatial layers, there are 3 different resolutions for raw videos. After getting raw videos, we encode them into SVC format by JSVM. For each SVC video, there are one main configuration file and multiple layer configuration files. The main configuration file defines the location of output SVC video, number of layers, encoding frames, and locations of each layer configuration files, etc. Each layer configuration file defines encoding parameter of its layer, such as location of raw video, resolution, frame-rate, and QP value, etc. If this layer configuration file is an enhancement layer, there is one additional parameter to indicate the dependent layer IDs.

Once the encoding process completes, MPD Generator uses GPAC [8] library to generate the MPD and to chop SVC video into one initial segment and multiple media segments. We divide the MPD Generator into three steps: (i) At first, GPAC tool parse the SVC format video to acquire the number of layers, the resolution of each layer and frames data. (ii) Second, GPAC imports the parsed SVC bit-stream into ISO Base Media File Format (ISOBMFF). (iii) Last, GPAC chops the ISOBMFF file, which results from GPAC importing, into one initial segment and multiple media segments. The chopped

segments are also the ISOBMFF files. For the last step of MPD Generator, we only set one parameter which is segment length, and others remain default. We place chopped segments and MPD files into DASH Content component which can be requested by clients. In other words, these segments and MPD files can be addressed by accessible URL. So far, the server side is ready for streaming.

On the client side, user selects one video at first for streaming. The DASH Client starts to parse the corresponding MPD file and stores the information. The information in MPD we're concerned about includes the URL addresses of segment, number of layers, layer dependency ID, and resolution for each layer. The most important information is the URL address, because other information is also stored inside the segments. But we cannot obtain the information from the segment until the segment is extracted by Extractor. Based on the situation, we decide which component to use its information. The Segment Requester operates on the basis of the selected quality level (i.e. download how many layers) and addresses of segments. Our quality selection is based on the user preference, and this also can be decided by the algorithms. The Segment Requester keeps requesting the new segment until the free space of buffer is not enough to store the segment. As we mentioned, the segment is ISOBMFF which means we cannot decode this segment directly. To solve this problem, we implement the Extractor (more detail in Sec. 5.2.3). Once the Extractor receives one segment, it starts to parse the ISOBMFF format. The ISOBMFF format can be considered as a lot of different types of boxes. Each box contains two piece of information: box header which includes box type and box size, and payload. Different box types have different box structures and different payloads. Extractor parses the boxes and acquires the media data from mdat box. If Extractor receives the dependency segments, it begins to reconstruct the media data into compatible format for SVC Decoder. Once SVC Decoder gets the media data from Extractor, SVC Decoder starts to decode (decoder implementation in Sec. 5.1). After SVC Decoder completely decodes one GOP, the decoded frames are sent to Renderer and the Renderer shows the decoded frames on the screen. The SVC Decoder and Renderer are the most important components in this work. The steps from DASH Client to Renderer will keep going until player is stopped.

The last two components, Switch Event Handler and Data Recorder, are the auxiliary components. The Switch Event Handler can be triggered by algorithms or user itself (implementation in Sec. 5.3). Once the Switch Event Handler is triggered, Switch Event Handler sends the switching signal to DASH Client and SVC Decoder. For DASH Client, Segment Requester needs to change the number of layers for requesting, and Extractor cancels the segments if layer number exceeds the selected layer number. For SVC Decoder, decoder needs to change the decoding parameters and update the display size according to resolution of displaying frame. During the display, the Data Recorder keeps

measuring the information, like throughput, FPS, delay, etc. The recorded information can be used by switching algorithms or performance analysis. The above are the descriptions of our system, the next section talks about the implementation detail of SVC Decoder, DASH Client, and Switch Event Handler.

Chapter 5

Implementations

5.1 Multi-Core Decoder on Android Devices

5.1.1 Limitations of Single-Threaded Decoder

We first implement and evaluate a single-threaded SVC decoder on Android using OpenSVC library [11]. Since OpenSVC decoder is implemented in C/C++, it cannot be compiled as an Android application directly. Hence, we adopt Android Native Development Kit (NDK) to embed the OpenSVC library as native functions, and we develop a Java application, which interacts with these native functions via Java Native Interface (JNI). We evaluate our single-threaded SVC decoder using two Android 4.0 devices, with 1.2 and 1.4 GHZ CPUs, respectively. We decode five 375-secs videos coded at 960x544 and 24 FPS , and we found that the achieved frame rates are always less than 50% of the coded frame rates. Hence, we develop a multi-core H.264/SVC decoder in the following.

5.1.2 Software Architecture of Our Multi-Core Decoder

Fig. 5.1 shows the software architecture of our multi-core decoder which is proposed from our previous work [15]. It consists of two major components: (i) a Java front-end and (ii) a native decoder. The native decoder is implemented using Android NDK, and is interfaced with the Java front-end using JNI. The Java front-end also interacts with Android's Java Framework offered by Android SDK. The native decoder consists of a Coded Frame Buffer (CFB), H decoder threads, and a Decoded Frame Buffer (DFB). The CFB holds the H.264/SVC video packets read from video files or networks. The decoder threads concurrently reconstruct the raw frames from the video packets, and store the resulting frames in the DFB.

Our SVC decoder works as follows. First, the Java front-end passes the initial arguments to the native decoder, and asks it to *decode*. Once the decoded frames are stored

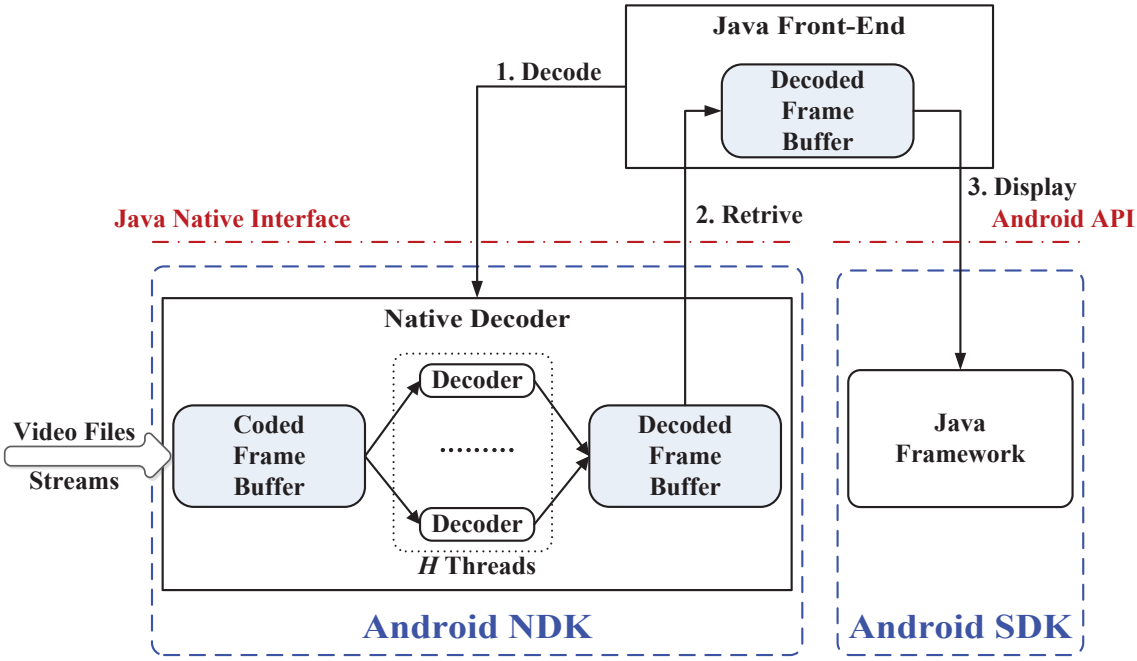


Figure 5.1: Decoder architecture.

in the DFB of the native decoder, the Java front-end *retrieves* the frames. Last, the Java front-end uses Android API to *display* frames in its DFB.

Our decoder threads employ the OpenSVC library [11] to decode the videos. Although this seems to be straightforward at first glance, using OpenSVC library in multi-threaded applications turns out to be fairly challenging because it is not designed to be multi-thread safe. We had to pay extra attentions to avoid race conditions while invoking functions in OpenSVC library. We make our decoder available to the research community [9].

5.1.3 Parallelism Strategy

Our decoder employs multiple decoder threads, where each thread works on a *group* of video packets at each moment. The groups are determined by a *parallelism strategy*. Several parallelism strategies have been proposed, e.g., at macro-block (MB), frame, and GOP levels [12, 18]. There are data dependencies among groups of video packets. That is, before decoding the next group of video packets, the thread must check its dependency. Each strategy has its advantages and disadvantages. For example, MB-level parallelism realizes finer-grained groups at the expense of complex group inter-dependency. GOP-level parallelism minimizes the interdependency among groups but demands more memory due to larger DFBs. GOP-level parallelism leads to the highest possible FPS, and hence we implement GOP-level parallelism in our decoder. In particular, we employ *pthread* to implement GOP-level parallelism. Other strategies are also possible by

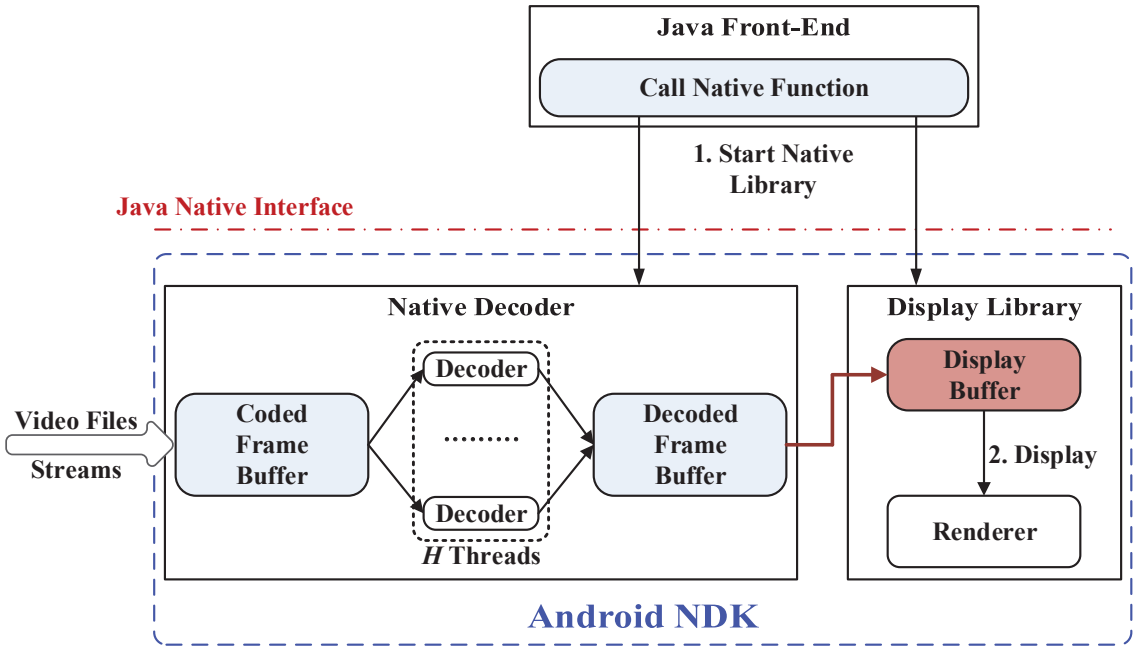


Figure 5.2: Decoder architecture with SDL.

redefining the groups, which is one of our future tasks.

5.1.4 Porting SDL Library as Renderer to Decoder

The SDL library is a popular cross-platform library to handle video and audio, and to access user inputs, like screen touch event and keyboard event. We found that the step 2 in our decoder architecture (Fig. 5.1) copies the data from native side to Java front-end by JNI. Every time for displaying, decoder movies large data from DFB from native side to Java. Because of decoder uses Android API to render the frames. If decoder uses the renderer in the native library, the step 2 can be removed from our decoder architecture. Hence, we port the SDL library to render the frame in native side rather than Android API. Fig. 5.2 is our new decoder architecture with SDL library. After porting the SDL library into our decoder, decoder can render the frames by SDL library directly. We know that there is an event queue inside the SDL library. Each event, including SDL predefined events and costumed events, will be put into event queue. If decoder, which without SDL, handles input events, it possible can not get the user events when rendering thread is displaying on Java front-end. The reason is that Android main thread is charging of all UI events, like rendering and screen touch event. When main thread is displaying, the screen touch event can not be captured at the same time. Porting SDL library which enables to use the frame buffer in the native side directly and to handle UI events easier.

5.2 MPEG-DASH Client with SVC Decoder on Android Devices

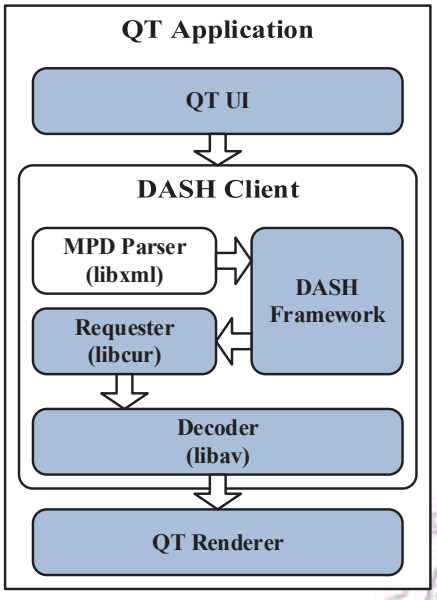


Figure 5.3: libdash architecture.

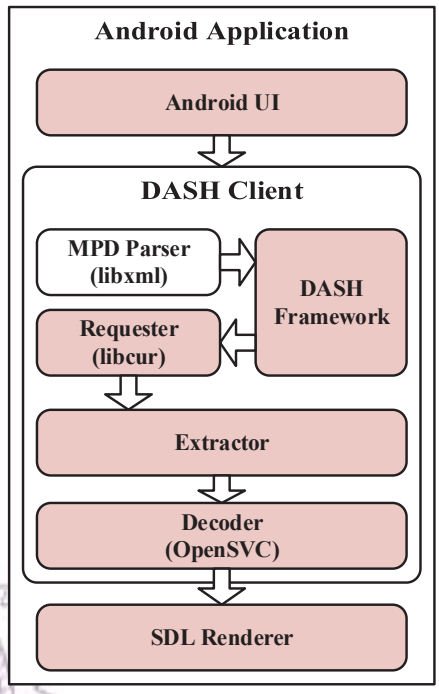


Figure 5.4: Our DASH client architecture.

5.2.1 Architecture of DASH Client

Our *DASH Client* is based on libdash library and we will describe our DASH Client in this section. The libdash is open source library available from [7] and Fig. 5.3 is libdash architecture. This library is implemented by c++ and has QT based simple player. The QT based simple player contains QT UI, MPD Parser, DASH Framework, Requester, Decoder, and QT Renderer.

The first component in libdash is *MPD Parser*. The MPD Parser is based on libxml library and MPD file is written by XML. After parsing the MPD, the data will be stored into *DASH Framework* component, such as number of layers, frame-rate, resolutions, address of segments and dependency layer ID. The Requester component is based on libcurl library. This Requester in charge of all TCP connections. Requester requests the segments from DASH server according to the parsed information which are stored inside DASH Framework. The *Decoder* in libdash uses libav as decoding library, and it

is locked until receives the segment from Requester. After decoding, the decoded frames are rendered by QT Renderer.

We reference libdash architecture and modify some components to support SVC streaming. Fig. 5.4 is our DASH Client architecture. We modify the DASH Framework and Requester to support SVC streaming and insert new component called Extractor (Sec. 5.2.3) to our DASH client. We also switch the UI from QT to Android, design interfaces for DASH Client, and integrate with our SVC Decoder.

5.2.2 Supporting H.264/SVC MPD Format

The general MPD file does not support SVC streaming. The structure of MPD for non-scalable video streaming is not the same as the structure of MPD for scalable video streaming, such as attributes and location of elements. For instance, each Representation contains one initial segment in general MPD, but there is only one initial segment for SVC format MPD. The initial information is located in Representation section for non-scalable streaming. For SVC streaming, initial information is located in Initialization section. The MPD Parser checks Representation section first, if initial segment is not found, then checks Initialization section. For non-scalable streaming, Requester requests one initial segment followed by one media segment. The number of requested initial segments is the same as media segments. For the SVC streaming, Requester requests the initial segment at beginning and remainder segments are media segment. But the SVC segment cannot be decoded by SVC decoder directly, we need an Extractor to deal with SVC media segment.

5.2.3 SVC Segment Extractor

The SVC decoder unable to decode segments directly. The reason is that segments are ISOBMFF format. Each SVC segment contains a lot of boxes , so segments cannot be decoded by SVC decoder. In order to solve this problem, we implement the Extractor. The purposes of Extractor are parsing boxes to get information and obtaining media data from segments. Once Extractor receives the segment, Extractor starts to parse the boxes.The information we concerned are layer ID, segment length, number of frames from downloaded segment and media data. If Extractor receives based layer segment, Extractor reconstructs the media data frame by frame. If Extractor receives the layer-dependency segments, Extractor reconstructs the media data into decodable form for SVC decoder. The approach of reconstruction for multiple layers is interleaving. For each iteration, Extractor inserts the NAL header followed by the corresponding frame data from lowest layers segment to highest layer segment and combines into one decodable frame data. After reconstructing, SVC decoder is able to decode this kind of data.

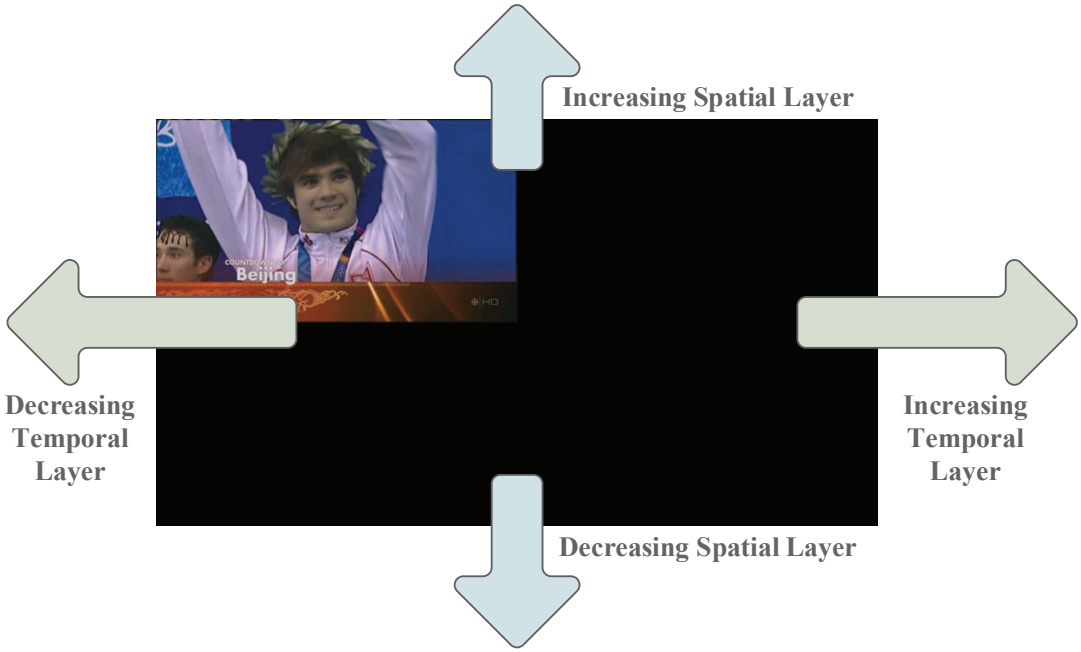


Figure 5.5: Client with gesture for switching spatial and temporal.

5.3 Switching Event Handler

This section describes the user interface of switching events which include switch to higher or lower resolution and frame-rate. At first, we design menu as our user interface to trigger switching events (Fig. 1.1) which is designed in our previous work [15]. As our knowledge, android API provides few kinds of menu, Option Menu and Context Menu. For Option Menu, user clicks the menu button to show the switch options and selects one switching event. For Context Menu, user presses on the screen for 1 second and options are showed on the screen, then selects switching event option. This approach, using menu to show options for user selection, needs at least two operations, showing option list and selection.

For reducing the number of operations, we design the gesture as our new interface for switching. We also increase the display size by removing the title and menu bar. Gesture only has one operation which is sliding. We define four directions for different switching events, like Fig. 5.5. Sliding up and sliding down mean switch to higher and lower resolutions, respectively. Sliding left and sliding right mean switch to lower and higher frame-rate, respectively. Once user selects a switching event, the switching signal is sent to MPEG-DASH Client and SVC Decoder. MPEG-DASH Client according to switching signal to request proper number of segments, and SVC Decoder according to spatial ID and temporal ID to decode the video frames.

The scope of this work is focus on the real implementation of MPEG-DASH with SVC streaming system. So far, the switching events of our SVC client are triggered by the user.

The SVC client can use the algorithms to trigger the switching events. Algorithms have two strategies for requesting segments: enhance the current segment quality or download the segments for future time slots. The decision of algorithms are based on available bandwidth, client buffer state, etc. Using algorithms to download the proper segment is our future work.

Chapter 6

Experiments

6.1 Multi-core SVC Decoder

6.1.1 Videos and Setup

We consider five HD videos: `doc`, `jeux`, `soap`, `sport`, and `talk`, which are available online [6]. The videos are provided by a leading broadcast company in Canada, and each of them lasts for 6 mins 15 secs, at 24 FPS. We encode each video using JSVM into three spatial layers ($S = 3$): 960x544, 480x272, and 240x144, and each GOP contains 16 frames, which leads to five temporal layers ($T = 5$). We fix the quantization parameter at 32; there is a single fidelity layer ($Q = 1$). The average video quality of the complete streams across all videos is 44.16 dB in PSNR, and more details are summarized in Table 6.1. We conduct the experiments on: (i) a dual-core tablet with a 1.4 GHz CPU, 1 GB memory, and a 1280x800 screen and (ii) a quad-core smart phone with a 1.5 GHz CPU, 1 GB memory, and a 1280x720 screen. We decode each video with different tuples $<t, s, q>$ and H, and we report the average FPS (frame rate) over each video. We also report the memory and energy overhead of our decoder.

6.1.2 Evaluation Results of SVC Decoder

We present sample results with $q = 0$ and $t = 5$.

Performance gains. We first report the performance improvement by using H decoder threads. Figs. 6.1 and 6.2 present the FPS values at 960x544 resolution ($s = 2$) on the tablet and the smart phone, respectively. We observe clear FPS increases for most videos when H increases from 1 to 3. For example, playing `jeux` on the smart phone with $H = 1$ achieves an FPS of 12.08, while doing so with $H = 3$ leads to an FPS of 19.39, a 60% gain. However, the performance gain saturates at $H = 4$, and the FPS gradually decreases when H goes beyond 4. This can be attributed to the thread synchro-

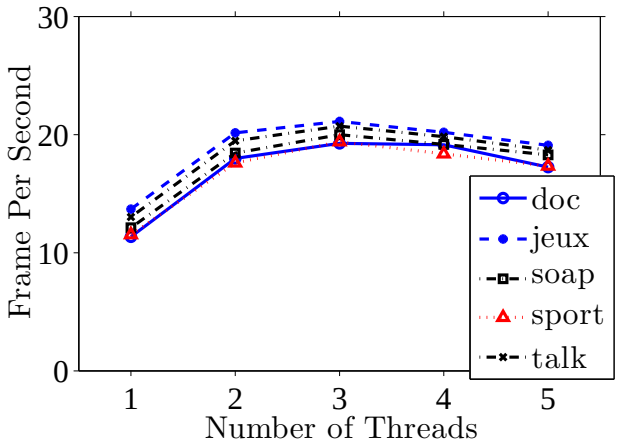


Figure 6.1: FPS, 960x544 videos on a tablet.

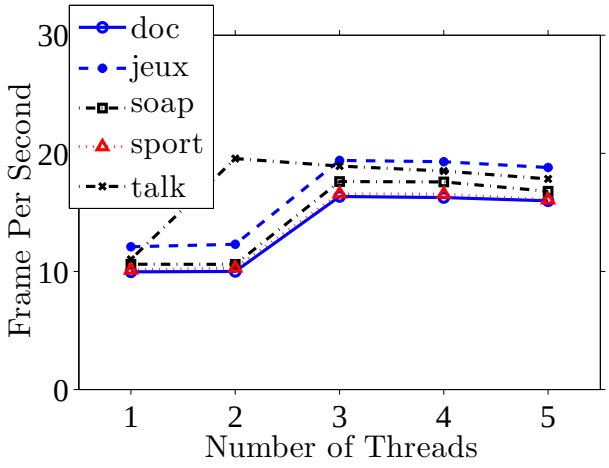


Figure 6.2: FPS, 960x544 videos on a smart phone.

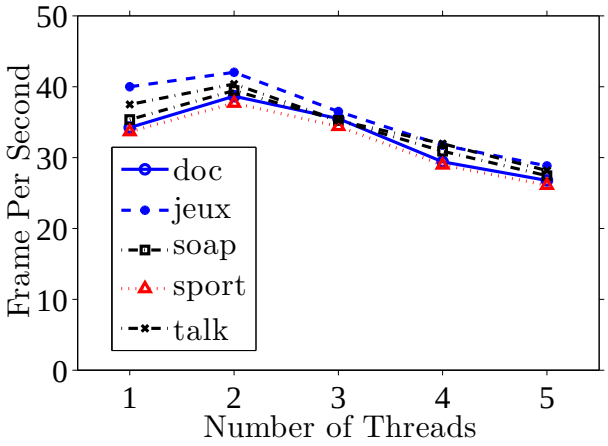


Figure 6.3: FPS, 480x272 videos on a tablet.

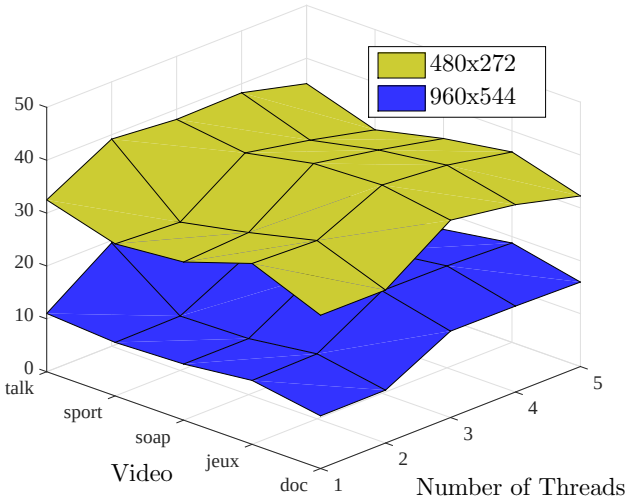


Figure 6.4: Trade-off between FPS and resolution on a smart phone.

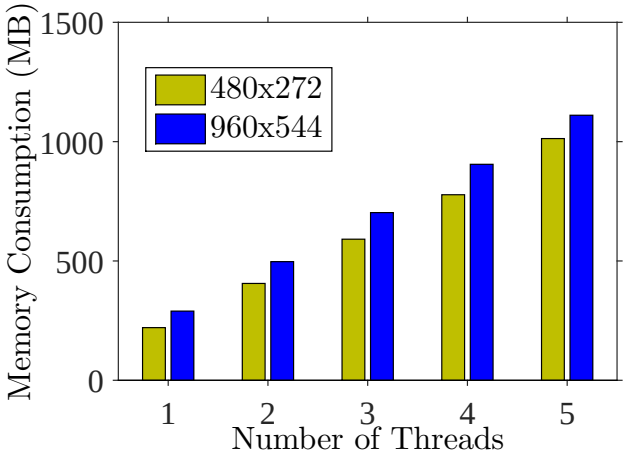


Figure 6.5: Memory consumption of our decoder, results from `sport`.

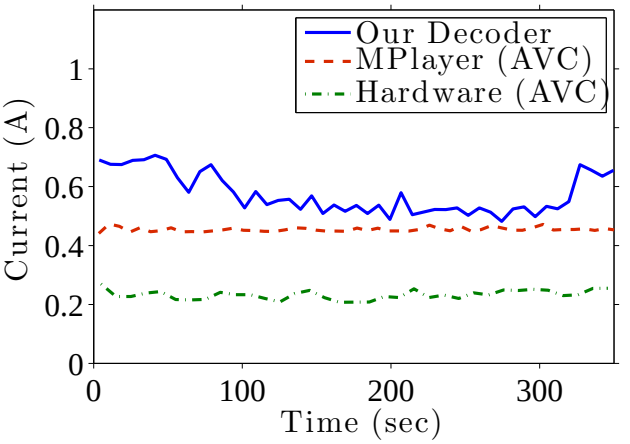


Figure 6.6: Power consumption comparison on a smart phone, results from `doc`.

nization overhead, and the competition among our decoder threads and other Android processes. We suggest setting $H = 3$ for 960x544 videos.

Fig. 6.3 presents the FPS values at 480x272 ($s = 1$) on the tablet. This figure shows a smaller performance gain, compared to Fig. 6.1. However, note that the achieved FPS values in Fig. 6.3 are as high as 42.03, much higher than the coded FPS of 24. In other words, decoding 480x272 videos incurs lower computational complexity and thus leaves smaller rooms for performance improvement. Indeed, the performance drops once $H > 2$. We suggest setting $H = 2$ for 480x272 videos.

Tradeoff between resolutions and frame rates. Fig. 6.4 compares the FPS at different resolutions on the smart phone. It is clear that decoding 480x272 videos is at least two times faster than decoding 960x544 videos. This reveals an important tradeoff: for real-time decoding at 960x544, we *must* reduce the frame rate to 12 FPS ($t = 3$). This indicates that a user may choose high resolution or high frame rate, but *not both*. Similar observations can be drawn from the results obtained from the tablet.

Memory consumption. We report the memory consumption of decoding 960x544 videos in Fig. 6.5, which shows that our decoder consumes more memory when H increases. Nonetheless, the total memory consumption at most $\sim$1 GB, which is the common specification of medium- to high-end mobile devices at the time of writing.

Power consumption. We use Agilent 66321D mobile communications DC source [1] to measure the power consumption of our SVC decoder. We also measure the power consumption of mplayer-android [3] and the default hardware decoder for comparisons; these two players only support H.264/AVC videos. For fair comparisons, we encode the videos in AVC videos with the same quantization parameters. We report the device-level power consumption with display brightness set to 50%. Fig. 6.6 presents the measured currents on the smart phone. This figure shows that although decoding SVC videos is much more complex, our SVC decoder only incurs small power overhead, as low as 7% during some time periods, compared to mplayer-android. We present the average power consumption in Table 6.2. This table shows that our SVC decoder consumes $\sim$26% (smart phone) and $\sim$31% (tablet) more power than software-based mplayer, and mplayer consumes $\sim$94% (smart phone) and $\sim$26% (tablet) more power than the hardware decoder.

6.2 Scalable Video Streaming over HTTP

6.2.1 Setup

We have also implemented an end-to-end H.264/SVC streaming testbed over HTTP. The testbed has a Linux server and an Android client using our multi-core H.264/SVC

Table 6.1: Bitrate and PSNR of Videos

Resolution	doc	jeux	soap	sport	talk
Bitrate (kbps)					
240x144	450.60	282.77	376.46	619.60	254.19
480x272	1425.71	809.41	1018.51	1760.64	909.11
960x544	4637.29	2446.46	3133.69	5228.77	3304.90
PSNR (dB)					
240x144	41.72	45.22	43.36	42.02	42.65
480x272	42.21	45.98	44.48	42.74	42.47
960x544	42.98	46.04	45.37	43.67	42.70

Table 6.2: Average Power Consumption (in Watts)

Decoder	doc	jeux	soap	sport	talk
Smart Phone					
Our Decoder	2.13	2.06	2.09	2.15	2.03
Mplayer (AVC)	1.68	1.60	1.67	1.69	1.65
Hardware (AVC)	0.86	0.88	0.81	0.88	0.83
Tablet					
Our Decoder	7.94	7.84	7.87	7.96	7.89
Mplayer (AVC)	6.22	6.37	6.20	6.27	6.32
Hardware (AVC)	5.26	5.79	5.21	5.33	5.22

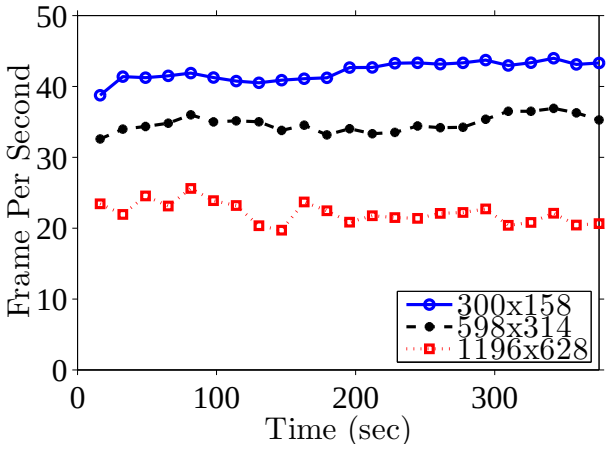


Figure 6.7: Sample FPS of three resolutions, results from `jeux`.

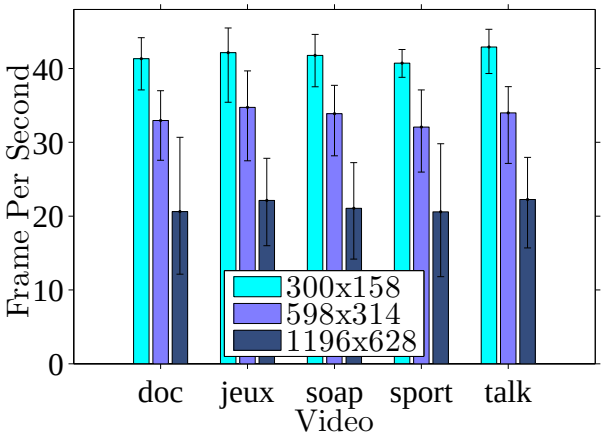


Figure 6.8: Mean FPS of all videos with different resolutions.

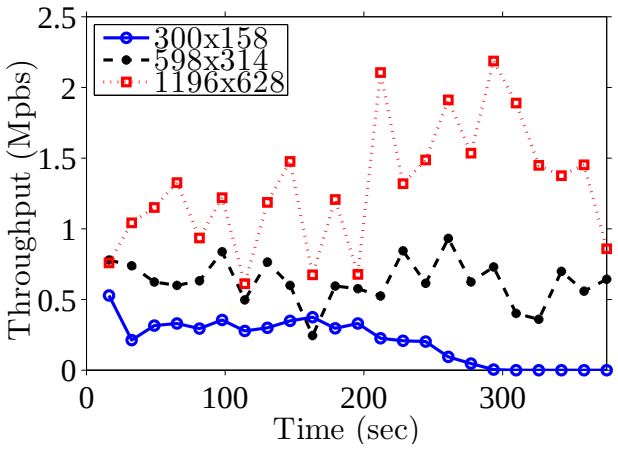


Figure 6.9: Sample throughput of three resolutions, results from `jeux` over 3G.

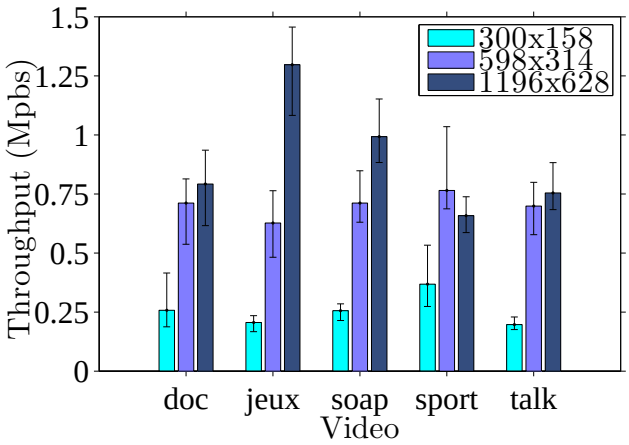


Figure 6.10: Mean throughput of all videos with different resolutions over 3G.

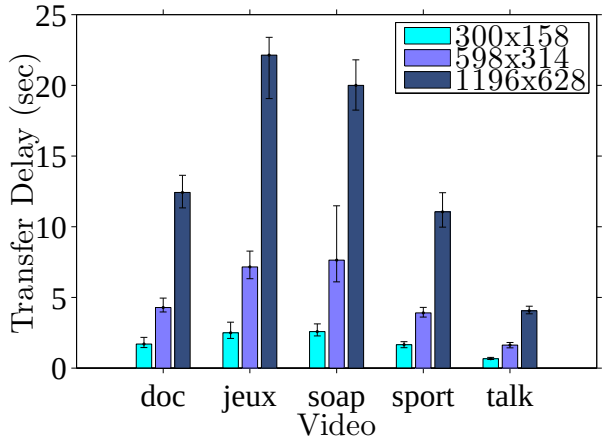


Figure 6.11: Mean transfer delay of all videos with different resolutions.

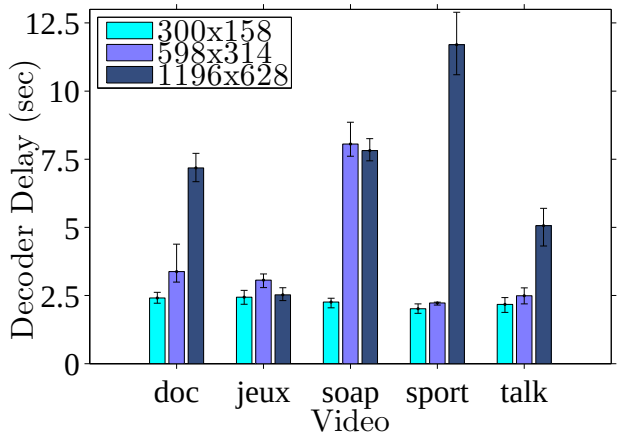


Figure 6.12: Mean decoder delay of all videos with different resolutions.

decoder. We configure each segment with $g = 8$ GOPs, number of decoder threads $H = 3$, the circular buffer thresholds $l = 1.5$ and $b = 15$ MB. We encode the five HD videos into three resolutions: 300x158, 598x314, and 1196x628, which better fit the screen resolution of our quad-core 1.5 GHz Android phones. We stream each video over the WiFi and 3G cellular networks, with different resolutions, in our lab. We repeat each experiment five times, and report average, minimum, and maximum performance whenever appropriate. The considered performance metrics are: (i) FPS, (ii) throughput, and (iii) initial delay. The initial delay is further divided into two parts: transfer delay, which refers to the time to fill up the circular buffer with l bytes, and decoder delay, which is the time to render the first frame. We currently use a conservative heuristic to determine the pre-buffering time, which leads to higher initial delay for some videos. It is our future work to address this issue.

6.2.2 Evaluation Results of HTTP streaming

Frame rate. We plot the instantaneous FPS of a sample run of streaming `jeux` over a WiFi network in Fig. 6.7. This figure reveals that our HTTP scalable video streaming client achieves 41+, 32+, and 20+ FPS with different resolutions, which are fairly acceptable for mobile video streaming. Fig. 6.8 reports the mean FPS of five runs for all videos, and the errorbars indicate the minimum and maximum FPS. This figure is consistent with the results in Fig. 6.7.

Network throughput. We plot the achieved network throughput when streaming scalable videos at different resolutions. Figs. 6.9 and 6.10 present the throughput over 3G from `jeux` and the mean throughput over 3G from all videos, respectively. These figures show that our HTTP scalable streaming client fully utilizes the available bandwidth: generally, higher resolution leads to higher network throughput.

Initial delay. We present the initial delay, which is divided into the transfer and decoder delay. We plot these two delays in Figs. 6.11 and 6.12. These two figures reveal that lower resolution leads to shorter delay. One way to leverage this property for minimizing initial delay is to first play a video with the lowest resolution, and then switch to the desired resolution once the buffer is filled. In particular, by doing so, the total delay is between $\sim$2.5 and $\sim$4.0 secs as illustrated in Figs. 6.11 and 6.12.

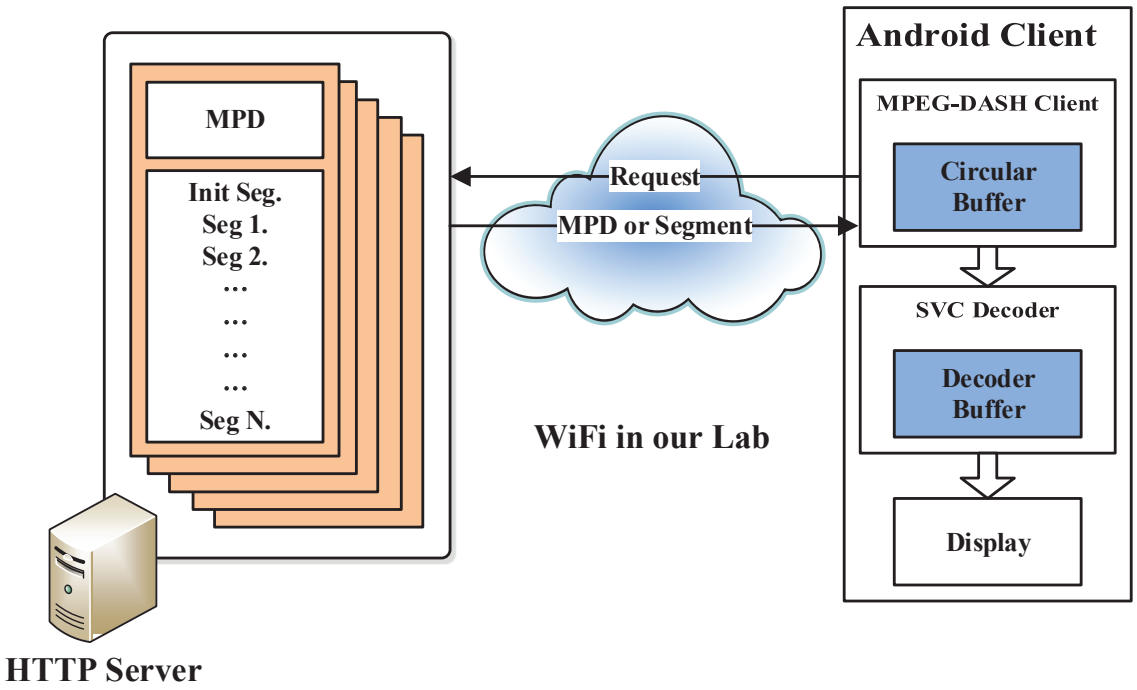


Figure 6.13: Streaming setup of MPEG-DASH.

6.3 Evaluation of MPEG-DASH Client

6.3.1 Setup

We port the MPEG-DASH standard into our H.264/SVC streaming testbed. Fig. 6.13 is an architecture of our testbed with MPEG-DASH standard. Our testbed consists of Linux web server and an Android client with our multi-thread SVC decoder and MPEG-DASH Client module. In our Linux web server, we encode five HD videos into three resolutions ($S = 3$): 320x180, 640x360, and 1280x720. We also generate corresponding MPD and chop each video into multiple segments. We set the segment length to 5.12 seconds, the GOP size is 16 frames, and each segment contains $g = 8$ GOPs. In our android client, we set circular buffer thresholds to $l = 1.5$ and $b = 15$ MB. We conduct the experiment on quad-core 1.5 GHz Android phones with 1 GB memory and 1280x720 screen size.

6.3.2 Evaluation Results of MPEG-DASH Client

We show the average throughput of five videos with different resolutions for MPEG-DASH streaming in Fig. 6.14. This figure reveals that the average throughput of MPEG-DASH Client achieves up to 15 Mbits per second for 1280x720 video streams. For 320x180 video streams, the average throughput of MPEG-DASH Client achieves at least 2.5 Mbits per second. We observe the trend of average throughput is the same among this

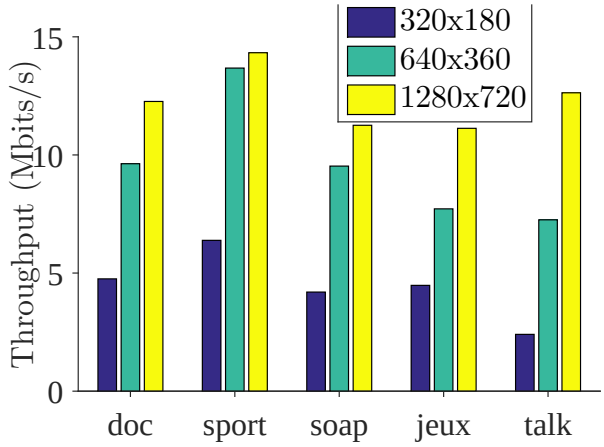


Figure 6.14: Mean throughput of all videos with different resolutions over WiFi.

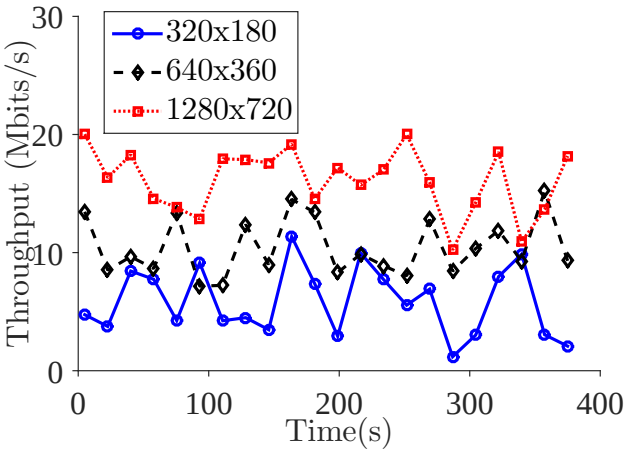


Figure 6.15: Sample throughput of three resolutions, result from `jeux` over WiFi.

5 different types of videos. In other words, the higher resolution streaming has higher throughput. It means the utilization of the available bandwidth is high in our MPEG-DASH Client.

Fig. 6.15 is the instantaneous FPS of sample run result from streaming `jeux`. The result of this figures is the same as our expectation. While streaming higher resolution videos, MPEG-DASH achieves higher throughput For some sample points, the throughput of higher resolution is less than the throughput of lower resolution. As we know, the reasons are: (i) Bandwidth condition is changing while downloading so the throughput is decreased. (ii) The free space of circular buffer is not enough for new segments, so the requester threads are stopped. The network condition is varied over time, therefore, a more efficient buffering strategy and scheduling algorithm is necessary. This will be our future work to further improve the system.

6.4 Effective SDL Rendering

6.4.1 Setup

We use the same testbed architecture as Fig. 6.13 in Sec. 6.3 and replace the Android Renderer API with SDL library in our H.264/SVC decoder. In this experiment we encode the five HD videos into three spatial layers ($S = 3$): 320x180, 640x360, and 1280x720 with 24 frame per second. We generate MPD and chop segments on the Linux server. We stream each video over WiFi in our lab and repeat 5 times. We use quad-core 1.5 GHz Android phones with 1 GB memory and 1280x720 screen as our experimental streaming client. We report the FPS for each video with varied resolutions and number of decoder threads on Android smart phone.

6.4.2 Evaluation Results of SDL Rendering

Fig. 6.16 shows average FPS of the five videos with different number of decoder threads. The FPS is enhanced by increasing the decoder threads to $H = 2$ on multi-core device. When number of decoder thread is set to 2, the FPS is increased at least 1.5 time than using only one decoder thread. The performance of FPS is decreased for most videos when number of decoder thread is ≥ 3. The reasons are the overhead of multi-threading synchronization and the resources competition among the processes, which includes our SVC decoder and other Android applications. This result is not consistent with Sec. 6.2, because the system architectures are not the same.

Fig. 6.17 shows the instantaneous sample points of FPS result from `sport`. Our SVC decoder achieves $\sim$60 FPS and $\sim$40 FPS for 320x180 and 640x360 video, respectively.

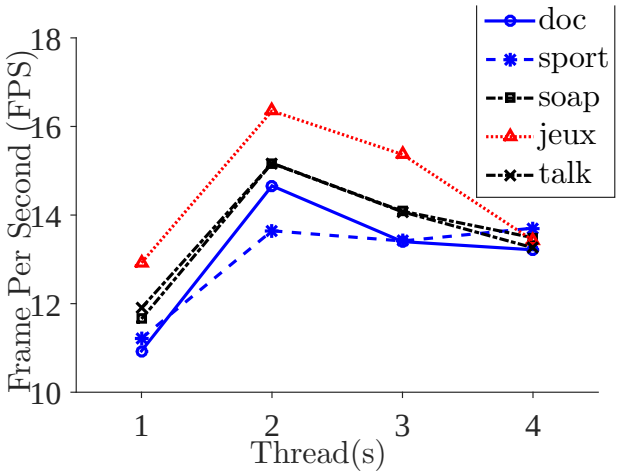


Figure 6.16: Mean FPS of all 1280x720 videos with different decoder threads.

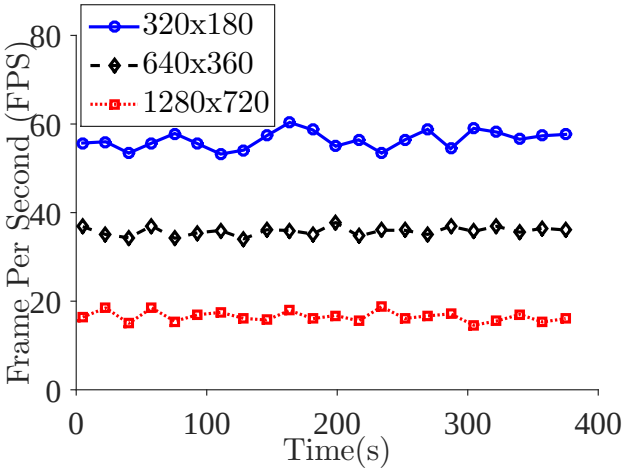


Figure 6.17: Sample FPS of three resolutions, results from `sport`.

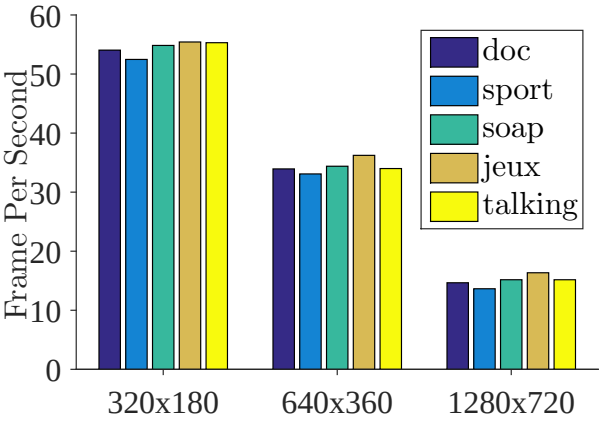


Figure 6.18: Mean FPS for all videos with three resolutions.

For 1280x720 video stream, our SVC decoder achieves ∼20 FPS. Fig. 6.18 is the average FPS of the five videos under different resolution. This figure reveals that our SVC decoder achieves at least 50 average FPS and 30 average FPS for 320x180 and 640x360 video streams, respectively. For the highest resolution 1280x720 video stream, our decoder achieves ∼15 average FPS. The results show that our SVC decoder can serve real-time streaming for 320x180 and 640x360 videos. Our SVC decoder still can be further improved, since it cannot achieve 24 FPS for 1280x720 videos. We will give directions in Sec. 7.2.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we have developed an end-to-end scalable streaming testbed, consisting of a multi-threaded SVC decoder and HTTP streaming server/client. Our experimental results show that our SVC decoder can decode 480x272 videos in real-time, and does not incur too much memory and power overhead. While our SVC decoder is not a replacement of a hardware-based solution, we believe that our decoder will stimulate more research on SVC-related systems, and encourage manufacturers to massively produce SVC decoder chips. We have also evaluated the proposed HTTP streaming server/client. Via extensive experiments, we show the practicality and efficiency of our proposed end-to-end scalable streaming testbed over HTTP for mobile devices. For example, streaming scalable videos over live 3G and cellular networks lead to high frame rate, $\sim$42 FPS, and short initial delay, $\sim$2.5 secs.

We further extend our testbed to support MPEG-DASH standard and implement the MPEG-DASH client in our SVC decoder. In order to handle segments for SVC decoder, we implement the extractor in the MPEG-DASH client. Our experimental results show that the throughput of our testbed with MPEG-DASH standard achieves at least 15 Mbits per second for 1280x720 videos. We replace the Android Renderer API with SDL library. SDL library and our SVC decoder are written in native-code (i.e. C/C++), which can render the decoded frame in native side without copying the frame data to Java front-end. We design a new switching event handler based on the SDL event queue, which uses gestures to switch the spatial and temporal layers, in our client. The experimental results of our decoder with SDL library show that our decoder achieves at least 50 FPS when streaming 320x180 videos.

7.2 Future Work

Our experimental results reveal an important trade-off between resolutions and frame rates. Due to resource constraints of mobile devices, a user may only pick either high resolution or high frame rate. The user's decision depends on the video genres, device types, and even user preferences. There are active projects, such as Song et al. [21], which conducts user studies and tries to model the Quality-of-Experience (QoE) of mobile video streaming. These user studies do not leverage scalable videos and only consider very few resolutions and frame rates for each video. Our end-to-end mobile scalable video streaming testbed allows us to conduct large-scale user studies using H.264/SVC videos on commodity Android devices. This enables us to derive a more flexible QoE model.

In this work, user triggers the switching events based on its preferences. The switching events can also be triggered by scheduling algorithms according to the available bandwidth, state of buffer, and so on to decide either to enhance the quality level or download more segments for later usage. Employing such scheduling algorithms in our client and obtaining a more flexible QoE model are among our future tasks.

The performance of our SVC decoder still has room to improve when rendering 1280x720 video. We found that the bottlenecks of our client are decoding and rendering. It is important to eliminate the bottlenecks and improve the performance of our SVC decoder. There are some possible approaches to improve the performance, such as designing more efficiency multi-threading structures, and using NEON instructions set to speed up the decoding and color space conversion. There is a more recent, emerging, scalable video coding standard, called H.265/SHVC, which is an extension of H.265/HEVC. We may use H.265/SHVC as our scalable video coder and evaluate the performance once the standard is finalized.

Bibliography

[1] User's guide, 66321B/D mobile communications dc source. http://cp.literature.agilent.com/litweb/pdf/5964-8184.pdf, 2005.

[2] MPEG-DASH - Part 1: Media presentation description and segment formats. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=65274, 2011.

[3] Mplayer-Android project page. https://github.com/ajeet17181/mplayer-android, 2011.

[4] Scalable video coding (SVC video)–Radvision. http://www.radvision.com/Solutions/Video-Communications-Technology/Scalable-Video-Coding/, 2012.

[5] Video conferencing technology platform–Vidyo. http://www.vidyo.com/technology/, 2012.

[6] Video library and tools. http://nsl.cs.sfu.ca/wiki/index.php/Video_Library_and_Tools, 2012.

[7] Opern Source libdash library page. https://github.com/bitmovin/libdash, 2013.

[8] GPAC open source multimedia framework. http://gpac.wp.mines-telecom.fr/, 2014.

[9] Multi-Core SVC Decoder on Android page. https://github.com/nmsl/svc_android_project, 2014.

[10] Cisco visual networking index: Global mobile data traffic forecast update, 2014–2019. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html, 2015.

[11] M. Blestel and M. Raulet. Open SVC decoder: A flexible SVC library. In *Proc. of ACM Multimedia'10*, pages 1463–1466, Firenze, Italy, October 2010.

[12] J. Chong, N. Satish, B. Catanzaro, K. Ravindran, and K. Keutzer. Efficient parallelization of H.264 decoding with macro block level scheduling. In *Proc. of IEEE International Conference on Multimedia and Expo (ICME'07)*, pages 1874–1877, Beijing, China, July 2007.

[13] Y. S. de la Fuente, T. Schierl, C. Hellge, T. Wiegand, D. Hong, D. D. Vleeschauwer, W. V. Leekwijck, and Y. L. Louédec. idash: Improved dynamic adaptive streaming over http using scalable video coding. In *Proc. of ACM Multimedia Systems (MMSys'11)*, pages 257–264, San Jose, CA, February 2011.

[14] ITU-T Study Group 9. Subjective video quality assessment methods for multimedia applications. *ITU Series P: Audiovisual quality in multimedia services*, 1999.

[15] Y. Li, C. Chen, T. Lin, C. Hsu, Y. Wang, and X. Liu. An end-to-end testbed for scalable video streaming to mobile devices over http. In *Proc. of IEEE Conference on Multimedia and Expo (ICME'13)*, San Jose, CA, July 2013.

[16] C. Müeller, D. Renzi, S. Lederer, S. Battista, and C. Timmerer. Using scalable video coding for dynamic adaptive streaming over HTTP in mobile environments. In *Proc. of European Signal Processing Conference (EUSIPCO'12)*, pages 2208–2212, Bucharest, Romania, August 2012.

[17] C. Müller and C. Timmerer. A test-bed for the dynamic adaptive streaming over HTTP featuring session mobility. In *Proc. of ACM Multimedia Systems (MMSys'11)*, pages 271–276, San Jose, CA, February 2011.

[18] H. Richter, B. Stabernack, and E. Muller. Adaptive multithreaded H.264/AVC decoding. In *Proc. of Asilomar Conference on Signals, Systems, and Computers (Asilomar'09)*, pages 886–890, Pacific Grove, CA, November 2009.

[19] H. Schwarz, D. Marpe, and T. Wiegand. Overview of the scalable video coding extension of the H.264/AVC standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 17(9):1103–1120, September 2007.

[20] C. Sieber, T. Hossfeld, T. Zinner, P. Tran-Gia, and C. Timmerer. Implementation and user-centric comparison of a novel adaptation logic for dash with svc. In *Proc. of IEEE Integrated Network Management (IM'13)*, pages 1318–1323, Ghent, Belgium, May 2013.

[21] 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 ACM Multimedia'11*, pages 403–412, Scottsdale, AZ, November 2011.

[22] G. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, July 2003.