GREENVIDEO: A FRAMEWORK FOR
ENERGY-EFFICIENT VIDEO STREAMING TO HANDHELD DEVICES

by

Xin Li

A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Information Security and Assurance

Committee:

Dr. Songqing Chen, Thesis Director

Dr. Robert Simon, Committee Member

Dr. Fei Li, Committee Member

Dr. Sanjeev Setia, Chairman, Department of Computer Science

Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: May 3, 2013

Spring Semester 2013
George Mason University
Fairfax, VA

GreenVideo: A Framework for Energy-Efficient Video Streaming to Handheld Devices

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science at George Mason University

By

Xin Li
Bachelor of Science
Huazhong University of Science and Technology, Wuhan, Hubei, China, 2003

Director: Dr. Songqing Chen, Professor
Department of Computer Science

Spring Semester 2013
George Mason University
Fairfax, VA

Copyright © 2013 by Xin Li
All Rights Reserved

Dedication

I dedicate this thesis work to my family and many friends. A special feeling of gratitude to my loving wife, Lin Wang, whose words of encouragement and push for tenacity ring in my ears. I would like to gratefully thank my parents and sister for always supporting me.

I also dedicate this thesis to my many friends who have supported me throughout the process. I will always appreciate all they have done, especially Dr. Zhan Ma and Dr. Mian Dong for helping me develop my technology skills.

I dedicate this work and give special thanks to my son Jonathan Li for being there for me throughout the entire process.

Acknowledgments

I wish to thank my committee members who were more than generous with their expertise and precious time. A special thanks to Dr. Songqing Chen, my committee chairman for his countless hours of reflecting, reading, encouraging, and most of all patience throughout the entire process. Thank you Dr. Robert Simon and Dr. Fei Li for agreeing to serve on my committee.

Table of Contents

List of Tables

List of Figures

Abstract

GREENVIDEO: A FRAMEWORK FOR ENERGY-EFFICIENT VIDEO STREAMING TO HANDHELD DEVICES

Xin Li M.S.

George Mason University, 2013

Thesis Director: Dr. Songqing Chen

With the exponentially growing smartphone market, more and more people desire to have a multipurpose handheld device that not only supports voice communication and text messaging, but also provides video streaming, multimedia entertainment, etc. A crucial problem with a handheld device that enables video streaming is how to prolong the battery lifetime given the large amount of energy consumed by video transmission, decoding, and presentation. Thus, it is essential to have an in-depth understanding of power consumption required by video transmission, decoding, and presentation. The knowledge can be utilized to identify power-hungry components and to provide insight into how power consumption can be reduced for such components.

Our experiments show that energy is mainly consumed by the wireless radio, the application processor, and the display system in a typical handheld device. More specifically, our research focuses on the streaming services where video is streamed over the 3G/4G network, decoded on ARM application processors and rendered on HD displays.

To tackle the problem, we first propose power optimization algorithms for the 3G/4G radio, the ARM processor, and the display subsystem individually. By integrating all the algorithms, we build GreenVideo, a framework for energy-efficient video streaming to

handheld devices. The system is validated with a large amount of real world videos from YouTube and the experimental results show that GreenVideo achieves significant power reductions.

Chapter 1: Introduction

1.1 Background

In recent years we have witnessed the booming of the smartphone market. More and more rich content is made possible to people through smartphones. Although the smartphone pervasion brings huge convenience to our daily life, power optimization on these devices continues to be a relevant problem due to the out of pace improvement of battery capacity. Rich content applications, by nature, require substantially more computations and data transmission, thus inevitably make the battery lifetime of smartphones a bottleneck. One rich content application, mobile video streaming, which is also among the most popular mobile applications (apps) according to recent studies [1, 2], demands prohibitively high power consumption. Consequently, power optimization solutions for mobile video streaming are with unprecedented importance.

In this thesis, we first conduct experiments to gain in-depth understanding of power consumption required by video transmission, decoding, and presentation. The results show that energy is mainly consumed by the wireless radio, the application processor, and the display system on a typical handheld device. More specifically, our research focuses on the streaming services where video is streamed over the 3G/4G network, decoded on ARM application processors, and rendered on HD displays. To tackle the problem, we first propose power optimization algorithms for the 3G/4G radio, the ARM processor, and the display subsystem individually. Specifically, we design GreenTube on power optimization of 3G/4G radio, Codec DVFS on power optimization of decoding running on ARM processor, and content-adaptive display power optimization.

By integrating all the algorithms, we build GreenVideo, a framework for energy-efficient video streaming to handheld devices. The system is validated with a large amount of real

world videos from YouTube and the experimental results show that GreenVideo achieves significant power reductions.

The rest of this thesis is organized as follows. Chapter 2 presents GreenTube, a system proposed and designed for 3G/4G radio power optimization. Chapter 3 describes Codec DVFS which aims to optimize power consumption of video decoders running on ARM processors. Chapter 4 presents our work on display power optimization, i.e., content-adaptive display power optimization. In Chapter 5, we present GreenVideo which consolidates all the individual power optimization schemes. Chapter 6 concludes this thesis.

Chapter 2: GreenTube

In this chapter we describe GreenTube, a system to optimize 3G/4G radio power consumption for mobile video streaming.

2.1 Introduction

We start with a brief overview of mobile video streaming, then introduce power states defined in 3G/4G network, and finally discuss the video streaming support in state-of-the-art smartphone systems.

2.1.1 Mobile Video Streaming

A video streaming system can be implemented in many ways. It can be built using a conventional Content Delivery Network (CDN) architecture such as Youtube, Hulu, etc, or purely peer-to-peer (P2P) such as PPLive, PPStream, or even hybrid CDN-P2P [3]. HTTP, RTSP, P2P protocols are used to implement different streaming systems. Among them, our focus is the popular and dominant HTTP video streaming. For instance, YouTube, Hulu, etc, use the HTTP protocol for content delivery.

Videos delivered over the Internet can be encapsulated using many container formats such as AVI, FLV, MP4, MKV, etc. FLV was dominant in the past due to the popularity of flash. Recently, however, streaming video contents are increasingly contained in MP4 format, rather than FLV, especially for HD contents. To support our study, we use a crawler to download 4 Terabytes of the most popular Youtube videos over a 45-day span. From this massive dataset, videos with 720p and 1080p resolutions are all encapsulated in the MP4 container. Specifically, MP4 is a multimedia container format that can combine different multimedia streams (such as audio, video, subtitle and images). Similar to other

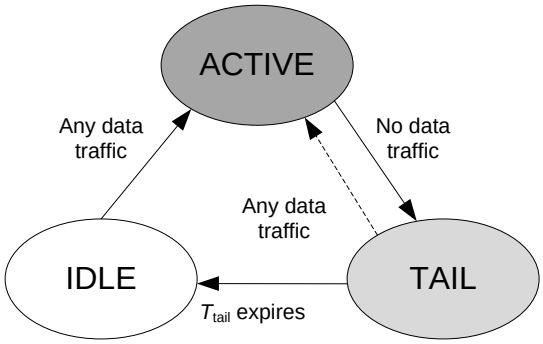

Figure 2.1: Simplified 3G/4G RRC States Transition

modern container formats, MP4 allows streaming over the Internet. A separate hint track is used to include streaming information (such as timing information) in the file. Without loss of generality, we use MP4 videos exclusively in this thesis.

2.1.2 Power States in 3G/4G Network

We next discuss the necessary background on state machine behavior and corresponding power characteristics of the 3G/4G network.

Both 3G and 4G networks share the similar generalized radio resource control (RRC) states, IDLE, ACTIVE and TAIL as shown in Figure 2.1. For a typical data exchange, assuming the device starts at the IDLE state, the current state is promoted to the ACTIVE state for data transmission with a certain delay (i.e., noted as promotion delay [4,5]). After data transmission, it is usually demoted from the ACTIVE state requiring higher radio resource and radio power consumption to the TAIL state with lower radio resource and power consumption, until the tail timer (noted as T_{tail}) expires and finally puts the device into the IDLE state.

Given the state machine for 3G/4G networks, the power trace on smartphones can be easily explained. As shown in Figure 2.2 for video streaming over Verizon LTE network, we observe that the network activities accurately match the different states with associated power levels. Smartphones consume almost zero power in IDLE state. Power consumption

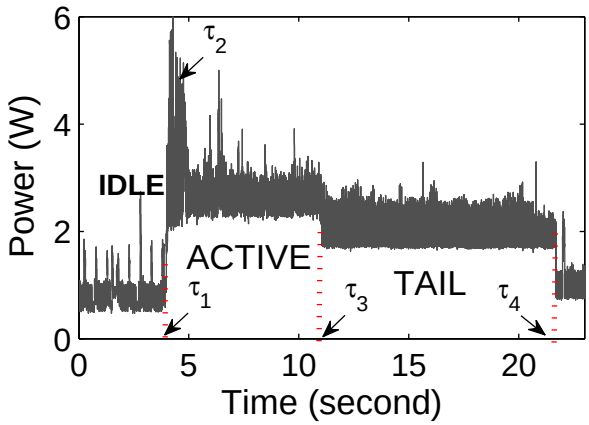

Figure 2.2: 3G/4G RRC Power States Trace

rises when the smartphone is promoted from the IDLE state to the ACTIVE state after sending a packet at time τ_1. After a certain promotion delay, the device starts active data transfer at time τ_2, until it enters the TAIL state with lower power at τ_3. After the tail timer expires at τ_4, the smartphone enters the IDLE state and the allocated radio resource is released. Note that our findings are consistent with the work presented in [4].

2.1.3 HTTP Streaming Support in Smartphones

We next introduce HTTP streaming support in Android, one of the most popular smartphone OSes.

In the Android Multimedia Framework, the major components involved in video playback include data source providers (such as disk file), video container demuxers, video codecs, audio codecs, and a media player. The HTTP streaming downloading behaviour is largely determined by the data source provider which interacts with the HTTP server and feeds the data to the media player. The HTTP streaming data source provider is stateful and there are three states: *Fetching*, *Idle*, and *Keep Alive*. In the data source provider, the video data downloaded from the HTTP server is held in a large cache in memory. The size of the cache is bounded by a high threshold. There is also a low threshold, the lower bound of the cache, which is used to improve the playback smoothness. By default, the

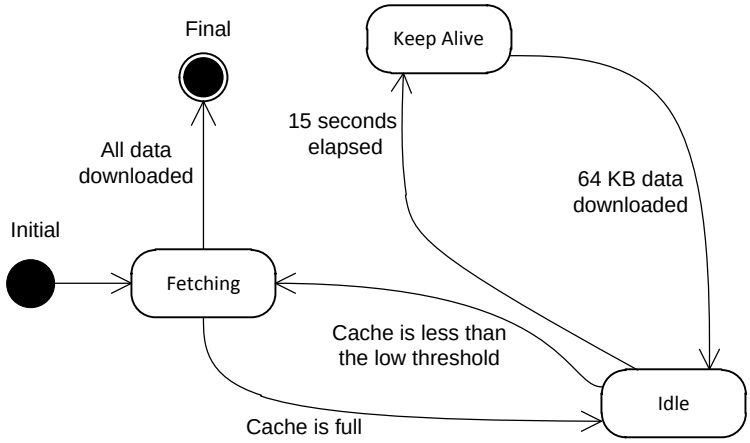

Figure 2.3: Android HTTP Data Source Provider State Machine

high threshold is set to 20 MB and the low threshold is set to 4 MB.

During an HTTP video streaming session, the data is downloaded in chunks. We now explain this process in detail using the state machine in Figure 2.3. Initially, when the user starts a video streaming session, the data source provider enters into the *Fetching* state and continuously downloads the video data from the HTTP server. When the cache size reaches the high threshold, it transits to the *Idle* state. By default, while in the *Idle* state, a keep-alive mechanism is automatically triggered every 15 seconds to transition into the *Keep Alive* state. On entering into the *Keep Alive* state, 64 KB video data is downloaded and then the state transits back into the *Idle* state. Once the size of the cached video content falls below the designated low threshold, the state moves to *Fetching* and another continuous downloading begins.

The different states of the HTTP data source provider can be mapped to the different states of the 3G/4G networks. In other words, the different data source provider states correspond to different power consumption states. As a result, the power consumption during video playback is largely determined by this periodic downloading behaviour. This relationship is further studied and explored in the following sections.

One exception worth noting is that although the Android stock YouTube app directly

utilizes the Android Multimedia Framework for video playback, it uses a hidden, undocumented feature which disables the keep-alive mechanism and closes the TCP connection every time the cache is filled up. This effectively eliminates all data traffic between the HTTP server and the smartphone in the *Idle* state. The benefit of doing so is further discussed in later sections.

2.2 Motivation

We next report three studies that directly motivate the design of GreenTube: mobile video streaming power characterization, YouTube usage by smartphone users, and LTE network speed measurement.

2.2.1 Power Characterization

2.2.1.1 Methodology

Video Content: As briefly mentioned earlier, we developed a YouTube crawler that automatically downloads 720p video files from different categories on the YouTube website. Using this crawler, we collected more than 16, 000 720p video files from YouTube over 40 days. The average size of the video files is 94 MB, the average duration is 358 seconds, and the average bitrate is 2200 Kbps. We randomly chose 500 out of the 16, 000 video files for the video streaming power characterization.

Choice of Smartphones: For power characterization, we chose the Galaxy S, Galaxy S II, and Galaxy Nexus smartphones. Table 2.1 lists their detailed specifications.

Table 2.1: Smartphone Specifications

Model	Galaxy S	Galaxy S II	Galaxy Nexus
Processor	Hummingbird	Snapdragon	OMAP4
Memory	512MB	1GB	1GB
Network	ATT HSPA	Tmobile HSPA+	Verizon LTE
Android	2.3.6	2.3.6	4.0.3

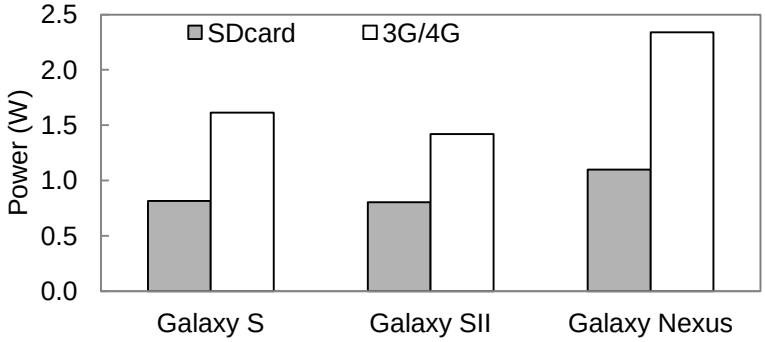

Figure 2.4: Power Consumption Characterization

Video Streaming Setup: We stored all the downloaded video files on an Internet-accessible HTTP server. We developed a simple Android video player based on the Android Multimedia Framework. This video player reads a list of test videos stored on the HTTP server and automatically plays the video files with a 10-second interval between two consecutive playbacks.

Power Measurement: We measure the power consumed by the smartphones using the Monsoon Power Monitor [6]. The smartphones are powered by the power monitor so they never run out of the battery during the experiments. The power monitor supplies current to the phone and is able to sample the current drawn by the device at a frequency of 5000 Hz. The power consumption data recorded by the power monitor software is time-stamped. The simple android video player also logs when a video playback begins and finishes. With the 10-second interval between playbacks, the offline analysis tool can extract the power consumption data for each video playback from the power trace recorded by the power monitor.

2.2.1.2 Results

For each smartphone, we measure the average power consumption for playing a video from SD card and from the HTTP server. Figure 2.4 shows the average power consumption of the two cases for three smartphones. As shown in the figure, average power consumption by the three smartphones is 1.4 W, 1.2 W, and 2.3 W, respectively. Considering the battery

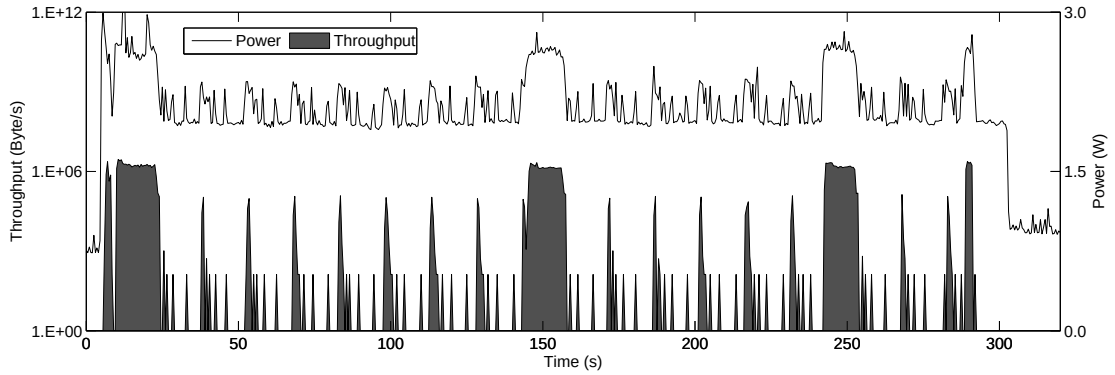

Figure 2.5: Power Trace of 720p HTTP Video Streaming Using Verizon LTE Network

capacity of each smartphone, we estimate the battery life of the three smartphones for mobile video streaming is 5, 6, and 3 hours, respectively. Moreover, Figure 2.4 also shows power consumption of video streaming via 3G/4G networks is about 2X of that of playback from SD card. In other words, the 3G/4G radio accounts for about 50% of the total system power consumption. To summarize, mobile video streaming via 3G/4G network is prohibitively power hungry, and 3G/4G radio is the most significant power consumer in smartphones for mobile video streaming.

A natural question is why does 3G/4G radio consume so much power? We shall now answer this question using the power trace. Figure 2.5 depicts the transient power consumed by a smartphone (Galaxy Nexus) that is receiving a 2 Mbps video stream from the HTTP server via Verizon LTE network. The figure also shows the corresponding time-stamped data traffic between the HTTP server and the smartphone.

As shown in the figure, there are three different data traffic patterns. Each of the four large continuous chunks represent a high-speed downloading session in the Fetching state to fill the default 20 MB cache; each of the 15-second interval peaks corresponds to a 64 KB downloading in the Keep-Alive state; and the tiny peaks are caused by TCP zero window probes sent from the HTTP server and the corresponding TCP ACKs sent as responses by the smartphone. The sequence of data traffic patterns strictly conforms to the state machine presented in section 2.1.3. Note that when the cache is full, i.e., the size of data in the cache

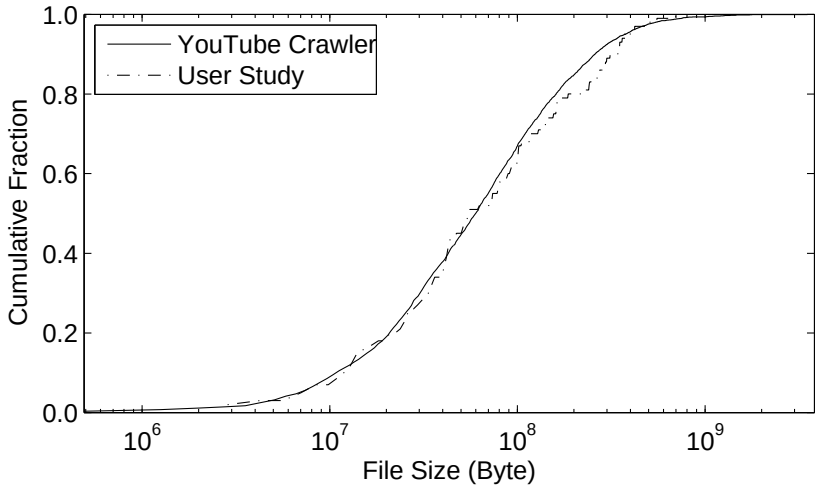

Figure 2.6: File Size Distribution of YouTube Videos in Our Experiment

reaches the high threshold, the smartphone stops reading data from the TCP socket and the TCP socket receive buffer will soon be filled up by data sent from the HTTP server. This causes a zero TCP receive window at the smartphone side and according to the TCP protocol, the HTTP server will proactively send probes to check when the smartphone's TCP receive window reopens and is ready to receive more data.

Interestingly, the power consumption data shows an almost identical pattern. The highest power consumption happens during the four continuous high-speed downloading, each of the 64 KB downloading slots causes a much lower power consumption peak and the tiny peaks in the data traffic generate several much narrower power consumption peaks. Obviously, there is a 10-second TAIL power state after the last chunk of continuous high-speed downloading which completes the streaming task. After the 10-second TAIL state, the LTE modem enters the IDLE state and the power consumption drops to a dramatically lower level. Given that the interval between 64 KB downloadings is 15 seconds (which is longer than the 10-second TAIL state), it is natural to wonder why the LTE modem never enters the IDLE state before the last continuous downloading. The reason is that the TCP zero window probes and the corresponding TCP ACKs are small packets that effectively bring the LTE modem from the TAIL state back to the ACTIVE state and thus prohibit

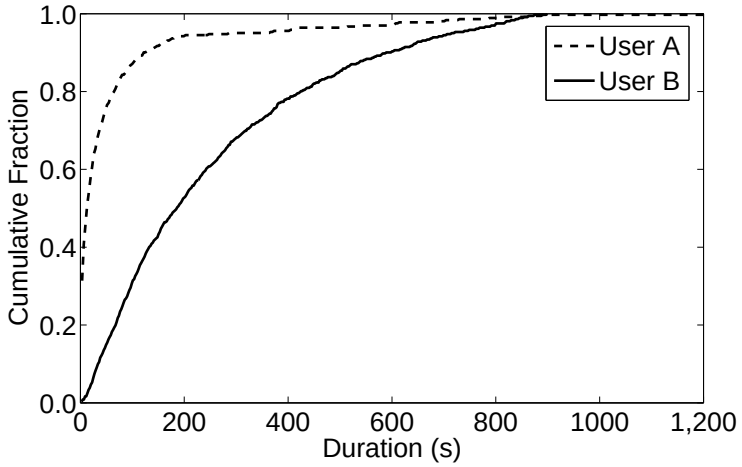

Figure 2.7: Distribution of User Watching Time

the LTE modem from staying in the TAIL state long enough.

Thus, excessive power consumption will occur when more than two downloading sessions are needed, i.e., the file size of the video is more than 20 MB. Figure 2.6 shows the file size distribution of the videos downloaded by our crawler. As shown in the figure, more than 80% of the 720p videos are larger than 20 MB and thus suffer from excessive power consumption.

2.2.2 YouTube Usage by Smartphone Users

2.2.2.1 Methodology

Subjects: We recruited 10 unpaid participants from the US branch of a Korean company. Their ages range from 25 to 36. Out of the 10 participants, 4 are female. Every participant is a self-described, heavy, mobile-YouTube user who watches at least 10 YouTube videos on his/her smartphone per day, according to our survey.

Apparatus: We implemented a logger to record the user's inputs in the YouTube application on Android-based smartphone, i.e., Galaxy Nexus. Unfortunately, we are unable to modify the YouTube application directly due to its close-source nature. Instead, we implemented the logger in the underlying media player of the Android Multimedia Framework. The logger is able to record both video file information and user's inputs during a video

playback when an upper layer application, including YouTube, calls for the services provided the Multimedia Framework. We are particularly interested in the following information:

- The user ID (UID) of the application which uses the media player service.
- The size and the duration of the video file.
- The time at which the user starts, pauses, resumes, seeks and stops the video playback.

Since Android assigns a unique UID for each application, we could filter the log and collect the traces for the YouTube application afterwards.

Procedures: We provided each participant a Galaxy Nexus, with the logger pre-installed, as his/her personal phone for two months. We paid their phone bills for the two months and asked them to use the Verizon LTE network whenever available. Finally, we collected the phones and analyzed the logs after the two month study.

2.2.2.2 Results

First, most video sessions are short. We define a video session from a start operation to a stop operation. We also treat a forward seek operation as starting a new video session if the seek destination is out of range of the caching content and thus requires a new downloading session. In particular, 80% of YouTube sessions in smartphones are less than half of the corresponding video durations. As a result, downloading the whole video into a large cache and turning off the radio will not solve the problem because of the waste in both data transfer and power. Instead, we should still use multiple download sessions in a video streaming session.

Second, user watching patterns are different. Figure 2.7 shows two typical users. For User A, around 80% of the session time is less than 70 seconds. While for User B, over 50% of the session time is larger than 200 seconds. Therefore, we should apply different downloading schedules to different users.

These observations naturally inspired us to explore the power consumption with different cache sizes. We studied this by running simulations with a 75 MB, 312 second long 720p

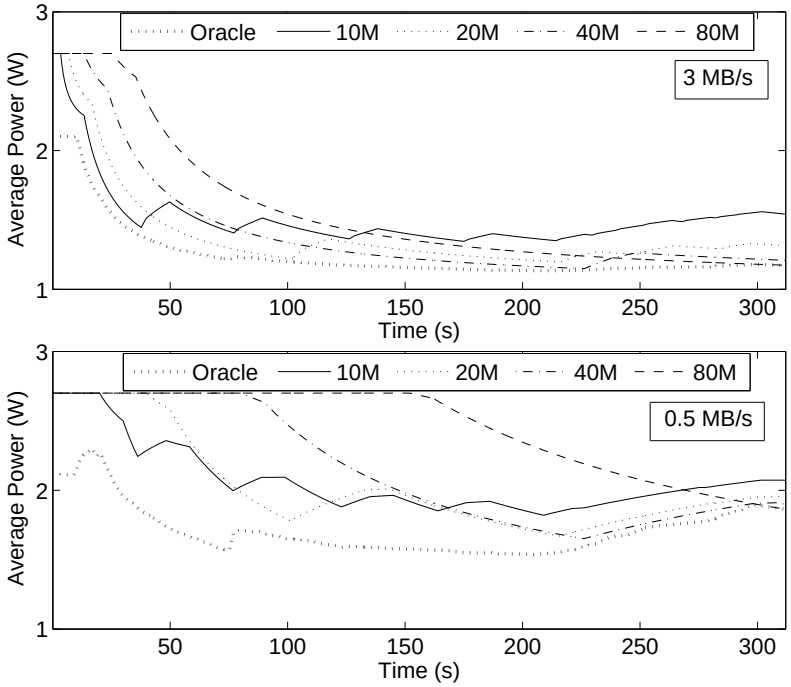

Figure 2.8: Average Power Consumption with Different Cache Sizes under Different Network Speeds

video file. The power consumption is calculated using the Verizon LTE modem power profile measured in section 2.2.1. We chose four different high threshold values: 10 MB, 20 MB, 40 MB and 80 MB. We also ran an Oracle scheme which knows exactly how much data the user consumes and thus downloads all the data in only one shot.

The top figure of Figure 2.8 shows the simulation result under 3 MB/s network speed. The bottom one is simulated under 0.5 MB/s network speed and the difference between the two simulations is explained in Section 2.2.3.2. In this section we focus on the top figure.

The Oracle always has the lowest average power consumption no matter when the user stops the video playback. When the watching time is short, for example, less than 60 seconds, the average power consumption of a small cache size is better than that of a larger one and the difference is usually non-trivial. With longer watching time, a choice of large

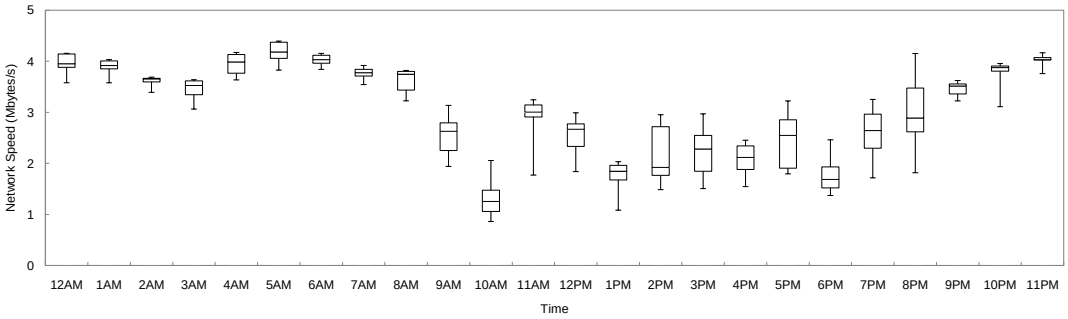

Figure 2.9: Verizon LTE Network Speed

cache size is preferred since it has better average power consumption. Another trend is that as the watching time increase, the average power consumption difference becomes much smaller, especially for the cache sizes 40 MB and 80 MB.

After observing the different user viewing behaviors, we now propose a better way to manage the cache size in order to improve power consumption: for users with short watching durations (User A in Fig. 2.7), a small cache size is preferred while for users with long watching durations (User B in Fig. 2.7), a large cache size is a better choice.

2.2.3 LTE Network Speed

2.2.3.1 Methodology

We developed a speed-test application running on the Galaxy Nexus. The application downloads a 100 MB file from the HTTP server every 60 minutes. The downloaded data is silently discarded. During the downloading, the application measures the average speed of each 5-second period and logs the speed to a disk file.

2.2.3.2 Results

We ran the speed-test application for a week and analyzed its log file. Figure 2.9 shows the result of a typical day. As shown in the figure, the average network speed measured at each hour varies dramatically throughout the day. We observe higher speed during night when the number of active users is much less. Most importantly, the network speed varies sharply

even within a short period. For example, at 8PM, the highest speed and lowest speed is 4 MB/s and 1 MB/s, or 4X difference.

To study the impact of network speed variation on power consumption, we performed a simulation similar to that described in section 2.2.2. However, we used different network speed settings in this simulation. Figure 2.8 shows the result. The network speeds for the top figure and the bottom figure are 3 MB/s and 0.5 MB/s, respectively. The average power consumption under 0.5 MB/s is higher for all cache sizes. The power consumption difference between different cache sizes are much larger and the curves converge much slower than the curves with the higher network speed. In observance of such obvious impact of network speed on power consumption, we must adapt the cache size to real-time network speed.

2.3 GreenTube Design

We now describe the design of GreenTube as motivated by the results from the preceding studies.

2.3.1 Key Design Decisions

The results from the preceding motivational studies lead us to make the following major design decisions for GreenTube.

Close the TCP connection with the HTTP server when the cache is full. To reduce excessive power consumption as illustrated in Section 2.1, GreenTube disconnects from the HTTP server after each downloading session when the cache is full. Note that there are many other options to disconnect from the HTTP server. For example, we could have turned off the 3G/4G radio and released the IP address. However, we chose to close the TCP connection for two reasons. First, both alternatives need to wait for the base station to re-allocate an IP address when a new downloading session needs to start. This re-allocation process can take as long as 5 seconds in Verizon LTE network according to our measurements. In contrast, to re-establish a new TCP connection with the HTTP server requires less than 0.2 seconds. This 25x delay reduction significantly decreases the risk of

streaming interruption and excessive power wastage. Second, unlike turning off the radio or releasing the IP address, closing the TCP connection will not affect background services that may require network connection, such as email checking and data synchronization. Actually, such TCP disconnection feature has also been adopted by the YouTube application that is shipped with the Android system, as described in Section 2.3. GreenTube improves over YouTube in the following aspects.

Record user-specific watching history. GreenTube keeps a record of the user watching time for each streaming session and utilizes this historical record to generate a probability distribution for the user watching time. Such a probability distribution, as demonstrated in Section 2.2, has a significant impact on average power consumption of the streaming session. At the beginning of each downloading session, GreenTube uses the probability distribution to estimate the expected watching time for the user and chooses the optimal cache size that leads to minimal expected power consumption. Note that GreenTube treats forward seek operation as starting a new video session if the seek destination is out of range of the caching content and thus requires a new downloading session.

Adapt cache size to network speed. GreenTube samples network speed in each downloading session. As discussed in Section 2.3, network speed has a huge impact on power consumption and varies dramatically over a short period. As a result, the optimal cache size should adapt to the network in real time. In each downloading session, GreenTube estimates the network speed every second using size of the downloaded data from the last second. Based on this estimation, GreenTube adjusts the cache size accordingly.

Set maximal cache size based on user's choice. GreenTube chooses a cache size from a fixed set. The maximal cache size is equal to the worst-case amount of excessive downloaded data in a streaming session and thus should be determined by the user based on his/her tolerance for data-transfer wastage. Besides the user's choice, GreenTube also sets a cache-size upper limit because the marginal power saving will diminish as the cache size increases. To quantitatively decide such a limit, we measure the average power consumption by Galaxy Nexus for video streaming via Verizon LTE network using different cache sizes. Given a

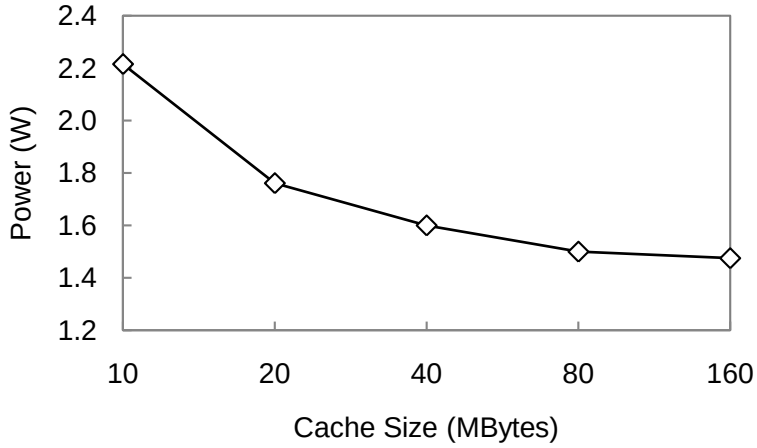

Figure 2.10: Power Consumption by Galaxy Nexus for Video Streaming via Verizon LTE Network Using Different Cache Sizes

cache size, we used it for all the test videos and then compiled Figure 2.10 to show the average power consumption corresponding to each cache size. As illustrated in the figure, the marginal power saving is negligible when the cache size is larger than 80 MB. Therefore, GreenTube sets the maximum cache size to be the lower of 80 MB and the user's choice.

To summarize, GreenTube fetches a video file from the HTTP server in multiple downloading sessions and disconnects from the server after each downloading session ends. The starting and ending time of each downloading session (except for the starting time of the first session and ending time of the last session) is determined by the cache size. GreenTube judiciously schedules downloading sessions by adaptively adjusting the cache size according to user watching history and real-time network speed. We call this adaptive adjustment process *Dynamic Cache Management*.

2.3.2 Dynamic Cache Management

We now present the *Dynamic Cache Management* (DCM) algorithm. The basic idea is to adaptively adjust the high threshold value of the cache based on the sampled network speed and the expected watching time calculated from the probability distribution for the user's watching time. After each second has elapsed, the algorithm computes the high threshold

value that correponds to minimal energy consumption assuming that the user will stop watching at the expected watching time.

Algorithm 1 shows pseudocode for the DCM algorithm. th_h and th_l are the high threshold value and low threshold value, respectively. Initially the high threshold value is set to 10 MB while the low threshold value always remains fixed at 4 MB. D is the video duration while A is the size. This information is extracted from the MP4 header after the header has downloaded completely. $S_{history}$ is the discrete probability distribution calculated from the user viewing history and it is normalized. Every time the user watches a new video, the data is added to $S_{history}$. B is the set of candidate high threshold values. P_{active}, P_{tail}, P_{idle} and T_{tail} are power profile related power consumption parameters derived from our measurements. Specifically for the Verizon Galaxy Nexus, P_{active} is the average power consumption when the LTE modem is in the ACTIVE state, P_{tail} is the average power consumption in the TAIL state, P_{idle} is the average power consumption in the IDLE state and T_{tail} is the maximum time the LTE modem stays in the TAIL state before it transits into the IDLE state.

The DCM algorithm is event-driven. It adjusts th_h every second during a continuous, active downloading. There is no need to adjust th_h in the *Idle* state because it will have absolutely no impact on the downloading behaviour. The adjustment is done in two steps: 1) Calculate the expected watching time using function EXPECTED-WATCHING-TIME; 2) Using the expected watching time, estimate the energy consumption for each high threshold candidate and thus determine the optimal th_h.

We obtain the energy-consumption estimate by simulating the periodic behaviour observed in our video playback power measurement. Given the current time t_{cur}, the expected watching time t_{exp}, the current network speed s, the size of the video A, the size of the consumed video content a_{cur}, the duration of the video D, and the various power-profile parameters (P_{active}, P_{tail}, P_{idle}, and T_{tail}), the function OPTIMAL-CACHE-SIZE first estimates the time spent in the ACTIVE, TAIL and IDLE states, respectively, in the time period from t_{cur} to t_{exp}. It then uses this time estimate to derive the energy consumption.

Algorithm 1 The DCM Algorithm

Wait for *event*
if *event* is downloading started **then**
  Set *timer* to expire in 1 second
else if *event* is timer expired **then**
  t_{exp} = EXPECTED-VIEWING-TIME(t_{cur})
  x_{opt} = OPTIMAL-CACHE-SIZE(t_{exp}, t_{cur})
  $th_h = x_{opt}$
  Set *timer* to expire in 1 second
else if *event* is downloading stopped **then**
  Cancel the *timer*
end if

function EXPECTED-VIEWING-TIME(t_{cur})
  $\widehat{S} = \{p_i | p_i \in S_{history}, i \cdot \frac{D}{|S_{history}|} \geq t_{cur}\}$
  $S'_{history} = normalize(\widehat{S})$
  $n = |S'_{history}|$
  $t_{exp} = \sum_{i=1}^{n} p_i \cdot (t_{cur} + (i-1) \cdot \frac{D}{n}),\ p_i \in S'_{history}$
  return t_{exp}
end function

function OPTIMAL-CACHE-SIZE(t_{exp}, t_{cur})
  s = current downloading speed
  Set bitrate $r = \frac{A - a_{cur}}{D - t_{cur}}$
  $e_{opt} = +\infty$
  $x_{opt} = 0$
  for each b in B **do**
    $t = t_{cur}$
    $t_{active} = 0$
    $t_{idle} = 0$
    while $t < t_{exp}$ **do**
      $t_d = min(t_{exp} - t, \frac{b - th_l}{s})$
      $t_d = min(\frac{A - a_{cur} - th_l}{s}, t_d)$
      $a_{cur} = a_{cur} + s \cdot t_d$
      $t_{active} = t_{active} + t_d$
      if $a_{cur} + th_l \geq A$ **then**
        $t_v = \frac{s \cdot t_d + th_l}{r}$
      else
        $t_v = \frac{s \cdot t_d}{r}$
      end if
      $t_v = min(t_v, t_{exp} - t)$
      $t_{idle} = t_{idle} + min(t_v - t_d, T_{tail})$
      $t = t + t_v$
    end while
    $t_{idle} = (t_{exp} - t_{cur}) - t_{active} - t_{tail}$
    $e = P_{active} \times t_{active} + P_{tail} \times t_{tail} + P_{idle} \times t_{idle}$
    if $e < e_{opt}$ **then**
      $e_{opt} = e$
      $x_{opt} = b$
    end if
  end for
  return x_{opt}
end function

2.4 Evaluation

In this section, we present the implementation of GreenTube and evaluate its performance.

2.4.1 Implementation

To evaluate the performance of GreenTube, we implemented a C++ prototype based on the Android Multimedia Framework. We created a new data-source provider that wraps the default HTTP streaming data-source provider and incorporates our dynamic cache management algorithm. Configuration settings allow upper layer android applications to decide at runtime whether to use our dynamic cache management. By doing so, the impact to the existing code is minimum and the modification is transparent to the upper layer

android applications. The implementation changes 6 source files and there are in total around 700 lines of code. According to our measurements, the computational cost of our implementation is negligible. On average, our DCM algorithm consumes less than 100 microseconds on the Galaxy Nexus.

2.4.2 Evaluation Methodology

As mentioned in section 2.2.2, we conducted a user study with the popular YouTube application and obtained 4000 video playback traces over a period of 2 months with 10 users. By analyzing the traces, we obtained the URLs of all the videos and downloaded them from YouTube. We stored these videos on an Internet-accessible HTTP server. The videos and the corresponding traces were used to evaluate GreenTube.

We implemented a simple media player which 1) uses the dynamic cache management enabled data source provider, and 2) is able to read the user study trace, set the video source to the user watched video stored on our HTTP server, and mimic the recorded user watching behavior. To run the experiment automatically without user intervention, the media player also reads a play list containing ordered sequence of video files for testing. The power meter is used to measure the power consumption during the experiment. To automatically synchronize the video playback and the power trace, we again leverage the logger facility in the Android Multimedia Framework to log the time when each video playback starts and stops. Also, there is a 10-second interval between two consecutive video playbacks, to enable our offline analysis tool to easily extract the power traces for individual video playbacks.

The videos watched by each user were randomly divided into 8 groups. In a round-robin fashion, each one of the 8 video groups is chosen as the test set and the traces of the remaining 7 groups are used only for prediction. By iteratively choosing each of the 8 groups as the testing group, all videos watched by the users are covered.

We compare the following four schemes:

- **Android:** The default Android scheme which is discussed in section 2.1.3.
- **YouTube:** The YouTube scheme which closes the connection with the HTTP server

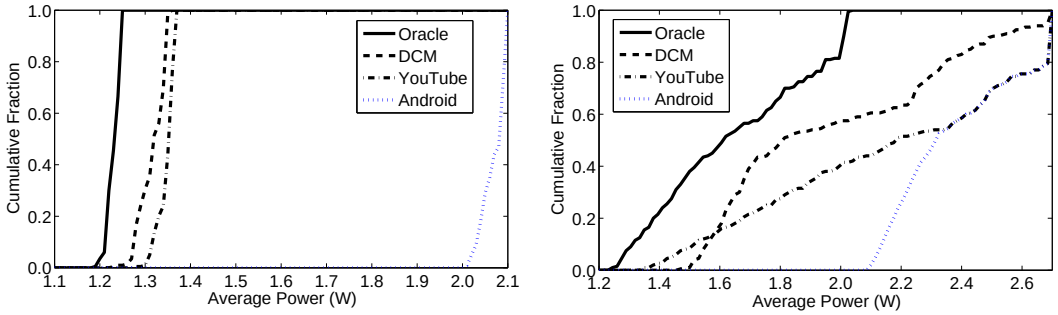

Figure 2.11: Power Consumption Distribution on Verizon LTE Network for Different Users

every time the cache is full. This greatly increases the time the LTE modem stays in the IDLE state so the power consumption can be substantially improved.

- **DCM:** Our DCM scheme, which is based on the YouTube scheme but also considers both the user behaviour and network speed variation and accordingly changes the cache size dynamically.

- **Oracle:** The Oracle scheme, which achieves the optimal power consumption by downloading exactly the amount of data consumed by the user. The Oracle scheme always knows when the user stops the video playback before it begins. Although impossible to implement practically, we include it as the optimal bound.

2.4.3 Experimental Results

We now present evaluation results from measurements and simulations.

2.4.3.1 Overall Power Reduction

Figure 2.11, 2.12 and 2.13 show the power consumption distribution of the four schemes on Verizon LTE network, ATT HSPA network, and T-Mobile HSPA+ network , respectively.

Figures on the left in each of the three pairs of figures depict users (for example, User B) who tend to finish watching the videos. Figures on the right depict users (for example, User A) who usually watch only the very beginning portion of the video.

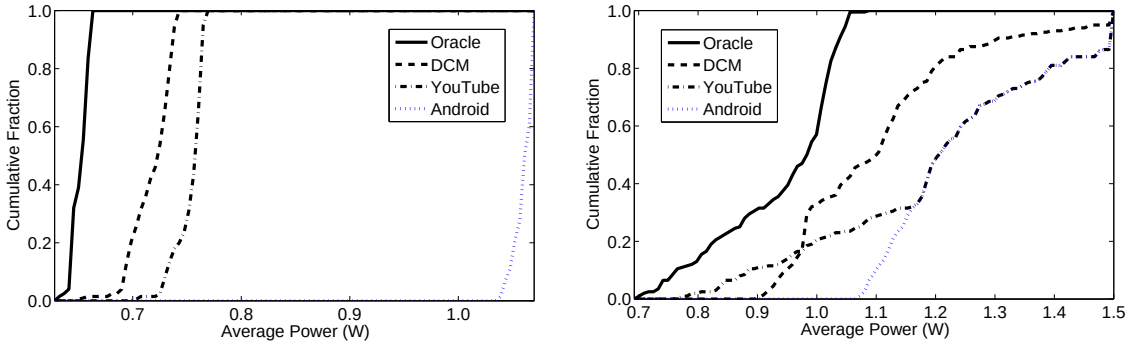

Figure 2.12: Power Consumption Distribution on ATT HSPA Network for Different Users

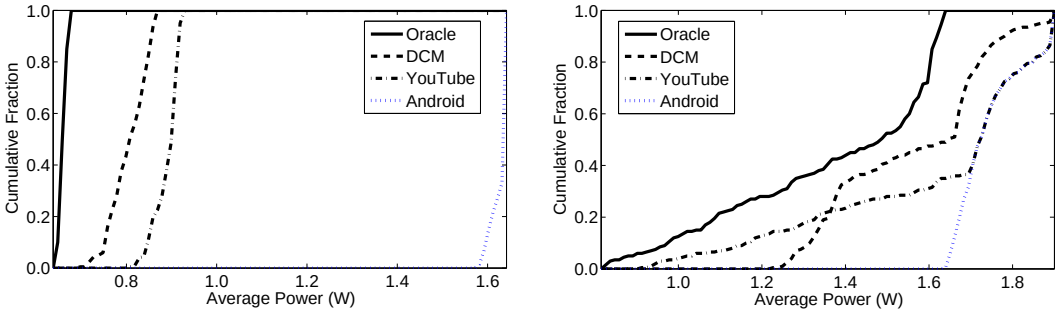

Figure 2.13: Power Consumption Distribution on T-Mobile HSPA+ Network for Different Users

For all networks, when the user tends to finish the video, our DCM scheme clearly consumes much less power than the YouTube and the Android schemes. The Android scheme has the worst power consumption and the performance gap is huge. Although both the DCM and the YouTube schemes use the "disconnect when cache is full" feature, our DCM scheme clearly outperforms the YouTube scheme because of the accurate watching-time prediction and the dynamic selection of a larger cache.

For users who stop watching a video very early, our DCM scheme is much better than the Android scheme. Comparing against the YouTube scheme, the DCM scheme performs slightly worse in a few cases while in other cases, the advantage is much more obvious than for the other type of users. Our analysis of the traces shows that when the predicted watching time is larger than the actual watching time, our DCM scheme tends to choose

a cache size larger than 20 MB which is used in the YouTube scheme. Thus our DCM scheme downloads more data and consumes more energy than the YouTube scheme. For other cases, our DCM scheme tends to choose a smaller cache size and the power saving is substantial.

Chapter 3: Codec DVFS

In this chapter we present Codec DVFS, a system to reduce power consumption incurred by video decoding.

We consider the case in which a video stream is decoded by the H.264/AVC software decoder running on ARM processors. Although video streams are usually decoded by dedicated hardware codecs on mobile platforms, software codecs resemble similar structure as hardware codecs and the power optimization algorithm proposed in this chapter can be applied to hardware codecs as well.

Prior studies show that in mobile devices using dynamic voltage and frequency scaling (DVFS), being able to accurately predict the complexity of successive decoding intervals is critical to reduce the power consumption [7].

Generally, there are two sources of energy dissipation during video decoding [8]: the memory access and the processor cycles. In this chapter, we focus on optimizing power consumption of the ARM processor. DVFS was introduced to modern processors and major operating systems long time ago to scale the voltage and the frequency of processors according to the predicted workload. Power consumption is thus reduced by capping the voltage and frequency using various algorithms. All DVFS algorithms deployed in the Linux kernel work in an application-agnostic way in which performance statistics are used to predicate the overall system workload and the prediction is used as the indicator for the voltage and frequency setting of the subsequent cycles. However, for video decoding, in order to achieve higher power reduction, an application aware DVFS algorithm with accurate video decoding complexity prediction is desired. Moreover, for mobile operating systems such as Android, with the foreground application process dominating the processor and all background application processes suspended, the codec's complexity of the running

video playback app is even closer to the overall system workload since all other application processes are incurring minimum activities.

In this chapter, we propose a method to accurately model the computational complexity of H.264/AVC video decoding. Based on the complexity model, we design a DVFS algorithm which achieves significantly more power reduction compared to the default DVFS algorithms in Android.

The rest of the chapter is organized as follows. Section 3.1 provides necessary background information. Section 3.2 presents the H.264/AVC decoder complexity model. The Codec DVFS system and its evaluation are described in section 3.3.

3.1 Background

In this section, we first give a brief overview of dynamic voltage and frequency scaling. Next, we introduce cpufreq and its governors in Linux kernel. Finally, we describe the decomposition of the H.264/AVC decoder.

3.1.1 Dynamic Voltage and Frequency Scaling

Dynamic voltage and frequency scaling (DVFS) is a widely used technique to reduce processor power consumption. DVFS is able to reduce the power consumption of a processor by reducing the frequency at which the processor operates, as shown by

$$P = CfV^2 + P_{static} \tag{3.1}$$

where C is the capacitance of the transistor gates (which depends on feature size), f is the operating frequency and V is the supply voltage. The voltage required for stable operation is determined by the frequency at which the circuit is clocked, and can be reduced if the frequency is also reduced. This can yield a significant reduction in power consumption because of the V^2 relationship and the frequency reduction shown above.

For the same workload, when DVFS is applied to reduce the processor frequency by a

ratio of r, the time to complete the task will be roughly increased by $1/r$. On the other side, the reduction of P is at least quadratic. As a result, even with the prolonged running time, the overall power consumption will be reduced by a large margin.

3.1.2 Cpufreq and Governors

In Linux kernel, cpufreq is the subsystem that allows the frequency of the processor to be explicitly set, either automatically by an algorithm or manually by the system administrator.

The primary components of the infrastructure include:

- Cpufreq module provides a common interface to the various low-level, CPU-specific frequency control technologies and high level CPU frequency controlling policies. Cpufreq decouples the CPU frequency controlling mechanisms and policies and helps in independent development of the two.

- CPU-specific drivers implement different CPU frequency changing technologies, such as Intel® SpeedStep® Technology, Enhanced Intel® SpeedStep® Technology [9], AMD PowerNow!™, and Intel Pentium® 4 processor clock modulation.

- The cpufreq features a plugin framework to accommodate frequency-changing policy governors. A governor implements a certain algorithm to change the CPU frequency according to the corresponding criterion, for example, CPU usage. The governor in use can be changed on the fly.

  There are multiple governors shipped with the Linux kernel. These include Powersave, Performance, Ondemand, Interactive, and Userspace just to name a few. The userspace governor, unlike other governors, does not change the CPU frequency in an autonomous way and gives the userspace application and the system administrator the freedom to specify the CPU frequency.

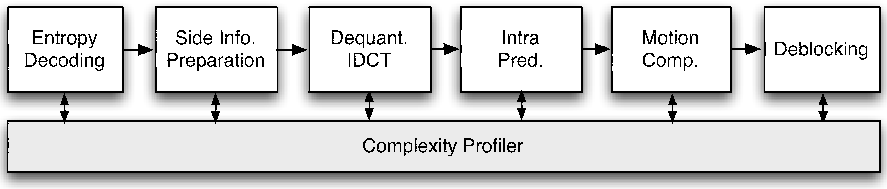

Figure 3.1: Illustration of H.264/AVC Decoder Decomposition

3.1.3 H.264/AVC Decoder Decomposition

As discussed in [10], the H.264/AVC decoder can be decomposed into the following basic decoding modules (DMs): entropy decoding (`vld`), side information preparation (`sip`), dequantization and inverse transform (`itrans`), intra prediction (`intra`), motion compensation (`mcp`) and deblocking (`dblk`) as shown in Figure 3.1. To decode a video stream, the bitstream is first fed into *entropy decoding* to obtain interpretable symbols for the following steps, such as side information (e.g., macroblock type, intra prediction modes, reference index, motion vector difference, etc) and quantized transform coefficients; the decoder then uses the parsed information to initialize necessary decoding data structures, which is so-called *side information preparation*. The block types, reference pictures, prediction modes, motion vectors, will be computed and filled in corresponding data structures for further usage. Doing so makes other decoding modules be able to focus on its own particular jobs, and such job isolation can make data preparation (for prediction purpose) and decoding more independent without interferences. The *dequantization* and *inverse transform* are then used to convert quantized transform coefficients into block residuals which are in turn summed with predicted samples, from either *intra prediction* or *motion compensation* to form reconstructed signal. Finally, the *deblocking filter* is applied to remove blocky artifacts introduced by block based hybrid transform coding structure.

3.2 H.264/AVC Decoder Complexity Measurement and Modeling

To build the complexity model, first we measure the complexity of each DM in the H.264/AVC decoder. Such measurements give us the baseline to evaluate the accuracy of the proposed complexity model.

We developed a complexity profiler to use on Galaxy Nexus with the TI OMAP 4460 [11] ARM Cortex-A9 [12] processor. We chose x264 [13] as the H.264/AVC decoder to profile due to its popularity and superb performance. The complexity is expressed in the form of the number of CPU cycles used for a certain DM. For the TI OMAP 4460, the Cortex-A9 Technical Reference Manual [14] defines the *Cycle CouNT* (CCNT) register from which the profiler can read the number of clock cycles elapsed since the register was reset. The complexity is collected on a per-frame basis. The trial run of the profiler shows that the overhead occurred by the profiler code is negligible.

Table 3.1: Essential DM and its CU in the H.264/AVC decoder

DM	Functionality	CU
`vld`	side info. parsing and quantized transform coefficients decoding	bit parsing
`sip`	side info. data structure init.	MB data structure init.
`itrans`	inverse transform module	MB dequant. & IDCT
`intra`	inverse intra prediction	MB intra pred.
`mcp`	motion-compensation	Half-pel interpolation
`dblk`	deblocking filter	α-point filtering

For each DM, we define an unique complexity unit (CU) to abstract required fundamental operations. For example, for the entropy decoding DM, the CU is the process involved in decoding one bit, whereas for the `itrans` DM, the CU is the process involved in dequantization and inverse transform for one macroblock (MB). Note that a CU includes all essential operations needed for a basic processing unit (a bit for `vld`, a MB for `itrans`) in a DM, instead of the basic arithmetic or logic Ops, such as `add`, `shift`, etc. Table 3.1

summarizes each DM and its corresponding CU.

Let $C_{\texttt{DM}}$ denote the required computational cycles to decode one frame by a particular DM, then the overall frame decoding complexity is the sum of individual complexity required by each DM. The complexity of each DM can be written as the product of $k_{\texttt{CU}}$ - the complexity of one CU, and $N_{\texttt{CU}}$ - the number of CUs required to decode each frame. Therefore, the overall frame decoding complexity is the sum of the complexity required by involved DMs, i.e.,

$$C_{\texttt{frame}} = \sum\nolimits_{\texttt{DM}} C_{\texttt{DM}} = \sum\nolimits_{\texttt{DM}} k_{\texttt{CU(DM)}} \cdot N_{\texttt{CU(DM)}}, \tag{3.2}$$

where $k_{\texttt{CU(DM)}}$ indicates the complexity of the CU for a particular DM, and $N_{\texttt{CU(DM)}}$ is the number of CUs involved in a DM.

We define the CU for each DM in such a way that k_{CU} is either constant (i.e., $k_{MBitrans}$, $k_{MBintra}$ and k_{half}) or can be predicted easily (i.e., k_{bit}, k_{MBsip} and k_{half}) using the complexity data from previous decoded frame in the same layer. To facilitate accurate complexity prediction, we propose to embed those N_{CU} that cannot be easily extracted from the video bit stream (i.e., $n_{intraMB}$, n_{nzMB}, n_{half} and n_{half}) in the bitstream, for example, in the header of the container. Assume we need 2 bytes to specify each number for a frame, 8 bytes of data is required in total per frame, which is far less than the size of the video payload. For each new frame, from either the constant or predicted k_{CU}, and the extracted N_{CU} from the bit stream, the decoder can easily predict the total complexity for this frame using 3.2.

In order to validate our complexity model, we created test bitstreams using standard test sequences, e.g., Harbor, Soccer, Ice, News, all at CIF (i.e., 352×288) resolution. These four video sequences have different types of content, in terms of texture, motion activities, etc. A large quantization parameter (QP) range, from 10 to 44 in increments of 2, is chosen to create the test bitstreams.

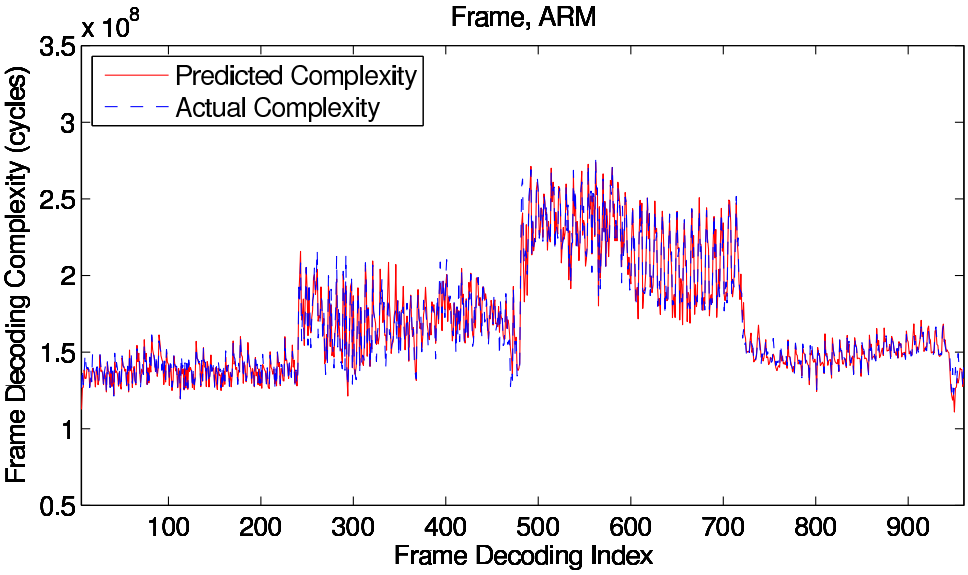

Figure 3.2: Predicated and Actual Profiled Complexity of Sample Video at QP 24

Predicted and actual cycles of per-frame decoding are shown in Figure 3.2. The experimental data for other QPs are similar as presented. The figure shows that the proposed complexity model can predict the decoding complexity very accurately.

3.3 Codec DVFS and Evaluation

We design and implement Codec DVFS as an Android video player app. The app takes two files as input: the MP4 video file and the corresponding complexity description file.

The complexity is obtained offline by analyzing the video files using the proposed complexity model. The prediction is then stored in a disk file with one line for the complexity of each video frame, in ascending order.

In order to be able to change the CPU frequency, upon launch, the app modifies the cpufreq governor to Userspace and sets the frequency on demand during runtime through the sysfs interface.

Before decoding each frame, the app calculates the required minimum frequency and

rounds it up to the nearest frequency level supported by the CPU. Considering that occasionally, there are activities other than decoding, such as disk I/O, interrupt handling, etc, it is necessary to reserve some headroom, so as to prevent workload underestimation which will otherwise cause frame loss. Based on our experience, a 8% headroom is sufficient.

We run the experiments on the Galaxy Nexus phone, which features the TI OMAP4460 ARM CPU. There are four discrete frequencies: 350 MHz, 700 MHz, 920 MHz and 1200 MHz. 300 720p MP4 video files are randomly picked from the YouTube video repository for the experiments. Similarly, we use the Monsoon power monitor to measure the power consumption. Each video file is played twice, using the Android default DVFS governor and our Codec DVFS algorithm, respectively.

On average, the power consumptions of playing the video file using the Android default DVFS and our Codec DVFS algorithm are 1.6 W and 1.45 W , respectively. In other words, the Codec DVFS is able to reduce around 9% overall system power consumption.

For videos with more motion, for example, basketball games, the Codec DVFS achieves higher power reduction. For videos with a lot of static scenes, for example, TV news, the power reduction tends to be marginal. The reason is that the default Android cpufreq governor is quite sensitive to workload peak and relatively slow in reducing the frequency when workload becomes light. As a result, the CPU was kept at an unnecessarily high frequency for a prolonged time period in the case of videos with more motion, this leads to excessive power consumption ; on the other hand, for videos with more static scenes, the frequency set by the Android default governor converges over time and the power consumption difference is not that much.

Chapter 4: Content-Adaptive Display Power Optimization

The display panel is another main source of power consumption on handheld devices. On Android devices, the display usually accounts for 40% to 70% of the total power consumption according to the system statistic data. Our measurements show that for the display, the maximum power consumption is about 600 mW on a smartphone with 4 inch display and up to 2.7 W on a tablet which has a 10 inch display.

On handheld devices, there are mainly two types of display panels: Liquid-Crystal Display (LCD) and Organic Light-Emitting Diode (OLED) display. They work very differently and have different power consumption characteristics. In this chapter, we focus on power optimization on LCD since OLED displays usually require a special hardware control circuit to manage its power consumption [15].

In order to reduce the power consumption of LCD displays, researchers [16, 17] have proposed the concept of *Backlight Scaling*, by which, the backlight of the display is dimmed dynamically to conserve its power consumption while increasing the transmittance of the LCD panel to compensate for image fidelity loss due to reduced backlight.

In this work, based on Backlight Scaling, we propose a novel way to implement display adaptation for LCD displays on Android in order to reduce display power consumption. The implementation uses OpenGL ES to dynamically scale the pixel luminance as the compensation to dimmed backlight. The computation overhead is negligible and the measured power consumption overhead is marginal compared to the savings we can achieve.

The rest of the chapter is organized as follows. Section 4.1 introduces the necessary background information. Section 4.2 presents the design of the content-adaptive display power optimization system. The implementation and the evaluation are described in Section 4.3.

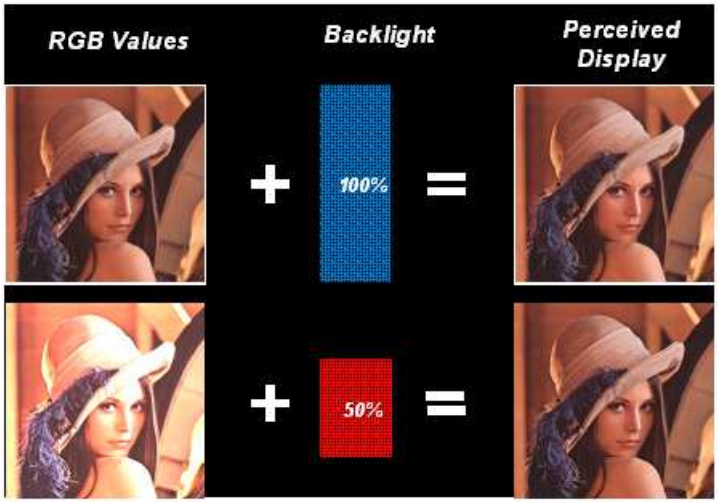

Figure 4.1: Content-Adaptive LCD Backlight Scaling

4.1 Background

4.1.1 Liquid-Crystal Display and Backlight Scaling

A liquid-crystal display (LCD) is a flat panel display that uses the light modulating properties of liquid crystals. Liquid crystals do not emit light directly. Instead, the backlight on the LCD panel illuminates the liquid crystals. A liquid crystal's transmittance can vary so that different pixels can have different luminance levels although the intensity of the backlight is the same for all liquid crystals.

Previous studies point out that the backlight of an LCD display dominates the energy consumption of the display panel [18]. During video playback, the display needs to stay in active mode for the whole session; thus a reasonable way to reduce the power consumption is to dim the backlight. However, simply varying the backlight may lead to image distortion, i.e., affect image fidelity, which is normally defined as the resemblance between the original video image and the backlight-scaled image [19, 20].

One way to resolve the problem is by concurrent brightness and contrast scaling [21]. The idea is captured as shown in Figure 4.1. The top part shows the original video image

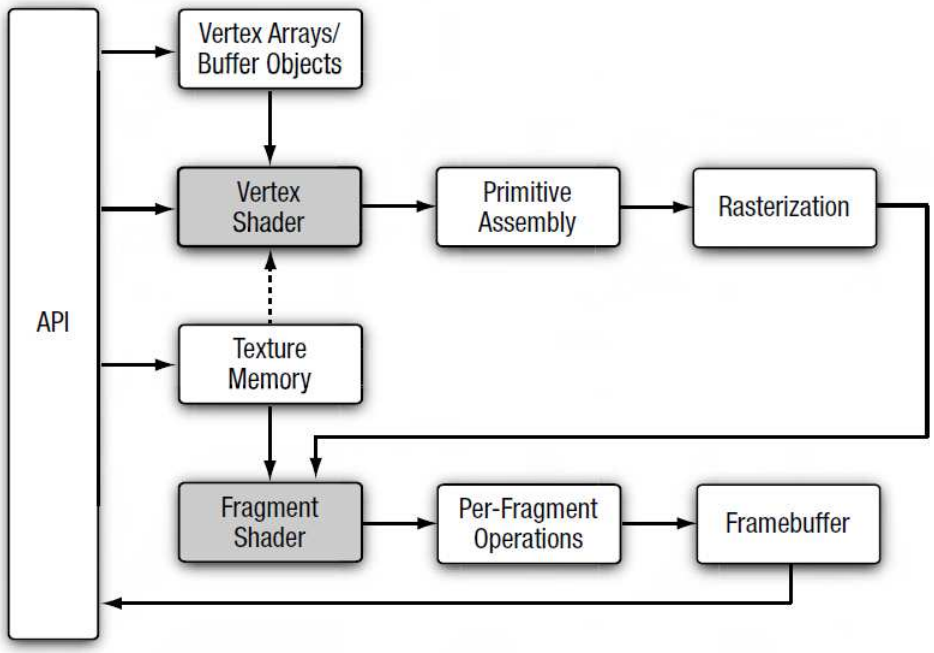

Figure 4.2: OpenGL ES 2.0 Graphics Pipeline

presented with 100% backlight level. In the bottom part, in order to reduce power consumption as well as keep image fidelity, we need dim the backlight and increase the RGB values at the same time so that the contrast of the output image is preserved.

4.1.2 OpenGL ES

OpenGL for Embedded Systems (OpenGL ES) [22] is a subset of the OpenGL 3D graphics API. It is designed for handheld and embedded devices such as mobile phones, PDAs, and video game consoles. Notable platforms supporting OpenGL ES 2.0 include the iPhone 3GS and later, Android 2.2 and later, and WebGL.

Figure 4.2 provides an architectural view of the OpenGL ES 2.0 graphics pipeline. The shaded boxes indicate the programmable shaders of the pipeline. Shaders are written in OpenGL Shading Language (GLSL), whose syntax is similar to C language. Usually the shader source code is stored as strings in an application which utilizes OpenGL ES shaders. Upon initialization, the application invokes OpenGL ES API to compile and link the shaders

into an OpenGL program. The program is then loaded into the GPU for execution. OpenGL ES defines various mechanisms for the application code running on the CPU to exchange data with the program loaded and executed on the GPU. Similar mechanisms are defined for the programmable shaders to pass information as well.

As shown in Figure 4.2, there are two shaders: Vertex Shader and Fragment Shader.

The Vertex Shader is called once for each input vertex. The main task of the Vertex Shader is to provide vertex positions for the following stages of the pipeline. Additionally, it can calculate further attributes that can be used as input for the Fragment Shader later. The most basic shader just takes vertex positions as input and directly assigns the input data to the `gl_Position` varying variable.

Similar to the Vertex shader, the Fragment Shader is called once for each primitive fragment (i.e., pixel). The main task of the Fragment Shader is to provide color values for each output fragment. The most basic Fragment Shader just assigns a constant value to its `gl_FragColor` output. Typically, the Fragment Shader does a texture lookup and implements lighting based on the lighting parameters the Vertex Shader computed previously.

4.1.3 Android OpenGL ES Support

Android includes support for high performance 2D and 3D graphics with the Open Graphics Library (OpenGL), specifically, the OpenGL ES API. The OpenGL ES 1.0 and 1.1 API specifications have been supported since Android 1.0. Beginning with Android 2.2 (API Level 8), the framework supports the OpenGL ES 2.0 API specification.

The usual way on Android for a video player app to render the decoded video frames on the display is either through hardware overlay or direct framebuffer manipulation. Hardware overlay is generally used together with hardware video decoders in which decoded video frames from the decoders are directly rendered onto the display, without GPU involved. For software decoders, the framebuffer, a portion of main memory reserved to store and update the raw image data for the display chip, is updated with the pixel data from each decoded frame. This is without the participation of GPU either.

Alternatively, with OpenGL ES, we can create an OpenGL surface with certain size, load a decoded video frame into an OpenGL texture object, map the texture object onto the surface, and render it onto the display through the OpenGL ES graphics pipeline. The benefit of doing so is that we can apply a lot of sophisticated filters as shaders to achieve various graphic effects, for example, changing viewing angle, color transformation, sharpening, etc.

The Android framework provides the `GLSurfaceView` class which is a special implementation of `SurfaceView` that uses the dedicated surface for displaying OpenGL rendering. On the other hand, the `MediaPlayer` class is able to take a customized `Surface` as the sink of the decoded video frames. A `GLSurfaceView` then needs to be wrapped in a `Surface` object to get the decoded frames.

4.1.4 YUV

Video decoders usually decode video streams into YUV frames. YUV is a color space typically used as part of a color image pipeline. It encodes a color image or video taking human perception into account, allowing reduced bandwidth for chrominance components, thereby typically enabling transmission errors or compression artifacts to be more efficiently masked by the human perception than using a "direct" RGB-representation.

Y stands for the luminance component (the brightness) and U and V are the chrominance (color) components. To calculate a proper backlight dimming level, Y values of a video frame will be used. The YUV values are per pixel.

When the decoded frame is to be rendered on the display, a conversion from YUV color space to RGB color space is needed since display chips use RGB color space. The equations to convert between YUV and RGB color spaces can be found at [23].

4.2 Display Adaptation Design

In this section, we present the design of the Display Adaptation system.

Generally speaking, the system is composed of two parts. The second part depends on the output of the first part.

Backlight Scaling Data Generation. This step takes the decoded YUV frames as input and calculates the maximum Y values as the basis to generate backlight scaling data. The backlight scaling data is a series of float numbers within the range of (0, 1], each with the corresponding frame index at which the backlight scaling data should be applied.

Next we discuss the major design decisions we made for this step:

- *Generate the backlight scaling data offline.* Video processing is data intensive. In the case of analyzing a 30 FPS 720p video stream with the resolution of 1280x720, it requires a CPU running at several hundreds of MHz. Such a high computation overhead will lead to much higher power consumption which adversely offsets the power savings achieved through backlight dimming. Moreover, multiple-iteration global optimization is only possible when we run the the algorithm offline. Such kind of optimization can improve the power savings as well as minimizing the flickering.

- *Perform scene based backlight scaling instead of frame based.* Per frame backlight scaling usually causes inter-frame brightness distortion, or flickering, when the variation between two consecutive backlight scaling levels is above certain threshold. On the other hand, it is impossible to adjust the backlight promptly for every image frame because the underlying hardware takes some time to react and usually there is limitation. This renders per frame backlight scaling impossible. Instead, we choose scene as the basic unit to do backlight scaling. Frames in the same scene tend to have similar luminance histograms, this naturally leads to a design which uses the same backlight intensity level. As a result, we use the maximum Y value of all frames in the same scene to calculate the backlight scaling level. Otherwise, some frames in the scene may show over-saturated images when the pixel luminance is scaled up. Such a design addresses both issues discussed above: 1. There is no flickering between frames in the same scene. 2. The duration of a scene is larger than the reaction time of the backlight so it is safe to do backlight scaling on a per scene basis.

- *Limit the variation between two consecutive backlight scaling points.* Scene based backlight scaling eliminates the flickering between frames in the same scene. Still, there are chances that when there is a scene change, the variation between the backlight levels of the two scenes causes inter-frame brightness distortion. For example, when we change from a dark scene to a bright scene, or vice versa. In order to mitigate such brightness distortion, we define the inter-scene variance constraint ratio r that the relative difference, either backlight increase or decrease, is not greater than the current backlight level multiplied by r.

Backlight and Contrast Scaling. This step is essentially done online. When it is about to render a frame at which, according to the backlight scaling data, needs to have the backlight intensity level changed, we set the backlight intensity to the new value b; at the same time, for each pixel in the frame, we get Y', which is the product of the original Y multiplied by $1/b$, and the original U and V values to do a color space transformation from YUV to RGB. Once it is done, the video image is displayed. Note that for frames at which there is no backlight intensity update, we need scale the pixel luminance using the current backlight intensity level to keep contrast.

In this step, we made the following major design decisions:

- *Use OpenGL ES to scale pixel luminance.* Similar to the offline part, CPU is not designed to handle such a massive data operation although the operation for each pixel only takes several tens of CPU cycles. Even a higher frequency CPU is able to accomplish the task in time, the extra power consumption, in some cases, will be larger than the savings achieved by dimming the backlight. On the other hand, GPU, by nature, is specialized in highly parallel processing. Though the tiny processors in GPU runs at a much lower frequency than CPU, it is able to parallelize the task and finish in time. Nevertheless, there is also power consumption overhead concern on GPU. This will be discussed further in the following section.

- *Scale the backlight intensity to the levels supported by the hardware.* All backlight

devices only support a limited number of discrete intensity levels. The Android API does not expose such information and it simply allows any value in the range (0, 1]. The Android system leaves this to the device driver and/or the hardware to determine the appropriate level. Directly applying the value calculated offline leads to contrast inaccuracy since the pixel luminance is derived from the offline calculated backlight intensity , which is different from the intensity reflected by the backlight of the display. The visual effect would be image distortion to some extent. In order to mitigate the issue and maintain the contrast as accurate as we can, we need round up the intensity level to the one supported by the hardware. This needs to be done online since otherwise we need generate the set of backlight scaling data of a video file for each display.

In order to present the backlight scaling data generation algorithm, we define the following terms. Let $V = \{S_1, S_2, ..., S_n\}$ denote the decoded video stream where $S_i (1 \leqslant i \leqslant n)$ is a scene. S_i is defined as $S_i = \{f_{i1}, f_{i2}, ..., f_{im}\}$ with f denoting a decoded frame in YUV format. The function $MaxY(f_1...)$ takes variable length arguments and is used to get the maximum Y value from the sequence of YUV frames. b is used to denote the calculated backlight scaling value. r is backlight variance constraint ratio. The algorithm is depicted by Algorithm 2.

Set B contains the optimized backlight scaling data. Note that the for loop used to apply the backlight variance constraint may run multiple times. This is because in case the subsequent backlight scaling value is more than $(1 + r)$ times larger than the current value, we need increase the current value and this increase might make the current and the previous values break the constraint.

4.3 Implementation and Evaluation

The backlight scaling data generation is implemented as a standalone application running on desktop machines instead of the target device, as explained in the previous section. We

Algorithm 2 The Backlight Scaling Data Generation Algorithm

```
B = ∅
for each S in V do
    b = MaxY(∀f ∈ S)
    B.append(b)
end for

B[0] = max(B[0], 1 − r)
done = False

while !done do
    done = True
    for i = 1 to n - 1 do
        if abs(B[i] − B[i − 1])/B[i − 1] > r then
            done = False
            if B[i] > B[i − 1] then
                B[i − 1] = B[i]/(1 + r)
            else
                B[i] = B[i − 1] ∗ (1 − r)
            end if
        end if
    end for
end while
```

use x264 [13] to retrieve the scene information, i.e., the sequence of scenes and the indices of frames belonging to a scene. The application takes the scene information and the video file as input. Video decoding is done using the H.264/AVC decoder provided by the ffmpeg library. The generated backlight scaling data is stored into a disk file with each value coupled with the corresponding frame index.

The online part, the concurrent backlight and contrast scaling, is implemented in an Android video player app. As explained in the background, we use the `MediaPlayer` provided by Android to decode the video stream. Instead of rendering the video frames onto the default `Surface`, we create a `GLSurfaceView`, wrap it into a `Surface` object and set the `MediaPlayer` to use this `Surface` as the video data sink so as to divert the video frames into the `GLSurfaceView` object.

In the `GLSurfaceView`, we have a customized `Renderer` that implements the Vertex Shader and the Fragment Shader. When there is a new frame decoded and passed to the `GLSurfaceView` for rendering, we get the current backlight scaling value: if the new frame is the first frame of a scene, the new backlight scaling value is retrieved and rounded up to the next high value supported by the display; otherwise, we directly reuse the last backlight

scaling value.

The Vertex Shader just sets up the vertex positions without any transformation. In the Fragment Shader, since the pixels are described in RGB color space with the conversion from YUV done by the Android system, we first convert it back to YUV color space. Next, we scale the Y value of the pixel to Y' and convert it back to RGB color space using Y', U and V.

At the same time, the code running on the CPU adjusts the backlight intensity value if the new frame is the first frame of a scene. By jointly scaling the backlight intensity and the pixel luminance, power consumption is reduced and the image contrast is maintained.

We choose the Samsung Galaxy Tab2 10-inch tablet as the evaluation device. Specifically, the tablet features the TI OMAP4430 SoC which includes the PowerVR SGX540 GPU and the display supports 16 backlight levels. To build the supported backlight level table, we evenly pick 16 numbers in the range (0, 1].

First, we want to check how much power consumption overhead is introduced by using OpenGL ES, i.e., the GPU. As previously discussed, traditionally, video playback through either hardware decoder or software decoder does not get GPU involved. And per Android design and implementation, whenever GPU is not used, it is put into deep sleep state in order to preserve power consumption. We check this by comparing the power consumption numbers of playing the same video using the stock video player and using our video player app but have the display adaptation turned off. The data shows that when GPU is in use, the power consumption is around 250 mW. We believe GPU accounts for most of this power consumption overhead since the complexity of the extra code running on CPU is negligible. The overhead is relatively small on a tablet since the overall system power consumption of a tablet when playing videos is usually around 4.5 W.

Next, we run automated experiments using 300 720p videos randomly selected from our YouTube video repository. Again, each video is played with both the Android stock video player app and our video player app. To study how the backlight variance constraint ratio r affects the power savings, for each video file, we generate 3 backlight scaling data files

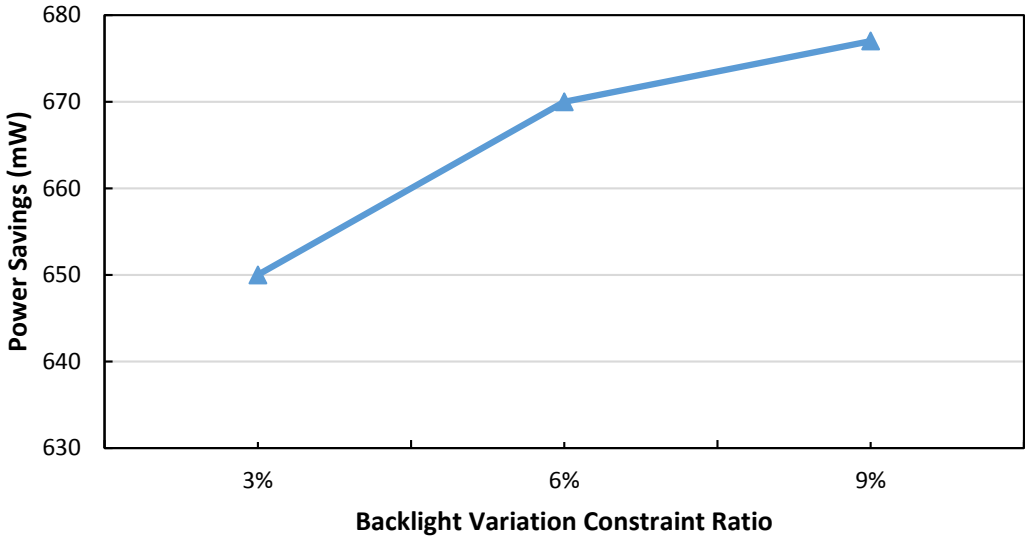

Figure 4.3: Average Power Savings under Different Backlight Variation Constraint Ratios

with a ratio of 3%, 6% and 9%, respectively.

We average the power savings measured from the experiments for each backlight scaling variation constraint, and present the result in Figure 4.3. The average power savings using backlight scaling constraint ratio 3%, 6% and 9% are 650 mW, 670 mW and 677 mW, respectively. Compared to the original power consumption, the system achieved around 15% overall power savings. The difference of using different ratios is negligible. In order to understand this, we manually checked some videos. We found that for videos with interleaved scenes with high difference in luminance, a higher backlight scaling constraint ratio produces much higher power savings. Since most videos do not exhibit such characteristics, i.e., the luminance difference between consecutive scenes tends to be moderate, the average power savings do not differ much.

To check the flickering, we manually watched several videos and found that with a 9% ratio, there are some transient flickerings between scenes with distinct luminance levels. We can hardly feel any flickering with 3% and 6% ratios. Thus it can be concluded that 6% ratio is practical for real system deployment.

To evaluate the overall video quality, we recruited 10 people and asked them to watch

10 videos, with and without our display adaptation algorithm. We use a 6% ratio for the test. 9 out of 10 people said they cannot tell any difference. Interestingly, one person said he feels the one with display adaptation looks better.

Chapter 5: GreenVideo

In this chapter we present GreenVideo, a framework for energy-efficient video streaming to handheld devices. We build the framework by systematically integrating our schemes on power optimization for 3G/4G radio, the application processor, and the display subsystem in the context of mobile video streaming. Experiments show that the framework can achieve substantial power savings for mobile video streaming.

5.1 GreenVideo System

In order to maximize power saving, we build a framework by systematically integrating our techniques of power optimization for 3G/4G radio, the application processor, and the display subsystem. We call this framework GreenVideo.

For this system, we integrate all the components into the Android framework and create a class `GreenVideoPlayer`, which is the only class an app developer has to know and use in order to enjoy the benefit of all mobile video streaming power optimizations proposed in the thesis. Doing so encapsulates all the details of the framework and greatly eases app developers' pain to migrate to the new framework.

In order to make it as transparent as possible, the `GreenVideoPlayer` class is modeled after the `MediaPlayer` provided by the Android framework. The major differences are:

- In `GreenVideoPlayer`, it is not allowed to set a `Surface`. Otherwise, when an app developer accidentally sets a `Surface`, decoded video frames will be routed to the new `Surface` object and the display adaptation component will be disabled.
- `GreenVideoPlayer` only allows HTTP video streaming, i.e., local video playback is turned off since in this thesis, mobile video streaming is our primary focus.

- `GreenVideoPlayer` only supports H.264/AVC video streams encapsulated in MP4 container since we use the x264 H.264/AVC video decoder and the MP4 format exclusively in this thesis.

Based on the Android system architecture, the `GreenVideoPlayer` Java class is a simply wrap up of the real implementation class which is part of the underlying Andriod Multimedia Framework written in C++. In order to integrate all of our techniques, we created a C++ class in the Android Multimedia Framework with the same name `GreenVideoPlayer`. In this player, the following functionalities are implemented:

- The DCM scheme for 3G/4G radio power consumption. This part can be easily integrated since the original work is also done inside the Android Multimedia Framework.
- Codec DVFS. We first integrate the optimized x264 H.264/AVC decoder into the `GreenVideoPlayer` and we complete the functionality by adding the mechanism to download the complexity description file, which can be located on the streaming server using the video file name. We add control logic at the point where a video frame is to be decoded to apply the Codec DVFS algorithm.
- Display Adaptation. In order to divert the decoded YUV frames to be processed by our OpenGL ES shaders so as to apply the display adaptation algorithm, internally, we create a `GLSurfaceView` and attach it to the `GreenVideoPlayer`. When a decoded YUV frame is to be rendered, we apply the logic discussed in Section 4.2 to do simultaneous backlight scaling and contrast scaling. Similar to the Codec DVFS integration, we add logic to download the backlight scaling data file from the streaming server using the video file name.

The sizes of the complexity description file and the backlight scaling data file are both about several KBtyes to several tens of KBytes and the files are downloaded in one shot right before the video playback begins. Taking the small size of these files, the framework is able to download them at the very beginning of the first video file downloading session. Thus the extra power consumption is negligible.

The integration in total required around 700 lines of code.

5.2 Evaluation

In order to evaluate GreenVideo, we created a simple app. The app directly uses the `GreenVideoPlayer` and otherwise shows no difference from a video player app using the original media player.

Similar to the experiments conducted in previous works, we randomly picked 300 720p video files from the YouTube video repository, generated complexity description files and backlight scaling data files with backlight scaling variation constraint ratio 6%. All files are put onto an Internet accessible HTTP video streaming server. The complexity description files and backlight scaling data files are named after the corresponding video files so the app can locate them.

We chose the Galaxy Tab2 10-inch tablet as the target device. The tablet has a 10-inch LCD display panel, uses TI OMAP 4430 SoC whose GPU is PowerVR SGX540 and provides access to T-Mobile HSPA+ wireless network. We did not choose any phone because all the phones we have are with OLED display panel on which the display adaptation can not work.

The baseline power consumption is measured on the tablet with a self-written video player app which uses the x264 H.264/AVC video decoder and streams the videos from the same HTTP video streaming server. The Android stock video player is not used as reference because it uses the hardware decoder which has much lower power consumption than the software decoder used and there is no known way to control the frequency of the hardware decoder.

We run the experiments for both User A and User B since the DCM scheme is user viewing history based. We follow the methodology discussed in 2.4.2. We collect the traces for all experiments and average the power consumption for each user category.

The experimental results are shown in Figure 5.1. The left figure shows the power consumption distribution for users (for example, User B) who tend to finish watching the videos. The figure on the right shows the power consumption distribution for users (for

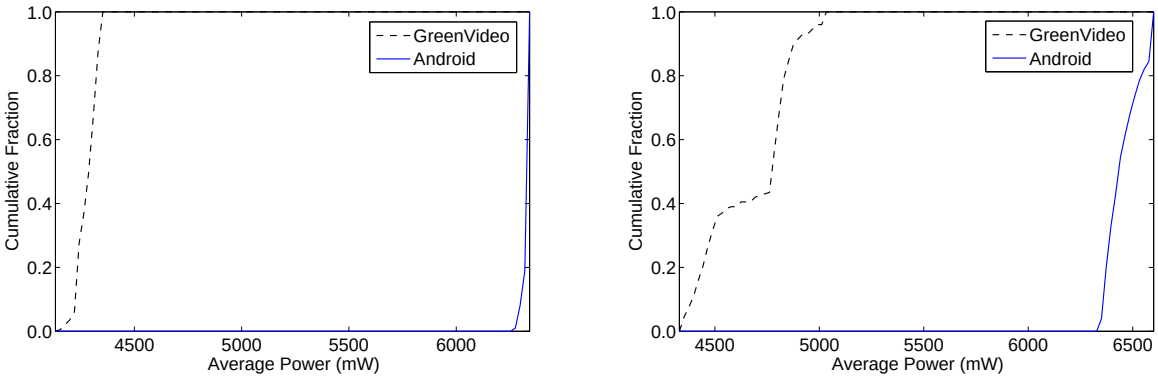

Figure 5.1: Power Consumption Distribution of GreenVideo for Different Users

example, User A) who usually watch only the very beginning portion of the video.

As shown in the figure, our system achieves more power reduction by integrating all of our power optimization techniques discussed before.

For User A, the overall power saving is around 1.6 W, or, 25% for the whole system. The ratio is not a simple addition because the previous experiments are done in isolation with the other power optimization mechanisms so the total power consumption is different even for the same video playback.

For users who behave like User B, the power saving is larger mainly due to the contribution of better power savings from the wireless radio. Other factors including the duration of the Codec DVFS and the display adaptation algorithm also contribute to the difference of power savings. By comparing with the previous experimental results for the DCM scheme on T-Mobile HSPA+ wireless network, we observe that the absolute power consumption differences for the two types of users in the two sets of experiments are about the same. Thus, we believe that they are not the main factors. The overall power saving in this scenario is about 2 W, or, 33% for the whole system.

To summarize, GreenVideo is able to achieve about 25% to 33% power reduction for the whole system on the Galaxy Tab2 10-inch tablet with T-Mobile HSPA+ wireless network. In other words, GreenVideo can prolong the battery lifetime for up to 1.5 times. We conclude that the power savings are in accordance with the power savings from each individual work

and GreenVideo substantially reduces power consumption for mobile video streaming with nearly no quality degradation.

Chapter 6: Conclusion and Future Work

In this chapter we conclude the thesis and discuss the future work.

6.1 Conclusion

Mobile video streaming has become one of the most popular applications in the trend of smartphone booming and the prevalence of 3G/4G networks. However, the power consumed by wireless transceiver, the application processor, and the display system severely hinders the user experience of mobile video streaming.

In this thesis, in the context of mobile video streaming, we present our techniques on power optimization of three key power consumption sources, namely, GreenTube on power optimization for 3G/4G radio, Codec DVFS on power optimization of decoding running on application processor, and Content-Adaptive display power optimization.

By systematically integrating the individual work, we build GreenVideo, a framework for energy-efficient video streaming to handheld devices. Experiments show that GreenVideo achieves substantial power savings for mobile video streaming. The measured power savings are among 25% to 33% for the whole system, which can prolong the battery lifetime for up to 1.5 times.

6.2 Future Work

6.2.1 GreenTube

As discussed in previous chapters, the fundamental reason why 3G/4G radio consumes so much power is the presence of relatively long tail state. Therefore, an alternative and effective solution to reduce 3G/4G radio power is to make the smartphone notify the base

station and initialize the power state demotion after completion of each downloading session. By doing this, the smartphone can immediately set the interface into IDLE state instead of waiting for the expiration of the tail timer. As aforementioned, such a tail state consumes a large portion of the power as well. By eliminating the tail-state power, the 3G/4G radio can be very power efficient. This type of feedback channel enabling between mobile client and base station is worthy of further study and standardization along with the process of 3GPP LTE and LTE-Advanced protocols.

6.2.2 Display Adaptation

The current solution requires the backlight scaling data generated offline and stored on a server. This inevitably requires update to both the streaming server and the video player app. One way to fix the issue is to add the backlight scaling data as metadata into the MP4 container. Still, it requies either the content provider or the streaming service vendor to generate and embed the metadata.

We envision that a transparent online solution which not only uses OpenGL ES to scale the pixel luminance, but also relies on OpenGL ES to generate the backlight scaling data on the fly is a better solution. There are technical obstacles which prevent us from achieving the goal at this moment. We plan to continue this work and seek for such a solution.

OLED display panels have very different power consumption characteristics. In short, the LED emits light by itself, without the need of a backlight in LCD. The power consumption of each LED can be individually tuned and it largely depends on the color. Some pioneer work has been done on designing and implementing hardware control circuit to optimize OLED power consumption [15]. We plan to continue our research and apply similar content-adaptive display power optimization algorithm to OLED displays.

Bibliography

Bibliography

[1] "Allot MobileTrends Report," http://www.allot.com/MobileTrends_Report_H2_2011.html.

[2] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, "Livelab: measuring wireless networks and smartphone users in the field," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 3, pp. 15–20, Jan 2011.

[3] H. Yin, X. Liu, T. Zhan, V. Sekar, F. Qiu, C. Lin, H. Zhang, and B. Li, "Design and deployment of a hybrid CDN-P2P system for live video streaming: experiences with livesky," in *Proc. of the 17th ACM international conference on Multimedia*, 2009, pp. 25–34.

[4] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A close examination of performance and power characteristics of 4G LTE networks," ser. ACM MobiSys '12, 2012.

[5] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, "Profiling resource usage for mobile applications: A cross-layer approach," in *Proc. of the 9th international conference on Mobile systems, applications, and services*, 2011, pp. 321–334.

[6] "Monsoon Power Monitor," http://www.msoon.com/LabEquipment/PowerMonitor/.

[7] Z. He, Y. Liang, L. Chen, I. Ahmad, and D. Wu, "Power-rate-distortion analysis for wireless video communication under energy constraints," *IEEE Trans. Circuits and Systems for Video Technology*, vol. 15, no. 5, pp. 645– 658, May 2005.

[8] M. Horowitz, A. Joch, F. Kossentini, and A. Hallapuro, "H.264/AVC Baseline Profile Decoder Complexity Analysis," *IEEE Trans. Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 704–716, July 2003.

[9] V. Pallipadi, "Enhanced intel speedstep technology and demand-based switching on linux," *Intel Software Net.*

[10] Z. Ma, H. Hu, and Y. Wang, "On complexity modeling of h.264/avc video decoding and its application for energy efficient decoding," *Multimedia, IEEE Transactions on*, vol. 13, no. 6, pp. 1240–1255, 2011.

[11] "TI OMAP 4460," http://www.ti.com/product/omap4460.

[12] "ARM Cortex-A9," http://www.arm.com/products/processors/cortex-a/cortex-a9.php.

[13] "x264," http://www.videolan.org/developers/x264.html.

[14] "Cortex-A9 Technical Reference Manual," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388f/DDI0388F_cortex_a9_r2p2_trm.pdf.

[15] D. Shin, Y. Kim, N. Chang, and M. Pedram, "Dynamic voltage scaling of oled displays," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: ACM, 2011, pp. 53–58. [Online]. Available: http://doi.acm.org/10.1145/2024724.2024737

[16] I. Choi, H. Shim, and N. Chang, "Low-power color tft lcd display for hand-held embedded systems," in *Proceedings of the 2002 international symposium on Low power electronics and design*, ser. ISLPED '02. New York, NY, USA: ACM, 2002, pp. 112–117. [Online]. Available: http://doi.acm.org/10.1145/566408.566440

[17] F. Gatti, A. Acquaviva, L. Benini, and B. Ricco', "Low power control techniques for tft lcd displays," in *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, ser. CASES '02. New York, NY, USA: ACM, 2002, pp. 218–224. [Online]. Available: http://doi.acm.org/10.1145/581630.581664

[18] T. Simunic, L. Benini, P. Glynn, and G. De Micheli, "Event-driven Power Management," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 20, no. 7, pp. 840–857, 2001.

[19] L. Cheng, S. Mohapatra, M. El Zarki, N. Dutt, and N. Venkatasubramanian, "Quality-based backlight optimization for video playback on handheld devices," *Adv. MultiMedia*, vol. 2007, no. 1, pp. 4–4, Jan. 2007. [Online]. Available: http://dx.doi.org/10.1155/2007/83715

[20] P.-S. Tsai, C.-K. Liang, T.-H. Huang, and H. Chen, "Image enhancement for backlight-scaled tft-lcd displays," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 19, no. 4, pp. 574–583, 2009.

[21] W.-C. Cheng and M. Pedram, "Power minimization in a backlit tft-lcd display by concurrent brightness and contrast scaling," *Consumer Electronics, IEEE Transactions on*, vol. 50, no. 1, pp. 25–32, 2004.

[22] "OpenGL ES," http://www.khronos.org/opengles/.

[23] "Color Conversion," http://www.equasys.de/colorconversion.html.

Curriculum Vitae

Xin Li grew up in Hunan, China. He attended Huzahong University of Science and Technology from 1999, where he received his Bachelor of Science in Electronics and Information Engineering in 2003 and Master of Science in Telecommunication and Information System in 2005, respectively. He then joined Intel Asia-Pacific Research and Development Ltd. and worked on the next generation BIOS for 3 years. In 2008, he went to George Mason University for his graduate study. He will receive his Master of Science in Information Security and Assurance in Spring 2013.