ABSTRACT

GOKARE, KUMAR S. Cache Aided Video Coding. (Under the direction of Injong Rhee.)

Studies [1] have indicated significant increase in video traffic volumes, scaling in the order of exabytes in next couple of years. Video publishers face massive bandwidth and storage costs incurred from such magnitudes of traffic which necessitates studying effective ways to reduce traffic costs. Typically, video publishers rent a network of caches in the Internet called Content Delivery Network (CDN) which reduces the bandwidth costs but results in additional CDN renting costs. Another approach is to improve efficiency of video compression schemes. However current compression schemes concentrate mainly on complex image processing techniques and do not exploit the presence of caches available in the network. In this thesis, we improve Variable Length Coding, a popular technique utilized in video compression with the aid of a caching strategy to gain additional cost reductions. We show that the implementation of this new paradigm of ‘Cache Aided Video Coding’ can further reduce traffic costs by about 5% on a global scale for all the videos streamed over Internet, measured over time. With current and forecasted volumes of video traffic, we estimate that with 5% traffic savings, large video publishers such as Youtube could save in the order of millions of dollars every year in bandwidth and CDN pricing costs employing our scheme.
Cache Aided Video Coding

by
Kumar S. Gokare

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science
Raleigh, North Carolina
2011

APPROVED BY:

_________________________  _________________________
Rudra Dutta                  Khaled Harfoush

_________________________
Injong Rhee
Chair of Advisory Committee
DEDICATION

To my parents and my brother.
BIOGRAPHY

Kumar Gokare was born in Bangalore, India. He received his Bachelor’s degree in Computer Science from Rashtreeya Vidyalaya College of Engineering (R.V.C.E), Bangalore, India. After completing his undergraduate degree, he worked as a Software Engineer in Cisco Systems, India on technologies such as VoIP and IP telephony. He joined North Carolina State University (NCSU), Raleigh, USA in Fall 2008 to pursue his Master’s degree in Computer Science. His areas of interests include Multimedia networking, Video Compression, Internet protocols and Peer to Peer systems. He also has internship experiences in Hewlett Packard and IBM. He plans to graduate in December, 2010 upon completion of his Master’s degree.
ACKNOWLEDGEMENTS

I would like to first extend my gratitude to my advisor Dr. Injong Rhee, for his support, advice and guidance throughout this work and in my studies. I would like to thank Dr. Rudra Dutta and Dr. Khaled Harfoush for taking time out of their busy schedules to serve as committee members for my Thesis defense.

I would like to thank the members of Network Research Lab for all the help, advice and the fun moments.

I am deeply indebted to my family, my father, mother and my brother for constantly being my support structure and for being there for me through good and tough times.
# TABLE OF CONTENTS

List of Tables ................................................................................... vii

List of Figures .................................................................................. viii

Chapter 1 Introduction ......................................................................... 1
  1.1 Overview .................................................................................. 1
  1.2 Motivation ............................................................................... 2
  1.3 Research overview ................................................................... 4
  1.4 Outline of Contributions ......................................................... 5

Chapter 2 Background .......................................................................... 7
  2.1 Variable Length Coding .......................................................... 7
    2.1.1 Static Variable length coding .............................................. 9
    2.1.2 Dynamic Variable Length Coding ..................................... 9
  2.2 Video coding process ............................................................... 10
  2.3 Context Adaptive Variable Length Coding ................................. 11
  2.4 Related Work .......................................................................... 13

Chapter 3 Dynamic Variable Length Coding ........................................... 15
  3.1 Cache Aided Variable Length Coding ........................................ 15
    3.1.1 Codebook granularity ....................................................... 17
    3.1.2 Codebook redundancy ..................................................... 18
  3.2 Cache Aided Video Coding ....................................................... 18
  3.3 Dynamic VLC Coding Algorithm ............................................ 20

Chapter 4 Cache Aided Video Coding .................................................. 22
  4.1 Cache Aided Video Encoding .................................................... 22
  4.2 Cache Aided Video Decoding ................................................... 22
  4.3 Codebook Management .......................................................... 24
    4.3.1 Codebook operations at the CDN Cache .............................. 26
    4.3.2 Cache operations at client cache ....................................... 27
    4.3.3 Codebook search ............................................................ 27

Chapter 5 Implementation and Evaluation ............................................ 29
  5.1 Implementation ........................................................................ 29
  5.2 Evaluation ................................................................................ 29
    5.2.1 Bit Savings ...................................................................... 30
  5.3 Codebook Redundancy ............................................................. 33
  5.4 Effect of Codebook re-use on savings ....................................... 35
  5.5 Codebook repository Size ......................................................... 35
  5.6 Codebook size Extrapolation .................................................... 35
  5.7 Net cumulative savings ............................................................. 39
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Youtube video statistics</td>
<td>5</td>
</tr>
<tr>
<td>5.1</td>
<td>Libraries used in the Implementation</td>
<td>29</td>
</tr>
<tr>
<td>5.2</td>
<td>Youtube data set statistics</td>
<td>32</td>
</tr>
<tr>
<td>5.3</td>
<td>Storage share for videos and codebooks with fixed cache storage</td>
<td>37</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1.1 Costs of Video Streaming ............................................. 3
Figure 2.1 Different components of VLC compression/decompression .......... 8
Figure 2.2 Video coding process ................................................ 11
Figure 2.3 CAVLC encoding sequence ........................................ 12
Figure 3.1 Cache Aided Variable Length Coding ................................. 16
Figure 3.2 DVLC in Stored video streaming .................................... 19
Figure 5.1 % Bit savings of DVLC over CAVLC ................................. 31
Figure 5.2 Total bytes Count for DVLC and CAVLC .......................... 31
Figure 5.3 Aggregate % Bit savings of DVLC over CAVLC (Youtube videos) . . 32
Figure 5.4 CDF for % Bit savings per file of DVLC over CAVLC (Youtube videos) . . 33
Figure 5.5 Codebooks redundancy and effects on compression ................ 34
Figure 5.6 Codebook repository size ......................................... 36
Figure 5.7 Power curve fitted for % fractional codebook repository size ....... 38
Figure 5.8 Residual values for curve fitting, $R^2$ value of 0.9894 shows a ’good fit’ for the curve ......................................................... 38
Figure 5.9 % Cumulative Bit savings of DVLC over CAVLC (Cache: Top 10% popular videos) ....................................................... 39
Figure 5.10 % Aggregate Cumulative Bit savings of DVLC over CAVLC for 10,000 views ............................................................. 40
Figure 5.11 Bytes delivered from cache and server for cache size: 500, 1000, 1500, 2000, 2500 videos .............................................. 41
Figure 5.12 % Bit savings of DVLC over CABAC ............................... 42
Chapter 1

Introduction

1.1 Overview

Internet video content has seen a significant surge in volume and popularity in the recent years with applications such as Internet video streaming, Video conferencing, video blogs becoming increasingly popular. Recent years has also seen a paradigm shift in the generation of video content from limited number of enterprise video publishers to a large no. of users publishing self made videos and sharing it with millions of other users forming a so called Social Network. This User Generated Content(UGC) has been popularized through video blogs(vlogs), Facebook and video sharing sites such as Youtube [2]. Trends also show that the volume of Internet video traffic will see a significant increase in the future. Video traffic, today accounts for over 30% of the current web traffic [1]. The overall web traffic itself is forecasted to quadruple in the next 4 years and online video traffic would account for 91% of overall traffic by the year 2014 [1]. Youtube itself accounts for 60% of all the videos watched over the Internet [3]. Such massive increase in volume of the video content has resulted in increasing demands on the resources such as storage and network bandwidth. Internet Video providers are faced with a challenge to cope up with scaling network infrastructure and managing incurring costs. According to Youtube, users upload about 65,000 videos everyday, about 24 hrs worth of video is uploaded every minute [3]. Such trends indicate significance of studying ways to efficiently manage network resources and scale up to the demands. Major players in the area have employed several techniques to sustain the scaling costs.

Vertical scaling refers to improvement of algorithms and functions within the system to meet the demands. For video distribution, installing advanced video codecs would result in better compression and an overall reduction in video traffic and the incurring costs. H.264 [44] for example, is an advanced video compression scheme developed by a joint team of ISO, IEC and ITU-T called Joint Video Team(JVT). H.264 was a massive improvement over its predecessors
achieving over 50% bit savings [44]. On2 technologies developed a similar codec drawing most of the characteristics from H.264 which was later acquired by Google and renamed as VP8 [4] with similar bit rates. In spite of these advancements, traffic and its costs remain high due to large number of users, who consume massive proportions of video content everyday.

To alleviate this problem, video publishers also scale horizontally employing caches closer to user and cache popular video content. Companies like Akamai [5], Limelight [6] have built a business model based on this idea and have deployed cache nodes at strategic geographic locations and rent the caching nodes to distribute video content on behalf of video publishers. They have built a so called Content Delivery Network(CDN) [7] of caching nodes in the Internet which distribute popular content. The video publishers cache relatively smaller volume of popular content and only service unpopular content directly. This technique reduces the bandwidth transit costs for the video publishers with increasing cache hits. However several major CDN players in the market have come up with Delivery price model along with Storage price model. This means that the CDNs charge based on the popular content delivered to users rather than storage sustained [8]. This pricing model again impedes the reductions in overall traffic costs for the video publishers.

A mechanism to combine both vertical and horizontal scaling is required to achieve additional reductions in global traffic costs for video content and scale up to massively increasing end user demands. In this thesis research we explore a strategy to improve upon popular video codecs with the aid of existing cache infrastructure to achieve reductions in overall video traffic at a global scale measured over a period of time. We implement a working prototype of this concept and evaluate the performance of such a scheme on real world data.

1.2 Motivation

To motivate the problem, we did a case study on the data collected from the Youtube. The aim is to highlight the scaling costs of video traffic distribution and how the current mechanism can be improved to achieve further cost savings. Figure 1.1 presents a picture of a typical internet video streaming infrastructure. A media sharing site such as Youtube deploys a collection of origin servers where the video content is stored and makes it available for viewing through a website. Caches are installed in the Internet at the edge of networks typically through a CDN. Previous studies [18] have shown that the video views exhibit long tail effect. This implies the majority, as much as 80% of the views are accounted by only a few, about 20% of distinct popular videos. This enables Youtube to cache a small amount of videos in the caching nodes and satisfy a large number of users. Youtube origin servers would only serve the unpopular content. There are two main cost factors in this type of setup that has to be considered. 1)
Server side level-3 bandwidth transit costs incurred by transporting popular content regularly to CDN caches and servicing unpopular content directly to users. 2) Client side CDN delivery costs due to pricing models of CDN which charge $/data delivered. Table 1.1 tabulates the information collected from Youtube for 400,000 videos across 3 categories recording average video durations and view counts. Youtube encodes the videos with variety of bit rates ranging from 0.25 Mb/s to 5 Mb/s. In our studies we assume average video bit rate to be 0.5Mb/s. As per Youtube, it receives 2 billion video views per day and 100 million unique downloads per day. Considering 20% of videos are popular(account for 80% of views), we can calculate the net costs of video streaming for Youtube.

For 100 million videos, we divide the share of videos among 3 categories as per the fractional views calculated in Table 1.1. Considering 0.5Mb/s(or 2.5MB/min) across board, we get average file size for Entertainment videos = 13.8MB, Music videos = 10.66MB and Comedy videos
= 11.95MB. From the calculations below, we can approximate that Youtube needs to spend approximately \$3.9 million per month\ to manage the traffic costs.

1. Video views = 2 billion views per day.

2. Unique downloads = 100 million videos per day.

3. Sever side costs = Transit costs for popular content from server to CDN + Transit costs for streaming unpopular content.
   Approximate transit costs = \$10/Mbps per month\ [9].
   Popular content = 20% of videos = 20,000,000 videos.
   Transit cost to CDN per month = \$10/Mbps \times (13.8MB \times 4400000 + 10.66MB \times 15400000 + 11.95MB \times 2000000) = \$10Mbps \times 92142 Mbps = \$921,422 per month.
   Transit cost to serve unpopular content = Cost of remaining 20% views among 80% unpopular videos = \$462,962 per month.

4. Client side costs = CDN distribution costs for popular content.
   Approximate CDN delivery pricing = \$0.01/GB per month\ [8].
   Popular video views = 800,000,000.
   Delivery cost = \$0.01/GB \times (13.8MB \times 176000000 + 10.66MB \times 154000000 + 11.95 \times 20000000)MB = \$0.05 \times 29154375 = \$2,525,062 per month.

Managing such scaling costs increasingly becomes difficult with caching alone. Horizontal scaling mechanisms such as caching are bound by hit ratio values. In case of video distribution, cost savings obtained by caching is bound by \textit{Cost} = \textit{Costs of (hitratio) \times popular content} + \textit{Costs of unpopular content}. Youtube has also employed advanced video compression schemes to reduce individual file sizes. However, video codecs are designed on a stand alone basis without making any assumptions about network infrastructure. In our thesis research, we design algorithmic improvements on a concept named Variable Length Coding employed in several popular codecs and build a caching strategy allowing video encoder to exploit the caches available in the network. We show that by employing such a scheme, we can gain additional cost savings beyond the limit of caching schemes. Our method can also be applied to any type of media: text, audio, image that are compressed using current implementations of Variable Length Coding.

1.3 Research overview

Advanced Video Codecs such as H.264 is a pipelined process of converting Video information to binary digits for storage/transport and vice versa. See Section 2.2 for a more detailed
Table 1.1: Youtube video statistics

<table>
<thead>
<tr>
<th>Category</th>
<th>No. of videos</th>
<th>Avg Duration(secs)</th>
<th>Avg.Views</th>
<th>% of views</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entertainment</td>
<td>200,079</td>
<td>332</td>
<td>88,994</td>
<td>22%</td>
</tr>
<tr>
<td>Music</td>
<td>102,221</td>
<td>256</td>
<td>114,460</td>
<td>77%</td>
</tr>
<tr>
<td>Comedy</td>
<td>161,320</td>
<td>287</td>
<td>30,977</td>
<td>1%</td>
</tr>
</tbody>
</table>

explanation of a generic video coding process. In the last stages of such a process, codecs employ a popular method to map video syntax elements to their entropy [10] called Variable Length coding (see Section 2.1). Variable Length Coding employs a set of codebooks to map symbols to their bit codewords during video compression. In many popular current implementations, a generic static codebook is used for all the video files compressed. We propose a scheme in which dynamic custom codebooks are generated for every file. We show that with such a mechanism, higher compression is achieved than the current static implementations. However it also leads to increase in the overhead traffic of codebooks. So we propose that these codebooks be cached at the existing cache infrastructure such as CDN and exploit the internal redundancy that exist among codebooks to reduce the overhead. We show that over a period of time, the overhead codebook traffic size would become negligible.

1.4 Outline of Contributions

We propose this research as a combination of algorithmic improvements to current implementations and a caching strategy that could be employed in the existing infrastructure of video distribution with limited or no additional deployment efforts. We term this scheme as Cache Aided Video Coding. Our contributions can be outlined as follows:

1. We propose and implement dynamic variable length coding for current video coding standard - H.264, which involves generation of dynamic codebooks during the encoding process to achieve higher compression.

2. We outline a strategy of caching the overhead data of codebooks and achieve reduction in overhead data by exploiting the existing redundancies among different codebooks. We show that over a period of time, the overhead codebook traffic size would become negligible.
3. We implement a video codec plugin which has all the components to be deployed for a video streaming application.

4. We evaluate the performance of our scheme on real world data and in some cases simulating the real world use case patterns and show the effectiveness of our scheme in reducing the overall traffic costs.

The Thesis is organized into following structure. Chapter 2 presents some of the important background concepts required to understand the research and also review some of the related research work done in this area. Chapter 3 describes in detail, the design and workings of our scheme: Cache Aided Video Coding. Chapter 4 discusses the implementation algorithms. In Chapter 5, we evaluate our approach by conducting a wide variety of experiments and discuss the results and its effects on the design of our scheme. In Chapter 6, we conclude the findings of our research and present some future research possibilities.
Chapter 2

Background

2.1 Variable Length Coding

Variable Length Coding (VLC) is a method that employs mapping of a set of symbols to their respective bit codewords of variable lengths (but relatively shorter). In practice, they are employed to achieve compression for different types of source files involving variety of symbols. VLC is considered a "lossless" compression as it does not result in loss of any information in the source files even with repeated runs of VLC compression/decompression(s). VLC schemes can vary in their strategies of mapping symbols to their bit codes. Some of them include: 1) Huffman coding 2) Lempel-Ziv coding 3) Arithmetic coding. Huffman coding assigns codewords based on the weights calculated for each symbol. Lempel-Ziv coding [49] finds longest common subsequence of symbols found in the source file and assigns codewords to subsequence rather than individual symbols. Arithmetic coding [46] encodes all the symbols into a single codeword by dividing the codeword into intervals for each symbol proportional to its probability or frequency. In all of our future references in this document, whenever we refer to Variable Length Coding, we refer to Huffman coding unless explicitly specified otherwise.

Huffman coding [25] is one of the most popular VLC methods employed in variety of applications. The main concept of the algorithm is to assign bit codes to symbols based on their weights. The weights could also represent a symbols’s frequency or probability of occurrence. Huffman’s algorithm builds a binary tree in which all the symbols are the leaf nodes and the path from root to a leaf provides the codeword for that symbol. Intuitively, more probable symbols are closer to the root and least probable symbols are further down in the tree. Simplest of the implementation uses a priority queue and run in $O(n \log n)$ complexity. It can be outlined as follows:
Step 1: Create leaf nodes for each symbol and add it to priority queue based on weights (lowest weight → highest priority).
Step 2: If priority queue contains more than 1 node goto Step 3; else goto Step 7.
Step 3: Remove two highest priority nodes $m, n$.
Step 4: Create a new node $p$ with weight equal to sum of weights of nodes $m$ and $n$.
Step 5: Add $p$ to priority queue
Step 6: Goto Step 2
Step 7: Tree is complete; return root node.

Once the tree is complete, we can extract the bit codewords for each symbol. The mapping of each symbol to its respective codeword becomes the entries in the mapping table called VLC "Codebook". In Huffman coding, following method is used to extract codewords.

Step 1: Start at root, for each symbol $s$ find path $p$ from root to $s$.
Step 2: At each hop on path $p$, if a left branch was taken, append a "0" else append a "1" to codeword $c$.
Step 3: Add mapping of $s$ to $c$ in the codebook.

In case of a network transfer of a source file compressed using VLC, Figure 2.1 shows the different components and costs involved in transferring and decompressing the source file at the client. Consider a simple scenario of a client requesting a file present in the server. VLC encoder present in the server generates a codebook and compresses the given source file. The decoder must possess the codebook to decompress the file. Server sends both the compressed file and the codebook associated with it to the client. The client uses the codebook to decompress the source file to obtain the information. Consequently, codebook information is considered an overhead data. Based on this concept different strategies can be adopted to perform the variable length coding.
2.1.1 Static Variable length coding

Static VLC involves encoding all the input source files with a single generic codebook which contains mappings for all probable symbols to respective codewords. This codebook is hard coded in both encoder and decoder and so is not part of the information that needs to be transported across to decoder. Generally, such a generic codebook trades off optimality to lowered complexity by making approximations on the weight distribution for all symbols. For example; If a text file containing all the English alphabets were to be encoded, we could assign codewords to symbols based on their statistical distribution in common English texts such as books, news articles, literature etc; Statistics show that letters e, a occur more commonly in typical English text than x or q. So we could assign shorter codes for e, a as opposed to x, q. This way a generic codebook can be built which can be used for all input files. Due to these factors, Static VLC remains more popular in implementations of VLC. However, these assumed weights could hurt compression for varied inputs. For example: a text containing a large number of x’s and q’s would not achieve optimal compression. Here is the outline of the observations on static VLC:

- Static VLC is easier to implement and is popularly adopted.
- There is no codebook overhead involved.
- Leads to sub-optimal compression due to assumed weights.
- Net traffic cost in static VLC is equal to cost of transferring the file, $C_{static}$

\[ C_S = C_{static} \]  

(2.1)

2.1.2 Dynamic Variable Length Coding

In Dynamic variable length coding, a first pass on the source file records the weights of the symbols present in the given input file and generates the mapping specific to that file. The second pass utilizes this 'custom'/'dynamic' codebook to compress the input file. Building a specific codebook for each file leads to better compression due to 1) Codewords are calculated for each symbol based on actual weights recorded specifically for the input file. 2) Codes for non-existent or unused symbols are not assigned which leads to more compact codebooks and overall shorter codes. Codebooks can be generated at different granularity such as 1 codebook per file, 1 codebook per paragraph of text and so on. Finer granularity leads to better compression. However with custom codebooks, the overhead is also increased. So it is important to achieve higher compression while minimizing the codebook overhead. Following is the outline of observations on Dynamic VLC.
• Achieves higher compression than Static VLC.

• It involves overhead of codebook transportation.

• Net traffic cost in dynamic VLC is equal to sum of costs of transferring the file $C_{\text{dynamic}}$ and codebooks $C_{\text{codebooks}}$

\[
C_D = C_{\text{dynamic}} + C_{\text{codebooks}}
\]  

(2.2)

2.2 Video coding process

Video coding is a pipelined process of converting video information into binary digits as output which can be stored as a file and/or sent over the network. Figure 2.2 represents a generic video coding process. The decoding works exactly the same way, but in the opposite direction. Video is input as a 3D array of pixels, 2 of which represent spatial information and 1 represents the temporal information. The Video file is divided into multiple encoding units called frames. Frames represent still images which are played consecutively fast enough to deceive the human eye to perceive it as motion video. A standard good quality video plays about 30 frames per second. Frames are further divided into smaller, typically 8x8 sub-partitions called Macroblocks. Each block undergoes a block level coding to proceed to the next stage. The blocks undergo following treatment in this stage. Advanced codecs such as MPEG2, H.264 etc; make use of redundancies that exists within a frame and across frames. They employ a technique called Motion Prediction to exploit these redundancies. Previously encoded blocks are buffered and used during encoding of current block. When the current block is to be encoded, a prediction for current block is calculated based on the data from previously coded block in the current frame (intra-prediction) or blocks from neighbouring frames (inter-prediction). The prediction is compared against the current Macroblock and a difference between the two is calculated. The information stored/sent is vastly reduced by calculating the 'relative difference' rather than encoding the whole frame. This difference is called the 'Residual' component. Residual samples are then transformed using a 4x4 or 8x8 Discrete Cosine Transforms(DCT) which converts them from Time domain to frequency domain. The resultant values are called co-efficient(s) which represent weights for a set of 'basis pattern' of pixels. When the weights are combined with basis patterns, original block can be recalculated at the decoder. The co-efficient(s) are then subjected to Quantization. In this stage, each co-efficient(s) is divided by an integer to produce a truncated value matrix. Due to quantization, the co-efficient(s) gain a unique property. If taken in the form of a matrix of values, the top left part of the matrix which represent the low frequency domain will have non zero co-efficient(s) whereas most part of the bottom right side of the matrix corresponding to high frequency domain will be filled with zeroes. This completes the block level coding. The co-efficient(s) are then re-ordered in...
each matrix in a zig-zag fashion from top left until the last non-zero co-efficient is found, rest of matrix will be filled with zeroes and is not coded. The re-ordered co-efficient(s) are also called levels and are coded with variable length coding or arithmetic coding. Variable length coding achieves further compression by encoding these levels based on their frequencies. The output of this stage is the final bit stream which can be stored as a file or streamed over the network. Thus variable length coding is employed in the last stage of encoding and is the first stage in the decoding process.

2.3 Context Adaptive Variable Length Coding

As pointed in the Figure 2.2, a video coding standard such as H.264 employs a Context adaptive VLC(CAVLC) [22]. This section provides the basic algorithm of CAVLC. We use CAVLC as a benchmark to evaluate our scheme. CAVLC encodes the residual, re-ordered co-efficient(s) in 4x4, 8x8 sequences. CAVLC makes use of several characteristics of syntax elements to achieve compression. It uses a set of static codebooks to replace syntax elements with variable length codewords. It can be summarized in following steps.

1. After transformation and quantization, most of the co-efficient(s) are 0’s. Also in most cases, most non-zero co-efficient(s) are +/-1’s, CAVLC encodes this information as Trail-ing ones or $T1$’s.

   - Encode the number of non zero co-efficient(s) and trailing ones T1’s with a single element $num\_trail$. A choice among 4 codebooks - NUM0-NUM3 is made.

   - Encode the trailing 1’s in the reverse order in the re-ordered array. Up to 3 trailing 1’s are encoded only by their signs, rest are treated as normal non-zero co-efficient(s).

2. Non-zero co-efficient(s) also called $levels$ are coded with a set of 7 codebooks VLC0-VLC6
Figure 2.3 illustrates the sequence of bit stream generated for each block and the codebooks used for lookup for respective syntax elements. Even though, CAVLC adapts by switching between codebooks, it suffers from 2 drawbacks, 1) Codewords are allocated to several levels which are not present in the current block. 2) The frequencies of the levels are assumed to be static and inversely proportional to their magnitude i.e; smaller magnitude levels occur more frequently than higher ones. Correspondingly shorter codes are assigned to higher frequency symbols. This results in reduced compression in cases of videos with varying symbol statistics.

Figure 2.3: CAVLC encoding sequence

with VLC-0 biased to lower magnitude levels and VLC-6 towards high magnitude ones. Codebooks are selected based on pre-defined magnitude thresholds [38].

- Initialize the codebook to VLC0, encode the first non-zero co-efficient.
- If the magnitude of encoded level crosses the threshold, move to the next codebook and encode successive levels.

3. Encode the total no. of zeroes before the last non-zero co-efficient in the re-ordered array through TotalZeros element. This is to further run length code the zeroes in between non-zero co-efficient(s).

4. Encode runs of zeroes, preceding each non-zero co-efficient in the reverse order with run-before.
2.4 Related Work

In most practical implementations of Video delivery, the server load is offloaded to caches deployed through CDN(s) such as Akamai or Limelight. They use variety of caching strategies to provide best hit ratios for the users based on video view statistics, popularity, geographical locations etc; With inferences from [41] and [18], a popularity mechanism can be applied to make decisions on which videos to cache to gain significant server offloads. [41] provides a optimized framework to manage caches for scalable video streaming. However with increased video sizes such as Youtube partner videos which have a limit of 20GB, caching whole video files would prove counter productive. Caching granularity have also been studied in research such as [47] which defines Segment caching and provides mechanism to determine admissions of specific segments and cache replacements strategies. [33] makes decision on frame by frame basis and caches strategic video frames to improve upon end user QoS. [39] introduces the concept of prefix caching in which starting few seconds of video is cached which significantly reduces the start up buffering delay and consecutive portions of video can be downloaded in the background while the prefix is being played out. Another important concept related to caching is the identification and removal of redundancy of information across video files. This forms a broader area of study called Near duplicate information detection. UQCLIPS [40] provides an algorithm to detect 2 new identical video frames which can be applied to remove cache redundancy. [48] provides an algorithm to detect video frames which are either identical or transforms of one another. This technique can identify redundancy of similar frames across video files which have different transformed properties such as change in tone, color, contrast or featuring watermark inlays. Although in both of the above research, they prove the correctness of the algorithm, there is no study of similarity statistics of such video frames in a large video database. P2P caching [20] [21] promises to be an interesting idea which can be used to implement a massively scalable cache infrastructure. However P2P systems are dogged by issues such as unreliability, unavailability, churn and copyright issues.

All of the above caching strategies do not make any assumptions regarding video coding algorithm employed which otherwise can be exploited to gain additional efficiency on top of current caching schemes. With regards to video compression techniques, several advancements have been proposed and researched. H.264/AVC [44] is a major effort in taking a quantum leap forward in bit rate reductions. VP8 [4] from On2 technologies provides additional bit rate savings on top of H.264 sacrificing limited quality. In our research we do not intend to study the advancements proposed on top H.264 which deals with the block encoding process such motion prediction, transformations and quantization and concentrate more on advancements to entropy coding as our research work can be applied on top of any of the advancements implemented in the previous coding stages. With respect to entropy coding, there have been few advancements
to improve cost savings. [38] was a JVT draft proposal to implement CAVLC with truncated GOLUMB codes which do not require codebooks. However, codes generated by algorithms such as Huffman algorithm which require a codebook are more optimal than GOLUMB codes. JVT proposal indicates that these GOLUMB codes be generated once and used for all files which experience the same issues as current CAVLC coding. CABAC [32] is a context adaptive arithmetic coder which is a part of H.264 standard. Although it provides significant increase in savings over CAVLC, it far more complex and has additional issues which we discuss in 5.8. Some implementations of JPEG [11] such as Photoshop provides a mechanism to generate dynamic codebooks for variable length coding of still images and transport it as a part of the file. But such as mechanism cannot be applied to video coding as the codebook overhead is far more significant. Most of the above codecs and their advancements have been designed stand alone without considering any external dependencies such as caches.

To the best of our knowledge, our scheme of improving video coding based on variable length coding assisted by a caching strategy to achieve cost reductions in terms of increased compression gains is the first of its kind in its paradigm and in the implementation.
Chapter 3

Dynamic Variable Length Coding

3.1 Cache Aided Variable Length Coding

Figure 2.1 identifies the different components in a variable length coded file transfer. We propose our scheme of cache aided video coding, starting with discussing a generic Cache Aided Variable Length Coding. Figure 3.1 shows a generic setup of such a scheme for a network transfer. The client sitting in a campus network requests VLC compressed files from a server over the Internet. Caching nodes (CDN nodes) are distributed across the Internet, preferably closer to the client. Caches may contain any data that server wishes to store and serve to clients based on availability of content and proximity to clients. Clients also have access to local cache either on their local computer or an explicit cache node in their campus network. Any traffic between such a caching node and client is not considered to add up as additional traffic cost. In this section, we outline strategy of caching which can help reduce the overall traffic costs. As pointed out in the Section 2.1.1, Static VLC is a very popular method of coding in many compression schemes. We term it as the 'current' scheme, so that the current cost of traffic $C_C = C_{static}$. We outline a caching strategy in conjunction with Dynamic VLC which is termed as the ‘proposed’ scheme to achieve reduction in traffic costs to proposed $C_P$.

Comparing Equations 2.1 and 2.2, we can derive the following inferences.

1. The overall cost for dynamic VLC may be less than, equal to or greater than Static VLC.
2. The cost for dynamic VLC compressed file is always less than or equal to Static VLC.
3. The reason for this inequality is due to the presence of $C_{codebooks}$ which could dominate the compression gains of dynamic VLC.
4. The above inferences are generalized with Equations 3.1 and 3.2.
In this discussion, we propose the CDN nodes cache all the codebooks used by the server to encode all the files in its repository. If this codebook overhead is too large for cache, then only codebooks for popular content can be stored. The client would only download the file from the server and codebooks from CDN cache. With sufficient accumulation of codebooks in CDN cache, all the codebooks would be downloaded from the CDN cache by the client. Specifically the proposed traffic costs from the server’s point of view is given by Equation 3.3. Over the period of time, when the CDN cache has all the codebooks required, the hit ratio would approach 1 and Equation 3.3 would reduce to $C_{dynamic}$.

\begin{align}
C_S &= C_D \\
C_{dynamic} &\leq C_{static} \\
C_{server} &= C_{dynamic} + (1 - hitratio_{CDN})C_{codebooks} \\
C_{server} &\rightarrow C_{dynamic}
\end{align}
However, even with this setup the client needs to download all the files and related codebooks. So we propose a second level cache at the client either on the client’s computer or a dedicated caching node similar to above in the campus network. This is termed as the local cache. The client would look for codebooks first in local cache, next in the CDN cache and then with server in case of a miss. Over the period of time, when client has sufficient codebooks accumulated, client would only be downloading the compressed files from the server. The net traffic cost for client side of transaction is given by Equation 3.5. Over the period of time, increasing hit ratios would reduce the client side traffic costs again to $C_{\text{dynamic}}$. The client side cost has importance in our studies in the backdrop of the discussion about CDN pricing models in Section 1.2.

$$C_{\text{client}} = C_{\text{dynamic}} + (1 - \text{hitratio}_{\text{local}})C_{\text{codebooks}} \quad (3.5)$$

$$C_{\text{client}} \rightarrow C_{\text{dynamic}} \quad (3.6)$$

So In conclusion, the net traffic costs of the transactions over a period of time is reduced from the current $C_{\text{static}}$ to proposed $C_{\text{dynamic}}$, i.e. $C_{P} \rightarrow C_{\text{dynamic}}$. The 2-level cache further reduces the codebook overhead traffic. Consider the local cache at the client is cache coherent with the remote cache in the Internet. This can happen if the caching node at the campus network downloads all the codebooks from the remote cache. Then subsequent traffic costs from server point of view can be re-written as Equation 3.7 with diminishing codebook overhead costs.

$$C_{\text{total}} = C_{\text{dynamic}} + (1 - \text{hitratio}_{\text{local}})(1 - \text{hitratio}_{\text{CDN}})C_{\text{codebooks}} \quad (3.7)$$

### 3.1.1 Codebook granularity

Another design issue with Dynamic VLC is the granularity or the unit of encoding for which a custom codebook is generated. For example: While compressing a text file, we could generate a dynamic codebook for the whole file or 1 per page or 1 per paragraph. Finer granularity achieves higher compression but increases the overhead. This is especially true when symbol statistics such as frequencies exhibit considerable difference when measured at a file level to that at a page or a paragraph level. Adapting to frequencies for a particular paragraph produces more optimal codebook than 1 codebook adapting to aggregate frequencies of all symbols in the file. We discuss the Codebook granularity for Video coding in the Section 4.3
3.1.2 Codebook redundancy

As discussed in the previous section, codebooks at finer granularity might significantly increase the overall codebook repository size. To alleviate this problem we propose to identify and exploit redundancies that might exist among different codebooks and re-use them whenever possible. The input file is decomposed into $n$ encodable units. This could be for example a paragraph or a page in case of text files. During encoding of this unit, encoder generates a dynamic custom codebook for this unit and encodes this unit using that codebook and stores the codebook in a repository. If in future another unit is to be encoded and if we can find a matching codebook in the repository, we re-use the codebook instead of generating a new codebook. By reusing the codebooks during encoding, we reduce the overall codebook repository size thereby reducing the volume of codebooks transmitted/cached as well. To enable re-use of codebooks, we must prove that significant redundancies exists among codebooks. We prove the same for Video files in Section 5.3.

3.2 Cache Aided Video Coding

In this section, we discus how Cache aided VLC can be applied to video coding. As we recall from Section 2.2 Variable Length Coding is used in the last stage of video coding process in the codec standard such as H.264. We also noted that the current standard has an adaptive yet static implementation of variable length coding called CAVLC(See Section 2.3). We apply Cache aided dynamic VLC to video coding as follows:

1. The overall video content is divided into a large set of frames as the basic units of encoding.
2. During encoding, a dynamic Codebook is generated for each frame of a file based on actual symbol statistics.
3. A Codebook repository is built out of which some are then cached in both CDN cache and local client site cache.
4. We gain the advantage of compression gains achieved by Dynamic VLC over Static VLC.
5. We minimize the codebook overhead through caching and re-using codebooks generated from past frames to encode current and future frames.

Conceptually, the overall process can be viewed in terms of Phases in time studying the actions of users in terms of their uploads(which results in a Video encoding at the time of storage) and views(resulting in decoding of streamed content at the client). This is illustrated in the Figure 3.2. During the Phase 1(or gathering phase), users upload a set of video files onto
a media sharing site over a period of time. In this phase, for each file, codebooks are generated for each frame. The encoded video file is then uploaded onto the file server and the codebooks generated are uploaded onto the Codebook repository. These codebooks and the video files may later be cached. In Phase 2 (or re-use phase), after establishing a sufficiently large Codebook base, the next set of encodings try to maximise re-use of existing Codebooks to encode the frames. Re-usability of Codebooks offers 2 main advantages: 1) The size of Codebook repository is minimized. 2) Amount of Codebooks downloaded at the client or cached is decreased over time. In Phase 3 & 4, users view several videos by downloading Video streams along with Codebooks. To avoid repeated downloads of Codebooks, the clients cache the codebooks for the videos they have watched, on a local cache. Due to high degree of codebook re-use at the time of encodings, the clients would increasingly get a 'hit' for a codebook in the local cache and will gradually decrease the amount of codebooks downloaded. Higher compression achieved through Dynamic VLC reduces the net traffic flow on the network.
3.3 Dynamic VLC Coding Algorithm

In this section, we discuss the method to generate a dynamic codebook for a frame, used by CAVLC encoder at the time of encoding. We generate dynamic codebooks to encode level co-efficient(s) replacing the current set of 7 codebooks(VLC0-VLC6) with a single Dynamic codebook. A codebook is generated for each Frame $F_i$ of the video file which maps each of the level co-efficient(s) in $F_i$ to its variable length code. Algorithm 1 describes the generation of a dynamic codebook for a frame $F$ which is used by the CAVLC encoder while encoding frame $F$. Codebooks are generated based both on the composition and the individual frequencies of levels at the frame level. This avoids allocation of codewords to non-existent levels and adapts to actual frequencies of the levels present. In this way, we allocate shorter codewords to higher frequency levels present in the frame. Due to the extra pass made on the frame to generate a Codebook before the actual encode, there is an increase in the encoding times. But as the encodings happen only once(during video uploads), this time overhead can be neglected. The decoding algorithm works in the same way as the current CAVLC decoder, provided the codebook for the current frame is available at the time of decoding. The decoder reads in the bit stream either from file or off the network, scans and maps the bit stream to symbols as per the sequence described in Figure 2.3. During the decoding of level co-efficient(s), the bit stream is scanned for a matching prefix symbol. Each prefix symbol is reverse looked up in the dynamic codebook to obtain the corresponding level co-efficient. The level co-efficient(s) along with other syntax elements are used to calculate the residual component which is used in conjunction with a prediction to re-construct the original macroblock. The macroblocks are then re-arranged to retrieve the frame.
Algorithm 1: \texttt{generate\_code\_book}(F) - Generate a dynamic codebook for frame F

\begin{itemize}
\item[Data:] Frame F
\item[Result:] \(\Pi_F\), Codebook for frame F
\end{itemize}

begin

Partition F into MB Macroblocks \{4x4, 8x8, ..\}

\begin{algorithmic}
\Function{generate\_code\_book}{F}
\State \textbf{Data:} Frame F
\State \textbf{Result:} \(\Pi_F\), Codebook for frame F
begin

\State Partition F into MB Macroblocks \{4x4, 8x8, ..\}
\State \ForEach{\text{block } m \text{ in } MB}
\State \t /* Obtain the residual portion of the block */
\State \hfill \delta_m \leftarrow \text{Residual}(m)
\State \t /* Apply transforms and Quantize to obtain level co-efficients */
\State \hfill \phi_m \leftarrow \text{Transform\_And\_Quantize}(\delta_m)
\State \t /* Zig-Zag scan to re-order the block */
\State \hfill A_m \leftarrow \text{ZZ\_Scan}(\phi_m)
\State \t /* Update frequencies of each non zero co-effecient in the re-ordered array */
\State \hfill f_F(i) \leftarrow f_F(i) + 1, \forall i \in A_m \text{ and } i > 0
\State \EndFor
\State \t /* Compute codebook for frame F with huffman codes for each non-zero level in the decreasing order of frequencies. */
\State \ForEach{i \in \max(f_F)}
\State \hfill H \leftarrow \text{huffman\_tree}(f_F(i)), i > 0
\State \hfill f_F(i) \leftarrow 0
\State \EndFor
\State \Pi_F(i) \leftarrow \text{Add}(H(s, c)), \forall \text{ symbol } s \text{ and corresponding codeword } c
\State \Return \Pi_F
\EndFunction
\end{algorithmic}

end
Chapter 4

Cache Aided Video Coding

4.1 Cache Aided Video Encoding

Consider a particular video file Video-m being uploaded in phase 1. For each frame $F_i$, a codebook $T_i$ is generated and stored onto the repository. The aim in this phase is to gather as many diverse codebooks as possible. For each video file encoded, a meta-file is also created which maps each of the frame in the video file, with the codebook used to encode that frame. Meta-files contain a set of 3-tuple values - <frame-no>,<code-book hash>,<size> for each frame of the video file. The hash value is generated based on the level co-efficient(s) composition of the codebook. size represents the no. of entries in the codebook which is used as a heuristic for codebook searches(explained in Section 4.3). The meta-file is tagged to the original video file to instruct clients to download the meta-file when a corresponding video is downloaded for viewing.

Next consider a video file Video-n being uploaded in Phase 2. Due to the existing co-relation among frames across different files, codebooks can be shared across different files. We say that a codebook $C_i$ can be re-used for a frame $F_j$ if $C_j = C_i$ or $(C_i \subseteq C_j, |C_j| \leq |C_i| + LIMIT)$, where $C_i$ and $C_j$ are the codebooks generated for past frame $F_i$ and current frame $F_j$ respectively using Algorithm 1. We define a LIMIT in order to minimize the additional entries in case of superset match. The value is a trade-off between codebook match(re-use) and compression efficiency(codebook size). For example, in Figure 3.2 Video-n reuses codebooks $T_b$ and $T_d$ from past encodings. In our experiments we show that a majority of the frames encoded in Phase 2 (> 90%) re-use already existing codebooks. The encoding process is described in Algorithm 2.

4.2 Cache Aided Video Decoding

The Decoder runs at the client machine to where the video is streamed. It also maintains a local cache of codebooks for the video files, the user has viewed in past. The decoding process
Algorithm 2: \texttt{CAVEncode}(Video file F) - Generate an encoded video file, codebook and upload onto repository

\textbf{Data:} Video file V

\textbf{Result:} video file encoded and uploaded to the server

\begin{algorithmic}
\State begin
\State \texttt{/* Initialize the Output/Meta files */}
\State \texttt{output\_file} $\leftarrow$ \texttt{null}
\State \texttt{meta\_file} $\leftarrow$ \texttt{null}
\State \texttt{foreach Frame F in V do}
\State \texttt{/* Obtain the Codebook for frame F */}
\State $\Pi_f$ $\leftarrow$ \texttt{generate\_code\_book}(F)
\State \texttt{/* Compute hash on levels present in the codebook */}
\State $H$ $\leftarrow$ \texttt{hash}(e_1, e_2, ..., e_n) $e_i \in \Pi_f$
\State \texttt{/* Search the repository for a codebook match */}
\State $C = \texttt{Search\_Code\_Book}(H)$
\State if $C = \texttt{null}$ then
\State \texttt{/* Miss: Encode and Store the new codebook for future encodings */}
\State $B$ $\leftarrow$ \texttt{CAVLCEncode}(F, $\Pi_f$)
\State \texttt{Store\_Code\_Book($\Pi_f$, $H$, n)}
\State \texttt{/* Append tuple values to meta-file */}
\State \texttt{file\_write(meta\_file, F, H, n)}
\State \texttt{end}
\State else
\State \texttt{/* Hit: Re-use the codebook */}
\State $B$ $\leftarrow$ \texttt{CAVLCEncode}(F, $C$)
\State \texttt{/* Append tuple values to meta-file */}
\State \texttt{file\_write(meta\_file, F, C.hash, C.n)}
\State \texttt{end}
\State \texttt{/* Append bit stream to output file */}
\State \texttt{output\_file} $\leftarrow$ \texttt{output\_file} $\oplus$ $B$
\State \texttt{end}
\State \texttt{/* Tag the output file to corresponding meta file */}
\State \texttt{tag\_file(output\_file, meta\_file)}
\State \texttt{/* Upload the encoded file and meta file on to file server */}
\State \texttt{File\_server\_upload(output\_file, meta\_file)}
\State \texttt{end}
\end{algorithmic}
is summarized in Algorithm 3. When a user wishes to view a video, the client requests for the
file’s video stream. It also obtains the meta file associated with the video file. The client looks
up the codebooks listed in the meta file in the local cache and downloads from the Codebook
server if a local cache miss occurs. Given that a large number of codebooks are shared among
different video files during encoding, the decoder needs to download very few codebooks to
decode a large set of files. In our experiments, we observe that high degree of re-use during
encodings result in more no. of local cache hits at the decoder. In the Figure 3.2, phase 3
represents one such instance when, Video-m is being streamed. Decoder caches \( T_a, T_b, T_c, T_d \)
codebooks to local cache. Later when Video-n is streamed for viewing, the client need not
download the Codebooks as they are already present in the local cache reducing the overall net
traffic for that view. CDN nodes in the Internet itself can be viewed conceptually as a client
downloading video streams from media sharing site serving to all the clients in its region. As
CDN node caches the codebooks for the aggregate of the videos it is serving, the net traffic
between Main streaming servers and the CDN node is reduced over time. Thus achieving
reduction in server bandwidth and cache storage costs. Our experiments show that, over time
with several downloads, significant cumulative savings gain are achieved including the codebook
overhead.

4.3 Codebook Management

Codebooks generated during the encodings are stored in a Codebook repository which provides
3 basic functionalities. 1) Store a newly generated codebook. 2) Fetch a particular codebook
given its hash 3) Search for a matching codebook for a given frame to be encoded. The origin
media server stores all the codebooks from all the encodings of all the files in its database. The
CDN cache fetches and stores all the codebooks for the popular videos it caches. The local
cache on the client computer or in the campus network caches all the codebooks for videos
viewed by local user(s). Following are the major codebook management operations:

1. When a new video file is encoded, a meta file for that particular file is created which con-
tains the entries of all the codebooks it references. The Meta file is a CSV file containing
a number of entries of format \(<\text{frame-no}>, <\text{codebook-hash}>, <\text{codebook-size}>\).

2. A codebook hash is computed based on the composition of a particular codebook. i.e;
   the symbols present in a codebook have a 1:1 relationship with the computed hash.

3. A direct consequence of above is that 2 codebooks with same composition result in the
   same hash value.
Algorithm 3: CAVDecode(Filename N) - Download and Decode a video file for play-out/streaming

Data: Filename N
Result: Decoded Video file V
begin
  /* Download the video and meta files */
  input_stream ← File_server_stream(N)
  meta_file_tag ← get_tag(N,"meta_file")
  meta_file ← File_server_download(meta_file_tag)
  /* Initialize the decoded Output file */
  V ← null
repeat
  /* Read in the codebook hashes for each frame */
  frame_no, hash, size = read(meta_file)
  /* Search in local cache */
  \(\Pi_{frame_no}\) ← lookup(lCache, hash)
  if entry = null then
    /* Miss: Get it from server and store in local cache */
    \(\Pi_{frame_no}\) ← get_code_book(hash, size)
    insert(lCache, \(\Pi_{frame_no}\))
  end
until EOF ;
repeat
  /* Read in the video stream and decode the frames with corresponding codebooks */
  B ← read(input_stream)
  F ← CAVLCDecode(B, \(\Pi_F\))
  /* Reconstruct the video frame by frame */
  V ← V ⊕ F
until EOF ;
return V
end
4. A codebook $C_j$ is said to be a match for an input codebook $C_i$ if, $C_i = C_j$ or $C_i \subset C_j, |C_j| \leq |C_i| + \text{LIMIT}$, where $\text{LIMIT} \geq 0$. We assume that if 2 codebooks match in composition or if a codebook is a subset of the input codebook, then the matching codebook can be re-used during encoding. However 2 codebooks could match in composition, but not with individual symbol frequencies. However our experimental results show that this approximation doesn’t significantly degrade the performance of compression (See Section 5.4). The reason for specifying the LIMIT on the codebook size is to avoid additional unwanted entries in the codebook. Otherwise selecting a large super set for a given codebook might degenerate into Static VLC problem with codewords being allocated to unused symbols.

4.3.1 Codebook operations at the CDN Cache

The CDN nodes store popular content and the codebooks associated with it. As the popularity of the content changes over time, the cache must be replaced with new codebooks for the next set of popular content. Following are the codebook operations performed at the cache.

**Codebook Accumulation:**

1. Download popular videos from the origin server.

2. For each video $v$ downloaded
   - Download $m$, metafile($v$)
   - For each hash $h$ in the $m$
     - if lookup($h$) = miss
     - Download Codebook$_h$ and initialize reference_count(Codebook$_h$) = 1
     - else increment reference_count(Codebook$_h$)

**Codebook Cache cleanup:**

As the cache is replaced, codebooks need to be cleaned up to avoid unmitigated increase in the size of the cache. Following algorithm is employed to clean up the unused codebooks in the CDN cache.

1. Find next least popular video $v$

2. For each codebook $c$ used by $v$
   - if reference_count($c$) = 1 remove_codebook($c$)
   - else decrement reference_count($c$)
4.3.2 Cache operations at client cache

The local cache maintained at the client also perform similar operations on codebooks. The difference being, the clients download codebooks for only those videos watched by local users. Each codebook is also initialized with local reference count which is increment every time the codebook is accessed to decode a frame of a particular video. The cache clean up could be triggered for a different reason than the CDN cache. For example: triggered when the cache size limit is crossed. In such a case, all codebooks with reference count = 1 are removed, if the cache is still above threshold limit, then codebooks with reference count = 2 are removed and so on. Codebook lookups is a 3 step process with first lookup in local cache, if it results in a miss, then CDN cache is looked up and then finally origin server cache repository is looked up for the required codebook.

4.3.3 Codebook search

Codebooks are maintained in a set of hash tables in the Main server codebook repository. Similar structures are also used in caches. Each hash bucket represents a codebook size value. When a codebook lookup is performed, the size of the current codebook is also implicitly passed. The repository codebook hash table is searched by indexing the appropriate size bucket to find a perfect match. If a match is not found, the next size bucket is searched for a superset codebook. We would want to keep the size in check to avoid unnecessary entries. This is accomplished using a LIMIT parameter. The search function is described in detail in Algorithm 4.
Algorithm 4: \textit{Search\_Code\_Book}(H, S, HT\{MAX\_SIZE\}) - Search for a codebook with hash H and size S in the set of hash tables

\textbf{Data:} Hasht H, Size S, Hash tables HT[]

\textbf{Result:} CodeBook C - Best match codebook or Null

\begin{itemize}
\item[] 1 \hspace{1em} \texttt{begin}
\item[] \hspace{2em} 2 \hspace{1em} /* Lookup for C in the hash table for size S */
\item[] \hspace{2em} 3 \hspace{1em} \texttt{C \leftarrow lookup(HT[S])}
\item[] \hspace{2em} 4 \hspace{1em} \texttt{if C = null then}
\item[] \hspace{3em} 5 \hspace{1em} /* If we couldn’t find exact size match, search for superset */
\item[] \hspace{3em} 6 \hspace{1em} \texttt{for inc \leftarrow 1 to MATCH\_LIMIT do}
\item[] \hspace{4em} 7 \hspace{1em} /* Increment to next size hash bucket until we find a matching Codebook */
\item[] \hspace{4em} 8 \hspace{1em} \texttt{C \leftarrow X_i, X_i \in HT[S + inc], C \subset X_i, \ i = 1 to |HT[S + inc]|}
\item[] \hspace{4em} 9 \hspace{1em} \texttt{if C \neq null then}
\item[] \hspace{5em} 10 \hspace{1em} \texttt{break}
\item[] \hspace{4em} 11 \hspace{1em} \texttt{end}
\item[] \hspace{3em} 12 \hspace{1em} \texttt{end}
\item[] \hspace{2em} 13 \hspace{1em} \texttt{return C}
\item[] 14 \hspace{1em} \texttt{end}
\end{itemize}
Chapter 5

Implementation and Evaluation

5.1 Implementation

We implemented our video codec plugin using \texttt{x264} and \texttt{ffmpeg} open source library. \texttt{x264} is an open source H.264/AVC encoder popularly used in many applications. \texttt{ffmpeg} is a vast collection of open source codec libraries which has support to several media codecs spanning across image, audio, text and video. It is popularly used as \texttt{libavcodec} library. The Codebook repository is implemented as a stand alone server which communicates with sockets. The repository has the same implementation in both Servers and Caches barring specific operations specified in Section 4.3. All the implementations were on Ubuntu Linux kernel v2.6. Table 5.1 lists the open source libraries used in the implementations.

5.2 Evaluation

In our evaluation, we measure the savings in cost factors proposed by calculating the bit savings gained from Dynamic VLC over CAVLC which is the current scheme used in H.264. Once the advantage of dynamic VLC is established, we evaluate our approach of reducing codebook overhead by measuring the growth of codebooks and its effects on the net cumulative savings.

<table>
<thead>
<tr>
<th>Library</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>x264-devel</td>
<td>x264 developmental environment for encoder</td>
</tr>
<tr>
<td>libx264-67</td>
<td>x264 encoder</td>
</tr>
<tr>
<td>ffmpeg-devel</td>
<td>ffmpeg library</td>
</tr>
<tr>
<td>libglib-2.0</td>
<td>glib library 2.0 or greater</td>
</tr>
<tr>
<td>yasm</td>
<td>Assembler</td>
</tr>
</tbody>
</table>
Following observations are the goals of our evaluation.

1. What is the bit savings achieved through Dynamic VLC?
2. What is the effect of choosing different codebook granularity?
3. What is the measure of redundancy among codebooks?
4. What is the effect of codebook re-use on the compression?
5. What is the growth rate of codebook repository size?
6. What is the net cumulative savings including the overhead measured over time?

5.2.1 Bit Savings

In this section, we evaluate the savings gained from Dynamic VLC over static VLC. For the first set of experiments, we take a small subset of 18 standard video sequences. The sequences are part of the standard QCIF sequences [12] used in variety of video compression evaluation. Figure 5.1 shows the compression gains achieved by Dynamic VLC over Static VLC. We implement 2 schemes of DVLC. One in which a codebooks are generated for each frame and second, 1 codebook is generated per file. As per Figure 5.1, generating a codebook per frame achieves much better compression over CAVLC enforcing the hypothesis of generating a codebook per frame as more optimal. Bit savings are calculated per file as a % reduction in bits over CAVLC and is given by following equation.

\[
\% Bit \ savings = \frac{Bits_{CA VLC} - Bits_{DVLC}}{Bits_{CA VLC}} \times 100
\]  

(5.1)

However, generating codebooks per frame increases the codebook overhead and sometimes negates the compression gains achieved by Dynamic VLC. Figure 5.2 shows the total bytes for DVLC + Codebooks against the static VLC. As we can see, the cost of dynamic VLC and codebook could either be less than, equal or greater than the static VLC cost. To achieve low overhead, a large set of codebooks are required to form a repository which promote re-use of codebooks to exploit redundancy. To evaluate such a scenario, we require a larger set of video files and more extensive test cases. So we put together a large set of diverse collection of videos, collected from Youtube. Table 5.2 outlines the information about the input data set. A standard codec profile with acceptable video quality(30fps) was used. As the video files were converted to YUV format before encoding, adaptive bit rate option was set in x264 which selects appropriate bit rates to maintain the input quality.

Figure 5.3 shows the savings over CAVLC for aggregated bytes of video files encoded. After initial variations, the bit savings over time stabilized to about 5%. Bit savings were also
Figure 5.1: % Bit savings of DVLC over CAVLC

Figure 5.2: Total bytes Count for DVLC and CAVLC
calculated on individual file basis and Figure 5.4 depicts the CDF of savings per file. From
the graph we can see that over 50% of the files achieved more than 4.5% bit savings leading
upto a high of about 20% savings over CAVLC. Table 5.2 tabulates the diversity of the videos
collected in terms of video duration, categories and resolution quality. The videos picked are
also random in terms of views and popularity.

Table 5.2: Youtube data set statistics

<table>
<thead>
<tr>
<th>Source</th>
<th>Youtube</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of videos</td>
<td>10,000</td>
</tr>
<tr>
<td>Resolution</td>
<td>240p, 260p, 480p, 720p, 1080HD</td>
</tr>
<tr>
<td>Category</td>
<td>Entertainment, Music, Comedy, Random</td>
</tr>
<tr>
<td>Video Duration</td>
<td>&lt; 5 mins</td>
</tr>
<tr>
<td>Video popularity</td>
<td>Random</td>
</tr>
</tbody>
</table>

Figure 5.3: Aggregate % Bit savings of DVLC over CAVLC (Youtube videos)
5.3 Codebook Redundancy

To establish the case for re-use of codebooks, we measure the amount of redundancy that exists among the codebooks. For this set of experiments, we randomly select 2 sets of 5000 videos among our video collection. Set 1 is encoded and codebooks are generated for all 5000 videos. During the encodings of Set 2 videos, we try to re-use the codebooks collected during encodings of Set 1. During each encoding of a frame, a codebook is requested from the repository, a match results in a hit else a new codebook is generated. We measure the hit ratio for the all videos in the Set 2 and plot the values as a % per file. Figure 5.5(a) plots the CDF for % hit ratio per file for Set 2 videos. It is evident from the graph that 90% of the files had > 95% hit ratio for the codebooks. This establishes the fact that once sufficient videos are encoded and codebook repository is built, there is extensive redundancy that can be exploited in reducing the overall codebook overhead. We conduct 5 runs of this experiment by randomly choosing 5000 videos for both Set 1 and 2.
Figure 5.5: Codebooks redundancy and effects on compression
5.4 Effect of Codebook re-use on savings

As we described in the Section 4.3, we match 2 codebooks only based on composition. However if 2 codebooks with same composition differ in frequencies of individual symbols, compression might suffer. To evaluate the effects of re-use, we measure the drop in bit savings due to re-use. Figure 5.5(b) shows that 96-97% of time the drop is below 0.2%. Following steps are followed in this experiment.

1. Set 1: Encode all 10,000 files by generating a new codebook for frame.
2. Set 2: Encode all 10,000 files by re-using codebooks from previous encodings. i.e; search for a codebook, re-use if we match else generate a new codebook and store the new codebook in the repository.
3. Measure the bit savings for Set 1 and Set 2 individually over CAVLC.
4. Calculate the difference Set 1 savings - Set 2 savings and plot the drop in savings.

5.5 Codebook repository Size

Unmitigated codebook repository size could make this scheme inefficient. In this experiment, we measure the codebook repository size accumulated for all the video encodings. We encode all 10,000 videos and generate codebook repository and remove all the redundancies by eliminating duplicates. Figure 5.6(a) shows the growth of codebook repository with files encoded. The overall codebook repository size for 10,000 videos was found to be about 32MB. Figure 5.6(b) plots the codebook size as a fraction of video file size. 10,000 video files when encoded with dynamic VLC occupied about 29GB. As we can see from the graph, if we store about 7.23GB worth (about 2000 files), we need about 13.32MB (0.18%) worth of codebooks which is about the size on an average 1-2 video files. So caches can trade storage for bandwidth by reducing the no. of videos cached by a nominal amount to gain increased bandwidth cost reductions.

5.6 Codebook size Extrapolation

It is important to study the growth of codebook repository size in order to understand the storage requirements for codebooks. We proceed with this study by fitting a curve to the growth of codebook size as a % fraction of total video size. We observed that codebook fractional size follows a Power distribution given by Equation 5.2. Figure 5.6 shows the fitted curve which depicts a 'long tail' effect which implies that majority of the videos are encoded with small number of codebooks, but new codebooks, very few in number continue to get added for later
(a) Codebook repository size growth

(b) Codebook size as fractional % of Video file size

Figure 5.6: Codebook repository size
encodings. The result of this study enables us to approximate the required codebook size to store/cache particular amount of videos.

\[ y = 2.9x^{-0.319} \]
\[ R^2 = 0.9894 \]

This study helps us in designing the cache storage policies as tabulated in Table 5.3. For the set of our Youtube videos, we calculated the storage space required for storing popular content. If we consider the following set of videos to be popular, with Dynamic VLC, we can store additional videos and all the codebooks associated with it. For example, if the cache size is fixed at 3.3GB, we previously stored about 1000 videos coded with CAVLC. If we use dynamic VLC, we could store up to 1064 videos + the codebooks required to decode all of the 1064 videos. We not only improved the storage savings, but also increased the popular video service ratio by caching more popular videos.

We can also approximate the amount of codebooks required for all Youtube encoded for a large set of future videos. For example: according to Youtube, 24hrs worth of video is uploaded every minute. If we encode these videos at an average bit rate of 1Mb/s, assuming the same trend of uploads continues, we would require 22,705,920,000 MB or 21.14 Petabytes to store all the uploaded videos for the next 4 years. According to Equation 5.2, approximate codebook size required is 319GB which is negligible.

Table 5.3: Storage share for videos and codebooks with fixed cache storage

<table>
<thead>
<tr>
<th>Cache Size(GB)</th>
<th>Videos(CAVLC)</th>
<th>Videos(DVLC) with codebooks</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.30</td>
<td>1000</td>
<td>1064</td>
</tr>
<tr>
<td>6.40</td>
<td>2000</td>
<td>2145</td>
</tr>
<tr>
<td>9.48</td>
<td>3000</td>
<td>3187</td>
</tr>
<tr>
<td>12.66</td>
<td>4000</td>
<td>4322</td>
</tr>
<tr>
<td>15.64</td>
<td>5000</td>
<td>5344</td>
</tr>
<tr>
<td>18.95</td>
<td>6000</td>
<td>6493</td>
</tr>
<tr>
<td>21.70</td>
<td>7000</td>
<td>7534</td>
</tr>
<tr>
<td>24.79</td>
<td>8000</td>
<td>8615</td>
</tr>
<tr>
<td>27.90</td>
<td>9000</td>
<td>9622</td>
</tr>
<tr>
<td>31.06</td>
<td>10000</td>
<td>10000+</td>
</tr>
</tbody>
</table>
Figure 5.7: Power curve fitted for % fractional codebook repository size

Figure 5.8: Residual values for curve fitting, $R^2$ value of 0.9894 shows a 'good fit' for the curve
5.7 Net cumulative savings

In this section, we measure the cumulative savings measured from the client side considering all the overhead incurred. For this experiment, we populated the video views for all the videos and assigned ranks based on the views. We simulate a large number of video views based on these ranks among the 10,000 video input set. We calculate the cumulative bit savings over CAVLC with Equation 5.4. We define the overhead associated with each experiment.

\[
\% \text{ Cumulative Bit savings} = \frac{\text{Bits}_{\text{CAVLC}} - (\text{Bits}_{\text{DVLC}} + \text{Bits}_{\text{overhead}})}{\text{Bits}_{\text{CAVLC}}} \times 100
\]  

Equation (5.4)

For the first experiment, we assign Top 10% of the videos (by rank) as popular videos and cache them in the CDN. We generate 10,000 views according to the distribution of the ranks calculated. We define \(\text{Bits}_{\text{overhead}}\) experienced by the client = \(\text{Bits}_{\text{codebooks}}\) downloaded from the CDN cache. We calculate the cumulative savings for the views as per Equation 5.4. Figure 5.4 shows the savings for 5 runs. In each run, we randomize the set of videos and assign the ranks and generate views.

![Figure 5.9: % Cumulative Bit savings of DVLC over CAVLC (Cache: Top 10% popular videos)](image)

Figure 5.9: % Cumulative Bit savings of DVLC over CAVLC (Cache: Top 10% popular videos)
In this experiment, we increase the cache size in terms of videos from 500, 1000, 1500, 2000, 2500 and calculate the net cumulative savings. The experiment is run for 10,000 video views simulated according to the video ranks. We measure the $\text{Bits}_{\text{overhead}} = \text{Bits}_{\text{codebooks}}$ and the figure illustrates that we consistently achieve little less than 5% savings. Figure 5.11 shows the bytes delivered from cache and server. It illustrates that the bytes delivered from both server and cache are reduced leading to increased costs savings for server serving unpopular content and CDN serving the popular content. This results in reduced server bandwidth costs and CDN pricing costs.

![Figure 5.10: % Aggregate Cumulative Bit savings of DVLC over CAVLC for 10,000 views](image)

5.8 Comparison with CABAC

Context Adaptive Binary Arithmetic Coding (CABAC) [32] is the second option available in H.264 standard to entropy encode video syntax elements. CABAC works on the principle of arithmetic coding which sacrifices complexity to gain performance efficiency over CAVLC. CABAC is preferred for its compression gains (about 10% over CAVLC) in presence of a decoder which has sufficient computational power. However, CAVLC has been popularly adopted...
due to various reasons but has the disadvantage of being low in efficiency. Dynamic VLC provides an efficient way to increase CAVLC compression gains and bridge the gap to CABAC in compression efficiency. Following are important observations about CABAC.

- CABAC offers higher efficiency by trading off complexity.
- CABAC requires sufficient computational power with floating point operations. As a result is difficult to deploy on devices like Smart phones etc;
- CABAC decoding cannot be parallelized. As CABAC encodes the symbols into 1 message [46], either full message is decoded or none of it is.
- Arithmetic coding comes with licensing issues which make it harder to implement on open source codecs.

To measure where DVLC stands against CABAC, we encoded all 10,000 videos with CABAC and measured file size difference between DVLC and CABAC as %. Figure 5.12 shows the CDF of % bit savings of DVLC over CABAC per file. A mean of -4.2% was observed which bridges the gap that existed between CABAC and CAVLC to a certain extent. Also DVLC performed
better than CABAC in some cases. We do not intend to optimize DVLC to perform better than CABAC which is not possible. We try to achieve the positives of VLC in its simplicity and encoding/decoding speed and try to achieve performance closer to CABAC. Cache Aided Dynamic VLC makes CAVLC more viable for use.

Figure 5.12: % Bit savings of DVLC over CABAC
Chapter 6

Conclusions

Significant increase in video traffic volumes in the recent few years and forecasts of coming years threatens to burden bandwidth costs incurred by Video publishers. Current approaches of scaling the video delivery infrastructure such as Caching networks and advanced video compression schemes have proved considerably efficient to maintaining costs. However a new paradigm in video streaming is required in which caching infrastructure is used as a tool to aid in compression techniques to gain additional cost savings. Our research work explored possibilities of enhancing current video coding algorithms with the aid of caching infrastructure already existing in the Internet. We implemented this idea as an algorithmic improvement to Variable Length coding process of video coding by generating dynamic codebooks to encode video symbols. We outlined a caching strategy to minimize the overhead of codebooks resulting in reductions in overall net bit costs measured over a period of time. Furthermore, we implemented this scheme as a video codec plugin which contained video encode/decode operations together with cache management functions. We evaluated our work with a large set of Youtube videos and showed effective bit cost reductions. We also claim that this idea can be extended for coding all other types of media such as text, image and audio which use static variant of Variable Length coding.
REFERENCES


