Priority Progress Decoding

by

Lowell Kirsh

B.Sc., Queen's University, 2002

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science
in
The Faculty of Graduate Studies
(Computer Science)

The University Of British Columbia
September, 2005
© Lowell Kirsh 2005
Abstract

This thesis presents a framework for decoding video in the absence of sufficient computational resources. Most current decoding systems require there to always be sufficient resources available. We have implemented a video decoding module for the QStream video streaming system to adapt the decoding process to the available resources. A distinguishing characteristic of our framework is the ability to decode a video in priority order and drop low priority work if there are not enough resources to fully decode the video. In our particular case we adapt by dropping frames. Additionally, we make no use of a feedback loop as part of the data dropping mechanism. This is important because the variability of video content makes it difficult to predict the relationship between CPU requirements and video content. We argue that due to the inherent variability in both content and devices that an adaptive approach is necessary. We have found our approach to decoding to require minimal computational overhead.
Contents

Abstract ........................................................................................................... ii
Contents .......................................................................................................... iii
List of Figures ................................................................................................. v

1 Introduction .................................................................................................. 1
  1.1 A basic overview of the video decoding problem ..................................... 1
  1.2 Overview of our approach: the Priority Progress Decoding framework .... 2
  1.3 CPU Adaptive Decoding requirements .................................................... 3
     1.3.1 Prioritized video ............................................................................. 3
     1.3.2 Specialized decoder ....................................................................... 4
     1.3.3 Data dropping .............................................................................. 5
  1.4 Software Prototype .................................................................................. 5
  1.5 Thesis Statement .................................................................................... 5
  1.6 Outline .................................................................................................... 6

2 Background And Related Work ................................................................... 7
  2.1 Streaming ............................................................................................... 7
  2.2 Video Compression ................................................................................ 7
     2.2.1 MPEG ............................................................................................ 8
     2.2.2 Scalable Compression ..................................................................... 11
  2.3 Quality-Adaptive Streaming .................................................................. 12
     2.3.1 Priority Progress Streaming .......................................................... 13
  2.4 Real-time systems .................................................................................. 17
  2.5 Proportional share scheduling ............................................................... 18
  2.6 Reactive Programming ......................................................................... 19
  2.7 Summary ............................................................................................... 20

3 Description of Approach .......................................................................... 21
  3.1 The Implementation of Priority Based Decoding .................................... 21
  3.2 Frame priorities ..................................................................................... 22
  3.3 Assigning priorities to frame data .......................................................... 23
  3.4 Priority based decoding in XviD ............................................................. 24
     3.4.1 Difference between offset and priority based decoders .................. 24
     3.4.2 Buffering of frames ...................................................................... 25
     3.4.3 Decoder state ............................................................................... 27
     3.4.4 Frame re-ordering ....................................................................... 28
  3.5 Data dropping ....................................................................................... 29
  3.6 Summary ............................................................................................... 30
List of Figures

2.1 Decode/Offset (top) vs. Display Order (bottom) .............................................. 11
2.2 The structure of ADUs .................................................................................. 14
2.3 The Priority Mapper ..................................................................................... 15
2.4 Architecture of PPS ....................................................................................... 17

5.1 CPU time requirements (seconds) ................................................................. 46
5.2 CPU time requirements (seconds) ................................................................. 47
Chapter 1

Introduction

The world of digital video is headed in new directions. As the power of current computers increases, more and more computers are becoming powerful enough to play digital video. Already it is quite common to use a home computer for video conferencing, watching videos streamed over the Internet and playing DVDs. With the increasing number of portable devices equipped with colour screens, people are now looking at new ways of watching digital video. For example, many anticipate that they should soon be able to watch video streamed to their cell phones and they want to watch videos on their portable MP3 players. As desktop computers become more capable, people want to hold video conferences with several people at once.

However, there are several limitations to the current state of the art. People do not do video conferences with several others at once on their home computers and the processing capabilities of portable devices remains limited. While the limiting factor for this is usually network bandwidth rather than computational power, even systems with plentiful bandwidth are limited in what they can display. Of the current limitations, we are focusing on one, the decoding of video. Our overall goal is to improve video streaming capabilities across a wide range of general purpose platforms. Our work builds on existing work that has addressed some of the other limitations.

1.1 A basic overview of the video decoding problem

With increases in computing technology, we are constantly getting improved devices on the market. Among the improvements that come with time are increased memory sizes and increased processor speeds. There is also a trend towards portable devices now that it is quite possible to manufacture and sell pocket sized devices with colour screens and wireless networking for a reasonable cost. Although we continually get improved devices, this also means that diversity is actually increasing over time. We have increasing variability in both devices and content.

If we were satisfied with the current level of video quality, eventually even the lowest end players would catch up and be powerful enough to play today's most demanding content. However, as the power of high end technologies increases, so too do our demands for higher quality video. So while the low end devices are catching up to the high end devices of a few years ago, they are still unable to
keep up with our ever increasing demands for delivering higher quality video. It is the video decoding aspect of streaming with which we are most concerned in this thesis.

Although an individual video encoding may conform to a target average rate, over short time scales, the bit-rate requirements of the video will vary significantly. This is due to the inherent nature of the video content which is highly variable due to motion, changing camera angles, and changing scenery, among other things. The complexity of decoding is roughly proportional to the bit-rate, and thus the computational requirements of decoding are quite variable [5]. One might try to ensure the video they wanted to watch was encoded at a low enough bit-rate so that the device they were to watch it on would be able to meet the computational demands of decoding it. However this poses a problem because it would mean that every video would have to be encoded differently for each device it was to be played on. Also, due to the variability of the computational requirements, it might be the case that a given device could be quite capable of decoding an entire video in real-time, except during a small number of short intervals. So even if we knew ahead of time what our target video rate was, this rate ends up fluctuating and we can never be sure ahead of time that there will always be enough processing power to decode the whole video.

In addition to the desire to play a single video on a computationally limited device, we'd also like to play multiple videos on a single device. This places much greater computational demands on the device than playing a single video. There are many situations where we'd like to play several videos simultaneously including video conferencing with several people at once, and remote surveillance. As well as being able to play multiple videos at the same time, we would also like for their quality to be roughly the same rather than some of the videos' quality suffering due to the large computational demands of other videos. Achieving equal quality across videos is a difficult task because the relationship between video quality and resource consumption is complicated.

Since variability is inherent at several levels, we believe an adaptive approach is necessary. In an adaptive video decoding scheme, the video player could dynamically adjust the quality of the video playback depending on the level of resources available.

1.2 Overview of our approach: the Priority Progress Decoding framework

Our work is an extension to the QStream video streaming system [3]. QStream is a system for streaming video over best-effort networks with the goal of optimizing its use of the inherently variable network conditions. QStream consists of a server and a client program, as well as several other tools. As part of the system, a custom network streaming protocol as well as a custom video format were
designed and implemented. The protocol is called Priority-Progress Streaming (PPS). PPS tackles the problem of adapting video to the network. This thesis adopts and extends that approach to adapting to the CPU, hence the name Priority-Progress Decoding.

At the time of writing, the video format used by QStream is based on the XviD video codec and is called SPEG\[9\]. XviD is based on the MPEG-4 video coding specification\[4\]. Prior to any work on the priority based decoding, the video format had been enhanced with scalability features and is called SPEG. In order to implement priority based video decoding, we modified the client part of QStream to implement priority based decoding.

1.3 CPU Adaptive Decoding requirements

We are interested gracefully degrading the quality of a video when there is not enough CPU to decode the video at full quality. This is accomplished by selectively decoding only part of the video and dropping the rest of it. We use frame dropping as the means of matching resource use to availability, giving the system the ability to maintain timeliness. Choosing what work to drop is a non-trivial decision. It would be ideal to have some measure of what parts of a video are more important than others so that if we were to drop a portion of the video, we could drop the least important part. However, the SPEG format encodes information about pictures to be displayed on screen, with no encoding of which pictures are more important than others or which parts of a picture are more important than other parts.

1.3.1 Prioritized video

In order to decide which parts of a video we are to drop, we assign priorities to them, and choose to drop the parts with lower priority before parts with higher priority. But already when talking about parts of a video, we are being ambiguous. In order to decide relative importance of the parts of a video, we must decide upon a basic unit of work for the decoding of a video. Only once we define this unit can we begin to think about the relative importance of those units.

One of the most fundamental units of work we can think of is a video frame. A video is made of a sequence of frames (or pictures), and when the frames are played at a fast enough speed, the illusion of motion is achieved. If we were to only decode some of the frames of a video, we could still play the video although any motion would look more choppy.

Another unit of work we could think of would be partial frames. Using a scalable codec to encode each frame as a series of layers, a video could be encoded in such a way that it would be possible to.

\[1\] prior versions of SPEG were based on MPEG-1
decode frames partially, by only decoding some of the layers. Even if not all the layers for a particular frame were decoded, the frame could still be decoded enough for it to be displayed. The result would be that an individual frame of the video would be of lower quality than if it were decoded fully. There are other units of work which could be dropped as well. For example, audio quality could be reduced.

In our prototype we have chosen to limit how we adapt to insufficient CPU to dropping frames. That is, if we don't have enough CPU to decode the video, we will adapt by only decoding some of the frames. We subdivide the timeline into intervals during which we sequentially decode entire frames from highest priority to lowest priority. If there is enough time within the interval to decode all the frames in the interval, the video will be decoded at its full quality. If there is not enough time, we will have decoded the most important frames and dropped the least important. This approach is very similar to that taken by PPS, which was a successful strategy.

Once again, the importance of frames is not something that is present in the video data itself. We have to decide this. Because of the structure of the video format we use, there are some hard dependencies inherent in the data which affects the relative importance of frames. For instance, there are frames which are calculated as a combination of up to two other frames. If a frame is a combination of other frames, those frames must be decoded first. Thus we are constrained to giving those frames higher priority than any frames which depend on them.

That only gives us part of our priorities. We also must supply more priority information. There are several factors which help us choose priorities. We would like that the frames that we decode are as evenly spaced out as possible, so that we can have the smoothest video playback as possible. After devising a system for assigning priorities to the units of our video data, we can then decode the video in a priority based manner.

1.3.2 Specialized decoder

In the previous subsection, we mentioned that we prioritize the frames of video and, within a given time interval, decode the frames in decreasing order from highest to lowest priority. However, our priority order differs from the canonical order standard decoders use, which is generally the order which ensures that interframe dependencies are met and decoding latency is minimized. Thus we cannot use a standard decoder to do the job of decoding video given to it in priority order. We need a way of decoding our data in a priority order.

One option would be for us to design a custom video decoder able to accept video in priority order. However, the implementation of video decoders is far from being a simple task, and to get achieve high performance, they are often heavily optimized and contain custom assembly code for each platform they run on.
Instead we chose to extend a video decoder which has already been written. The SPEG codec upon which this thesis builds already makes use of a popular open source codec called XviD. We have modified the SPEG decoder to accept data in priority order. We have tried to minimize the changes that needed to be made to codec and keep as much of the logic as possible within the streaming client which uses the decoder. We felt this approach would be easier, since we are more comfortable changing QStream than XviD, and it should save time if we incorporate other codecs in the future.\footnote{For instance, the x264 codec is a likely candidate for future inclusion in QStream.}

1.3.3 Data dropping

The way we adapt to insufficient resource availability for decoding video is by dropping data of low priority. Thus a mechanism for dropping low priority data is required. We do this by using timers. As mentioned earlier, the timeline of the video is divided into intervals and within each interval the video is decoded in order from highest to lowest priority. Each interval represents a certain amount of time of video playback and we give the decoder an equal amount of time to decode the interval. Each interval has a timer associated with it which tells the system when to stop decoding that interval. If the timer expires before all the video in the interval has been decoded, the undecoded data will be dropped and the decoder will move onto the next interval. On the other hand, if all the video can be decoded, all the frames from the interval will be decoded and displayed with no frame dropping.

One novelty of our approach is that we do not use a feedback loop to determine when we drop data. This frees us from the need to predict CPU requirements in advance. To estimate the relationship between CPU requirements and video content is a difficult task due to the variability of video content.

1.4 Software Prototype

We have developed a software prototype to implement our system of priority-based video decoding. This prototype involved making changes to the streaming client of the QStream system. The changes pertained solely to video decoding. Modifications were made to the client itself as well as the SPEG decoder, which it uses.

1.5 Thesis Statement

This thesis presents a framework for decoding video in the absence of sufficient computational resources. A distinguishing characteristic of our framework is the ability to decode a video in priority order and drop low priority work if there are not enough resources to fully decode the video. In our particular case we adapt by dropping frames. Additionally, we make no use of a feedback loop as
part of the data dropping mechanism. This is important because the inherent variability of video content makes it difficult to predict the relationship between CPU requirements and video content. We argue that due to the inherent variability in both content and devices that an adaptive approach is necessary.

1.6 Outline

The rest of this document is organized as follows. Chapter 2 describes background and related work. Chapter 3 gives an overview of our approach to prioritized video decoding. Chapter 4 describes the implementation of priority based decoding in detail. Chapter 5 describes how we evaluated the system and the results we discovered. Chapter 6 describes our conclusions and possible future work in relation to this project.
Chapter 2

Background And Related Work

2.1 Streaming

Streaming media is the process of playing media while it is being delivered over a network such as the Internet. This is in contrast to having to wait for all the media to be delivered before beginning to play it. Although downloading is always an option, the instant gratification of streaming is an important enhancement. Just as the arrival of the hyperlink changed how people access documents on the Internet, streaming promises to enhance access to video and audio. Streaming is already being used for a wide variety of uses including watching video clips embedded in web pages and video conferencing.

2.2 Video Compression

Because of the large size of uncompressed video and the limited bandwidth of current networks, video compression plays a large role in the streaming process. Streamed videos are almost always compressed. Even at modest resolution, uncompressed video uses up a lot of bits. This is uneconomical when it comes to storing or transmitting it. Because of the costs of storing and transmitting data, it is desirable to compress video to reduce these costs. Since processors are increasing in speed much faster than networks are increasing in bandwidth, it is expected that video compression will be used for the foreseeable future. Using compression for video equates to a trade-off. The more a video is compressed, the less bandwidth its transmission requires, and at the same time, higher compression leads to decreased perceptible quality. There currently exist many compression schemes, and creating better codecs is a focus of much current research [8].

There also exists another trade-off with video compression. The tendency is that as codecs are increasing in their ability to compress video to smaller sizes, they are also increasing in the amount of processing power needed to do the compression/decompression. And with processors increasing in speed much faster than networks, this trend should continue as well.

Uncompressed video has a constant bit-rate. Each frame is composed of a fixed number of pixels and each pixel uses the same number of bits (usually 16). On the other hand, the bit-rate of compressed video tends to fluctuate. Video content is inherently variable. There are random
patterns of movement, scene changes and changes of camera angle. With predictive video encoding, certain frames (difference frames) are encoded as the difference between themselves and other frames (reference frames). This inherent variability in the video content leads to variability in the compressed video bit-rates. In cases where there is less variation in the original video, less bits are required to encode difference frames because of their similarity to their reference frames. On the other hand, with greater variability in the original video, more bits are required to encode difference frames because there is a greater variation between the difference frames and their reference frames.

2.2.1 MPEG

MPEG, short for Moving Pictures Experts Group, is both a set of standards and a group of organizations and individuals responsible for administering the standards [7]. The standards set out to specify many facets of digital multimedia, especially pertaining to video, audio and pictures. These standards have been quite successful over the years and most personal computers now have the capability of working with MPEG media [1]. Since we are talking about video streaming in this thesis, and MPEG video streaming in particular, our discussion of MPEG will be limited to the video portion of the MPEG video standards. In the rest of this thesis, the term MPEG will be used to refer to the standards only, unless otherwise noted.

The MPEG standards facilitate the compression and decompression of digital multimedia. A couple of examples may prove instructive. One popular use of MPEG compression is for video storage. In this case, the video is compressed so that it will use less storage space. The video is encoded by an MPEG encoder (or compressor) and stored on some type of storage media, usually a disk. To view the video, it must be retrieved from the disk and decoded before it can be viewed. While the steps of encoding and decoding add complexity to the process, the savings in storage space usually make this approach preferable to storing the uncompressed video itself. Another common use of compression is for the transmission of video over a network. In this case, the video is compressed by the sender, sent over the network, and then decompressed and viewed by the receiver. In the case of transmission, usually the bottleneck is the network, so compressing the video before sending it will result in a more efficient transmission.

The MPEG video standards each define a bitstream for digital multimedia. What is meant by the term bitstream is a format for the data which, when decoded, will produce a representation of the original media. Since the media is compressed, the bitstream will not be an exact representation of the original, but instead will be a similar representation. The degree to which it is similar to the original is a function of how compressed the data is. Usually more compression of the data leads to less similarity to the original.
Currently there are three MPEG video standards: MPEG-1, MPEG-2, and MPEG-4. MPEG-1, MPEG-2, and MPEG-4 are all similar to each other in the basic way that they work and what they try to do. Their main goal is the compression and decompression of digital video and audio. While this thesis will concentrate on MPEG-4 in particular, much of the concepts are directly applicable to both MPEG-1 and MPEG-2. MPEG-4 was chosen over the other 2 standards because of its greater suitability to video streaming because of its higher compression rates.

How MPEG works

MPEG4 is quite a complex standard comprising of many features which are rarely used. Due to the limited scope of this thesis we will only be concentrating on a subset of those features and will tend towards a simplified view of things.

MPEG video is a sequence of frames. A frame is synonymous with a picture. When frames are displayed in sequence at a high enough rate, they are perceived as motion. MPEG compression is lossy, meaning that when a video is compressed and then decompressed, the resulting video will not be identical to the original. This is acceptable because in the case of video, a dramatic amount of compression can be achieved with only a small decrease in perceptible quality.

MPEG compression takes advantage of the presence of redundancy in video. This redundancy takes two forms. The first form is known as spatial redundancy. This refers to redundancy within a single frame. For example, a frame may contain a large area in which all the pixels are the same, or similar, values. In this case a transformation can be made on that area in order to reduce the number of bits required to encode the data. MPEG makes use of a technique called the discrete cosine transform (DCT) to encode spatial redundancy.

The second form of redundancy taken advantage of in MPEG compression is called temporal redundancy. This refers to similarities between successive frames in their display sequence. A simple example of this is a video of a stationary object. In this case, each successive frame is very similar to the previous frame, and techniques can be used to take advantage of this fact to compress the video greatly. MPEG uses a technique called motion compensated predictive coding to encode the temporal redundancy.

Interframe coding is based on reference frames and difference frames. A frame may exist as a difference frame and as an reference frame at different points in its lifetime. Reference frames are frames which are used as a basis for decoding difference frames. Difference frames are frames which are expressed as a bit-wise difference of themselves and one or two reference frames.

The simplest example of an reference frame is an intra frame (I-frame). These frames are encoded independently of any other frames. The encoding technique used is similar to that used in the
JPEG format, a common format for storing digital photographs. Because I-frames are encoded independently of any other frames, their encoding is unable to make use of temporal redundancy and only makes use of spatial redundancy.

To make use of temporal redundancy, difference frames are used. There are two types of difference frames. Backwards predicted frames (P-frames) are encoded in reference to the most recent previous reference frame. Bidirectionally predicted frames (B-frames) are encoded in reference to the closest past and future reference frames.

P-frames are difference frames while they are being decoded, but once they are decoded, they can act as reference frames. Once a P-frame has been decoded, it may become a reference frame for subsequent difference frames which make use of backwards projection. It will remain a reference frame until some future time when it is discarded.

The fact that B-frames can depend on frames which come later in display order has implications on the ordering of frames. Since B-frames rely on future frames, in a sequence such as I,B,B,P, the 2 B-frames between the reference frames cannot be displayed until they have been decoded, but cannot be decoded until the P-frame has been decoded, since they are coded relative to that P-frame. There are two ways this could be dealt with. The frames could be transmitted in the order in which they are to be displayed. If this were the case, both the B-frames would have to be kept in memory in their encoded form until the P-frame arrived, and only then could they be decoded and displayed. The drawbacks of such an approach are that the decoder would need more buffers to hold the B-frames until the next reference frame arrived. In fact, the number of B-frames which can occur in a row in display order is unbounded, which means that a decoder would need an unbounded amount of buffer space to hold the B-frames in their encoded form. Also, there would be a latency between the arrival of the B-frames and their display, which would grow proportionally to the number of consecutive B-frames. In situations where latency is to be avoided (such as streaming live video), this could be a big problem. However, usually in latency sensitive applications, B-frames are either avoided or constrained to runs of one or two in a row.

MPEG deals with the situation differently. Rather than decode the frames in the same order that they are displayed, reference frames are always decoded before any difference frames which rely on them. Figure 2.1 shows the difference between the display order and the offset order for a sequence of 4 frames. The top row of frames are in offset order and the bottom row are in display order. Each frame is labeled with its type. An arrow from a frame A to a frame B means that A is a difference frame which relies on the reference frame A. It can be seen from the diagram that in display order, both of the B-frames depend on the P-frame which comes later than them in the sequence. On the
other hand, in offset order the P-frame occurs before both of the B-frames and thus all dependencies are on frames which occur earlier in the sequence. In the sequence the P-frame would be sent before both of the B-frames, since they depend on the P-frame for their decoding. When each of the B-frames are given to the decoder, both of the reference frames upon which they rely have been decoded and the B-frames can be decoded and displayed immediately. Thus, in an MPEG bitstream the ordering of the frames is not the same as the order in which they are displayed.

![Diagram showing decode/offset vs. display order](image)

**Figure 2.1: Decode/Offset (top) vs. Display Order (bottom)**

### 2.2.2 Scalable Compression

Typically when beginning a video stream, a target data rate is selected. After this, the video is transmitted at a constant average rate for its duration. Often when streaming from a website, the video author will have compressed the video at several data rates, and the video consumer can choose one of the videos depending on their connection speed. The drawbacks of this approach include the
storage overhead of storing several copies of each video, the coarse level of granularity for matching a video to the available bandwidth and the under/over utilization of available bandwidth.

Scalable compression is a new form of video compression that overcomes some of these shortcomings [2]. Scalable compression supports multiple quality levels by creating layers in the compressed video. With all the layers present, the video can be displayed at its maximum quality. With some of the layers not present, the video can still be displayed but with reduced quality. In the layering scheme, there is a base layer as well as one or more enhanced layers. (The number of layers is proportional to the granularity at which the video can be adapted). The base layer represents the lowest quality of the video and each of the other layers represent an enhancement to this quality. The layering is progressive, that is each layer depends on all of the layers below it and adds quality to that of the layers below it. Having such a scheme removes the need for maintaining multiple encodings of a single video.

QStream uses a video format called SPEG (scalable MPEG) to implement scalable compression. The current implementation of SPEG is based on the XviD implementation of MPEG-4. SPEG adds basic spatial scalability to MPEG. Each frame is encoded with eight levels of spatial scalability - one base layer and seven enhanced layers. With all the layers present, the video can be displayed in full quality. However, if some of the layers are not there, usually due to data being dropped during network transmission, the frame can still be displayed, albeit with reduced quality.

2.3 Quality-Adaptive Streaming

Adaptive streaming is an approach used by QStream to maximize bandwidth utilization during network video streaming. The streaming is adaptive in that the video quality can be dynamically adjusted during the streaming process to maximize its use of the available bandwidth, while avoiding interruptions due to buffer underflows.

Video and network rates are quite variable and the current popular streaming systems don’t take that into account. Due to the inherent variability of video that has been compressed, data bit-rates for compressed video are variable. However many systems try to keep the bit-rate of the video stream constant based on the assumption that available bandwidth will be constant. This assumption is often wrong in best-effort networks such as the Internet where there are no quality of service guarantees. In order for video to be encoded at a constant quality level, the bit-rate must vary with the variability of the video. Keeping a video’s encoded bit-rate constant leads to video of variable quality or wastes bandwidth with padding data. We’d prefer the quality of the video to stay constant from the perception of the viewer and thus have the bit-rate be variable. A viewer will
notice and may be annoyed by quality fluctuations, but fluctuations in bit-rate are not perceivable to a viewer.

Network rates are another source of variability. Due to the best-effort nature of the Internet, any stream will have to compete with other network traffic. If any of the links on a path from a sender to receiver become saturated, a drastic reduction in bandwidth can occur, and packets may be dropped. Thus we make it a basic assumption that network traffic on the Internet is variable.

A basic goal in streaming is to use as much of the available bandwidth as possible, but no more. If the video requires too much bandwidth, the player will have to pause playback to wait for data to arrive. If the video uses less bandwidth than is available, the result will be a lower quality video than possible. Thus matching the bit-rate of a video to the available bandwidth as closely as possible will result in video of the highest possible quality.

QStream uses quality-adaptive streaming to deal with these problems and maximize its use of the network. Quality-adaptive streaming in effect adjusts the compression ratio of the video dynamically, thus maintaining the timeliness of video playback. It is able to do this dynamic adjustment because it uses scalable compression for the video. QStream implements quality-adaptive streaming through its Priority Progress Streaming (PPS) algorithm.

2.3.1 Priority Progress Streaming

PPS is a protocol for adapting to network bandwidth variability during streaming. The PPS algorithm works in various stages. Before data is sent over the network, it must be prepared. The timeline is divided up into intervals called adaptation windows. The data for each adaptation window is prioritized by the sender so that it can be sent in priority order. Once all the data from an adaptation window has been prioritized, the streaming of that window can begin. It is sent over the network in priority order until either all data from the current adaptation window have been sent or until the time allotted for transmission of that window runs out. If the time runs out for a window, all unsent data is dropped by the server. Once the transmission of an adaptation window has finished, transmission of the next window can begin. The final stage of the process is for the receiver to re-order the data it has received from priority order into offset order. Once it has been re-ordered it can begin to be decoded and displayed.

The data units of PPS streaming

Over the course of the streaming process, the multimedia data will take on various forms. At the beginning, it starts out as a bitstream with the audio and video components interleaved in the stream. In our particular case, the bitstream is an SPEG bitstream. The source of the stream will
usually come from either a file or a live source such as a webcam. In order to begin preparing the video for streaming, the video is then divided up into application data units (ADUs). Each ADU represents a contiguous fraction of the bitstream, and each represents either video or audio. Each video ADU represents one spatial layer of a video frame. Thus for each frame there will be eight ADUs, representing the eight spatial layers used in SPEG. There will also be a smaller number of ADUs which represent metadata about the video such as headers which come before each group of pictures. Each ADU takes the form of a packet with a fixed length header and variable length payload. Figure 2.2 shows the structure of the ADUs. Although in the implementation there is one base layer and seven enhancement layers, the picture only shows one base and one enhancement level for simplification.

![Figure 2.2: The structure of ADUs.](image)

These ADUs are not themselves streamed but are assembled into larger units which are streamed. The basic units of data sent over the network are called streaming data units (SDUs). Each SDU contains a group of ADUs as well as a timestamp and priority. These are the most basic unit of work in the network transmission stage. Although they are divided up into fragments before being transmitted, if any SDU is only partially delivered, it will be dropped by the receiver. In summary, it is the job of the sender to divide the bitstream into ADUs, and then to collect the ADUs into SDUs which are then streamed.

**Priority mapping**

There are many ways the ADUs could be collected into the SDUs. It is the job of the priority mapper to choose how the SDUs will be constructed. Because video has more than one quality
dimension, there is more than one way to adapt its quality. The two quality dimensions which are currently adapted by QStream are spatial quality and framerate. It should be noted however that while QStream only adapts these two dimensions, the approach it takes is more general and can be used for adaptation to any number of quality dimensions. QStream uses a component called a priority-mapper to adapt the quality of a video based on user preferences.

How to adapt a given media stream depends on its usage. When streaming to a small screen, spatial detail may be deemed less important than framerate. On the other hand, for video that is to be displayed in slow motion, spatial detail would probably be deemed more important than framerate. Whatever the scenario, adaptation policy should be specified explicitly through utility functions according to content, user, or device considerations. These policies act as input to the priority-mapper and are specified in the form of utility functions. The mapper provides a mapping which gives maximum utility. The other input to the priority-mapper are the ADUs. The output is the SDUs. Figure 2.3 shows the structure of the mapper.

![Figure 2.3: The Priority Mapper](image)

The mapper works as follows. First it divides the timeline into time intervals called map windows. The boundaries of a map window are chosen to prevent broken data dependencies between different map windows. Next, for each map window, the mapper will partially order the data based on all possible hard dependencies. It will then augment the order with soft dependencies to improve the results of the mapping, for example, ensuring the dropped frames are evenly spaced. Next it uses a user specified adaptation policy to refine the ordering. This policy states user preferences the relative importance of the different quality dimensions of the video. Once priorities have been assigned are
then grouped into SDUs. Each SDU will hold all the ADUs from the map window which share the same priority. The priority of the SDU will be set to the same value as the ADUs of which it is composed. The timestamp of each SDU in a given map window will be equal to the timestamp of the beginning of the map window itself. Thus the mapping transforms a set of unprioritized ADUs with varying timestamps into a set of SDUs with equal timestamps, each having a unique priority within the map window.

Of all the elements of the PPS algorithm, the mapping of priorities to data as done by the mapper is of most importance our approach to decoding of the video by the receiver. Before the development of priority based decoding, the priorities of the streamed data were transient values which only existed for the duration of the transmission phase. However, in order to assign priorities for decoding the video, the decoder was modified to make use of the data priorities. As will be seen later, there are many similarities between the priority mapping done at the network level and the priority based decoding done later.

**Streaming over the Network**

The purpose of organizing the data into SDUs is to prepare it to be streamed over the network using the PPS protocol. The PPS protocol divides the timeline of the video into adaptation windows, each of which contains one or more mapper windows. Once the data has been prepared into SDUs, it leaves its preparation phase and is ready for the next phase which is the transmission phase. This is the phase where PPS sends as many of the SDUs within a given adaptation window over the network as possible within a given time frame.

Figure 2.4 shows the architecture of PPS. Two buffers sit on either side of the bottleneck, which in the case of network streaming might be the TCP transport. The upstream buffer is part of the server and the downstream buffer is part of the client. Once an adaptation window has entered its transmission phase, the upstream buffer is where all the SDUs which have not yet been sent are stored. Over the course of the streaming of an adaptation window, the SDUs flow from the upstream buffer over the bottleneck and are collected in the downstream buffer on the client.

At the end of the transmission of the current adaptation window its transmission phase ends and the transmission stage of the next adaptation window begins. If there was sufficient bandwidth to transmit all of the video data over the network, at this point the upstream buffer will be empty, with all the SDUs having been transmitted and now residing in the downstream buffer. However, as will often be the case, there will not have been enough bandwidth for all SDUs to be transmitted. In this case, upon the current window’s expiry, all SDUs in the downstream buffer will be discarded.
2.4 Real-time systems

Digital video is an instance of a class of applications which are called real-time systems. Whereas in most applications a goal is often simply to get a computation done as quickly as possible, in real time systems, time plays a more important role. In such systems, it is important for events to occur at (or sometimes before) certain deadlines. If a deadline is not met, the situation is problematic. The severity of the problem depends on the application.

Part of what makes up the fabric of a real-time system are events and their deadlines. An event represents a computation or an action to take place. An event may have associated with it a deadline, which represents the time when the event must either take place, or the time at which the carrying out of the event must be completed. Not all events have deadlines.

Real-time systems come in two general categories. In soft real-time systems such as a digital video system, meeting deadlines is important, but not imperative as it is in hard real-time systems, where a failure to meet a deadline can be catastrophic rather than just problematic. One might think of soft real-time systems as real-time systems in which missing a deadline won’t cause anybody to be harmed. In a hard real-time system such as nuclear power controllers, or medical equipment, a failure can lead to human harm or death.

In a soft real-time system, missed deadlines usually lead to non-optimal performance. If a video system were not to display a frame when it should, the video would look jittery. When a video is played, it is important that the rate at which frames are displayed on the screen is both fast and uniform. If a video is meant to be played at 30 frames per second (fps), there will be 30 times per second when a new frame should be painted to the screen. These are some of the deadlines of the system. Each frame has associated with it a time which it should be painted on the screen. If this
deadline is not met, the frame can either be discarded, or painted on the screen after its deadline. When a frame is discarded, the frame that was previously painted on the screen will remain there for longer than it should, which will result in the video having a slower frame rate. On the other hand, if the policy of the system is to not drop frames, but rather display them as soon as possible after a missed deadline, the frame rate will end up varying which will also result in the video looking jittery. For example, objects in the scene which move at a constant rate will appear to change their speeds over time. For the video to look good, it is important that deadlines for displaying frames be met, but if they cannot, there must be a policy which will minimize the effects of the missed deadlines. One example of a policy would be to never display a frame that has missed its deadline by more than 5 ms.

Real-time requirements require programs to be written differently than programs which have no real-time requirements. The notions of time and deadlines play a central role in such systems. A primary goal is that deadlines be met. If an event is scheduled for a particular deadline, it is important that the resources (such as the CPU) needed to satisfy the deadline be free in order to service the event. In a video streaming system, there are many things going on at once: video display, video decoding, and network activity among others. In order not to miss deadlines, it is important that trying to complete one of the events does not prevent another event from meeting its deadline.

2.5 Proportional share scheduling

When playing multiple videos at once, we would like to have balanced quality across the videos. If we allocate the CPU to be shared proportionally between them, the quality of the videos will differ. This is due to the inherent variability of video. The bit-rate of each video will vary over time, and thus the computational requirements to decode each video will vary over time. Also, the relative computational requirements for decoding any of the videos will vary over time. If there were not enough resources to decode all the videos fully and each video received an equal share of the processing power, the relative quality levels of each video would fluctuate. We would prefer for the videos to all be of equal quality, or if not, their quality should be related by specified proportion, for example, to achieve equal utility.

In real-time systems it is often desirable to share resources between consumers fairly according to some policy. In our case the consumers are videos and the resource to be shared is the CPU. The OS scheduler allocates a specific fraction of the CPU to each of the videos. An example of a policy for sharing the CPU between videos is for each video to receive an equal fraction of time from the CPU. This assumes that the videos are each of the same priority. If this is not the case, the policy
could be to give more of the CPU to the more important videos, where we have some metric of what is meant by importance. It could be that the video in focus is the most important or it could be that importance of a video is based on its size on the screen.

In our case we are looking to share the CPU between videos. As the system is currently implemented, all videos are of equal importance. What we'd like is for all the videos to be of equal quality. That is, we would like quality to be the metric for how to distribute the CPU. From the OS's point of view, it is very difficult to model the relationship between video quality and related resource usage. The OS scheduler, if it is to be general, should not know anything about application level metrics such as video quality.

If the job of scheduling the videos were left exclusively to the OS scheduler, it would be extremely difficult to share the CPU to allow the videos to achieve equal quality. The OS scheduler would have a much easier time ensuring that each video received an equal share of the CPU, but due to the variability of the videos, this would not ensure equal quality. Scheduling the videos for equal quality requires application specific information which is not available to the OS scheduler.

2.6 Reactive Programming

Since we are trying to share the CPU between videos which are all produced by the same application, it makes sense to try to keep scheduling decisions within the application, rather than have the OS make the decisions. The application has full knowledge of application-specific things such as quality so automatically has more information to work with than does the OS scheduler. The approach taken by QStream, and thus PPD, is to keep as much of the computation as possible within a single process and have the process be responsible for scheduling its own events. This reduces the overhead of context switching and keeps the system more responsive.

We use reactive programming as our programming model. Reactive programming is so called because it places an emphasis on reacting to events. Reactive programming has been used successfully in the implementation of QStream. Central to reactive programming are events and their handlers. In our case, the reactive model balances resource usage across multiple videos. With priority information, it ensures that highest priority events happen first. In our system, decoding is a step of the reactive model.

QStream implements the reactive programming model by implementing two libraries. The first is the GAIO (Gnu Asynchronous IO) API. The GAIO API provides an event driven interface for asynchronous input/output (IO). Traditional IO APIs are synchronous, which means that each IO operation will only return once its result is complete. In contrast, with asynchronous IO, each IO
request will result in the generation of a separate IO completion event to run in the future if the IO request would not return its results immediately. This leads to a much more responsive system. In addition to the GAIO library, QStream provides a middleware library called QSF (Quasar Streaming Framework). QSF provides high level networking functions, a debug tracing facility, as well as several other useful primitives.

In QStream events are non-preemptive. This means that once an event begins, it cannot be interrupted. To keep the system responsive we make sure to keep the run-times of individual events to a minimum. If an event may take too long to complete, we must divide it into multiple events.

Every event has a priority and a time of issuing. When the event dispatcher chooses an event to execute next, it chooses an event of the highest priority to run. If multiple events have the same priority, they are served in the order that they were issued. There are three types of events. The first type are deadline based events. These are scheduled to run at a time specified in the future and have the highest priority. They are often used to implement timers. IO completion events are another type which correspond to the completion of an IO request. Since all IO in QStream is asynchronous, any IO operation which cannot return its results immediately will generate an IO completion event to happen when the result of the IO becomes available. The last type of event is immediate events. They are scheduled to happen as soon as possible, meaning as soon as no other events of higher priority are outstanding. These types of events are used for much of the computationally expensive tasks such as video decoding. The application can assign priorities as it sees fit to both IO and immediate events.

2.7 Summary

This chapter focused on video streaming along with various related technologies. Video is usually compressed to facilitate streaming so we gave attention to MPEG compression as well as the SPEG scalability extension. We discussed PPS streaming along with its video decoding and network considerations. Finally we discussed real-time systems and related technologies and techniques including proportional share scheduling and reactive programming.
Chapter 3

Description of Approach

The previous chapter outlined related work and background information related to video decoding and streaming as well as some considerations about systems used to perform these tasks. Particular attention was paid to streaming video and in particular the QStream video streaming system. This chapter will outline what was involved in implementing priority based decoding at a high level, and the next chapter will describe it in more concrete detail.

3.1 The Implementation of Priority Based Decoding

The system that we implemented uses a new approach to video decoding called Priority Progress Decoding (PPD). It name is purposefully similar to the Priority Progress Scheduling (PPS) protocol used in QStream, which is the high level network protocol used for dynamic network bandwidth adaptation. The names are similar due to the fact that there is so much in common with both approaches to resource adaptation, as well as the fact that they are both parts of the same system. The approach taken by PPD to adapt to the amount of available CPU is similar to how PPS adapts to network bandwidth availability, except that instead of adapting to network bandwidth, PPD adapts to CPU availability. Priority based decoding used by the CPU scheduling algorithm is an analogue to packet re-ordering at the network level. Both approaches require no feedback loop and are able to adapt solely by dropping low priority data when deadlines expire.

The basic premise of how PPD works is that it will drop work depending on the availability of resources, in particular the availability of the CPU. Because video decoding is CPU intensive, the main resource that PPD adapts to is the CPU. We have developed a specialized video decoder to decode the video based on prioritized units, rather than decoding it the traditional way, based on offset order.

The PPD decoder divides the timeline into windows (corresponding to PPS map windows), each with a given duration determined by the number of frames in the window. These windows are referred to as decode windows or PPD windows. Rather than eagerly trying to decode all the video in a decode window, the PPD decoder will assign priorities to the units of data in the window based on what it deems to be the more important parts of the video for decoding. When decoding the
video, the decoder will give priority to decoding higher priority units. For each priority level, the decoder will try to decode all the window's data at that priority level before starting to decode lower priority video. One can think of the decoder jumping around the current decode window, decoding frames in priority order, rather than the traditional way of decoding video in its natural offset order. The decoder will stop decoding data in the current window as soon as either all the data have been decoded (in the case of enough CPU availability) or when the it runs out of time for decoding the current window.

In the current implementation of the PPD decoder, we only adapt to one quality dimension, framerate, by dropping frames. However, priority based decoding is not inherently limited to adapting to framerate only. This technique could also be used to adapt to any number of quality dimensions such as spatial quality within a frame and even audio quality\(^1\).

### 3.2 Frame priorities

Since we are adapting quality based on dropping frames, the frame is the basic unit of work to which we assign a priority. It is not trivial to assign priorities to frames in a decode window. The MPEG-4 video standard upon which SPEG (the format used in QStream) is built, has no concept of priorities for video data. The only thing standardized by MPEG is the video stream itself. It is a characteristic of a particular decoder implementation of how it behaves when the computational resources it has access to do not meet the demands of the stream it is trying to decode. Often decoders will produce poor results such as flickery video when there is not enough CPU power available.

The are several considerations which must be taken into account when determining which parts of a video are more important than others. The first thing to note is that almost any mapping of priorities onto video frames will not be universal. There are aspects of quality which may be more important to some users, while other aspects are of more importance to other users. Any system of prioritizing video will require some amount of choice in what is most important. There are, however, some factors which are inherent to the video format itself which must be adhered to. These hard dependencies dictate a partial ordering on data. One of these is the natural dependency of difference frames upon reference frames in MPEG. For difference frames to be decoded, any frames they refer to must have already been decoded and stored in memory. To ensure they are decoded first, they must have higher priority. Another hard dependency we must adhere to is that the header for a group of pictures must be decoded before any of the pictures in the group, so the frame containing that header must have the highest priority in the group.

\(^1\)However, audio decoding is relatively much less computationally demanding, so adapting audio quality would have a very small influence on overall resource usage.
Other factors which determine data priorities for video decoding are subjective. If we cannot decode an entire video window, do we give preference to having the frames which have been decoded being as evenly spaced out as possible, or would we prefer to have the highest number of frames decoded, regardless of their distribution? Is it more important to have high frame rates or more detailed frames?

In the current implementation of the PPD decoder, frame priorities are derived from the priorities given to the data during the network transmission of that data. The network priorities are useful because they adhere to the natural data dependencies inherent in the SPEG standard, as mentioned above. This is the simplest approach to take in giving priorities to the data.

3.3 Assigning priorities to frame data

Before priority based decoding was introduced to QStream, the video decoding module was only able to decode data in its natural offset order. It was not necessary to decode in any other order. Once a window worth of data had advanced to its decode stage and been re-ordered, it could simply be given the decoder sequentially. The re-ordering was simple, based only on an ADU's offset into the bitstream.

Prior to the development of PPD, the decoding process in the QStream player worked as follows. At the beginning of the decode stage, all the SDUs for the current adaptation window would be parsed into ADUs. Because the data was being given to the decoder in a fixed order, the ADUs merely had to be ordered into a single (per adaptation window) sequence and were subsequently submitted to be decoded. There was also some processing involved to collect all the ADUs from the current frame into one memory buffer which would be given to the decoder as a unit. This is due to the use of our scalable codec, which forces us to combine all the ADUs for a frame into a single memory buffer before submission to the decoder. Because there was only the need to deal with one frame at a time and only in sequential order, there was no need for any per-frame data structures, save for the memory buffer used to group one frame of ADUs together. The decoder was not designed to drop any data, as it was assumed that there would always be enough CPU availability to decode the entire video. So every ADU was given to the decoder sequentially and there was no jumping around within a window.

Before PPD decoding was implemented there was also no notion of data priorities. Once the ADUs had been extracted from their SDUs, the priorities were no longer relevant and were discarded. There was no need for priorities at the decode level because all ADUs would be decoded and there would be no data dropping at the decode stage. However, when decoding by priority, there is data dropping
and thus the need for priorities. Using the PPD decoder, the priorities can no longer be discarded. They are now used to determine the priorities of frames.

Since the basic unit of our priority based decoding is the frame, to assign priorities to our units of decoding we now have a need to collect our video data into frame objects. Each frame contains all its ADUs along with a priority and an offset. A frame’s priority is based on the priorities of its ADUs. However, ADUs did not have priorities associated with them before the PPD decoder was used. To give a priority to each ADU, a priority field was added to the ADU structure and the priority is inherited from the SDU from which the ADU was extracted. A frame’s priority is then calculated as the the highest priority of any of the ADUs it contains. The offset of a frame, representing its offset into the bitstream, is calculated as the lowest offset of any of its ADUs.

Within a given decode window, the frames are ordered by priority and then offset. In a given decode window, there may be multiple frames with the same priority in which case they are then ordered by offset. This is the order in which the frames are decoded. Now that we are grouping the ADUs into frames, we still pass ADUs one by one to the decode subsystem but now rather than retrieving them in offset order from a single queue, we retrieve them from the frames in priority order.

3.4 Priority based decoding in XviD

Prior to developing our priority based decoder, QStream used a modified version of the XviD codec for encoding and decoding video. The modifications had to do with using the SPEG scalable video format, which is used in the network stage for adapting to network bandwidth availability. Like a typical video decoder, the SPEG decoder expected data to be given to it in the order of transmission, also know as offset order. However, the PPD algorithm requires the video to be decoded in priority order rather than offset order. Implementing the PPD algorithm required that the SPEG decoder be modified.

3.4.1 Difference between offset and priority based decoders

To get an idea of the conceptual difference between the SPEG decoder and the PPD decoder, an analogy is helpful. One can think of the SPEG decoder as a black box. Encoded frames are its input, which are given to the decoder in decode order, which is the order in which they appear in the bitstream. The output is a sequence of decoded frames in display order, which is the order in which they will be displayed on the screen. Because of the relationship between display and decode order, it can deduce which frames are referenced by a difference frame. If a P-frame is input into the decoder, then the previous reference frame which was decoded is that frame’s reference frame. If a
B-frame enters the decoder, the two previous reference frames to be decoded are its references. The decoder must keep the appropriate reference frames in memory so that they'll be available when they are needed for the decoding of a difference frame.

Because of the fact that the PPD algorithm requires frames to be given to the decoder in a different order than the SPEG decoder, when the PPD decoder is given a difference frame to decode, it can no longer deduce which reference frames it references. It will often be the case that a difference frame given to the decoder will reference neither of the preceding 2 reference frames. More information is required in order for the decoder to know which reference frames are to be used for its decoding. When a difference frame is given to the decoder, the decoder must somehow be told which frames it references.

Because the priority of any reference frame is always higher than the priorities of the difference frames which reference it\(^2\), it will always be the case that a reference frame will be decoded before any difference frames which reference it. Thus when a difference frame is given to the decoder (along with pointers to its reference frames), it can be decoded immediately.

All attempts were made to keep the modifications to the decoder minimal. The decoder's API was kept as similar as possible to how it was before our modifications were made. Although it currently supports only one video format, QStream is not tied to always being limited to the single format. It is quite likely that in the future support for other codecs will be added. We tried to keep the interface to the decoder as simple as possible so that future decoders, each of which will have a different API, will be more easily ported to the system. Also, we tried to keep as much of the application logic as possible in the QStream application rather than the XviD decoder, so that such effort would not have to be reproduced in the future upon the inclusion of any new formats.

### 3.4.2 Buffering of frames

Another change that had to be made to QStream's decoder was to change how it handled video buffers. The SPEG decoder must use buffers to temporarily store reference frames (which can be either I-frames or P-frames). These reference frames must be kept in buffers after they've been decoded. Once they have been decoded, they still might be needed to act as reference frames in the decoding of subsequent difference frames (P-frames and B-frames). Since a P-frame is encoded as the difference between the frame it represents and the previous reference frame, the previous reference frame must be available in memory while the P-frame is being decoded so that it can be used in the calculation. A similar situation exists for B-frames, except that B-frames are encoded as a

\(^2\)Actually, priority can be equal, but offsets are used as a secondary key, and the offset will always place the reference frame first.
combination of differences of its two closest surrounding reference frames. Thus both of those frames must be available in memory when the B-frame is being decoded. Having enough buffer space to hold 2 frames of data is the absolute minimum required to decode a bitstream containing B-frames. It is also a logical number of buffers to limit a decoder to keeping track of, since no advantage can be had of keeping old frames around in memory after they've stopped being used as reference frames.

In the SPEG decoder, the rule for when to release one of its two buffers is simple - as soon as a new reference frame has been decoded, the older of the two buffers is released. However, when decoding in priority order, the decoder has no way of knowing when to release its buffers. Because the frames are being input in a different order, more than two buffers will need to be kept. This is information that is available within the application but not in the decoder.

It would be possible to add logic to the decoder so that it could manage many buffers instead of just two. If we were to do that, we would need a mechanism for the application to communicate to the decoder when a buffer could be released. However, this would require substantial changes to the decoder and we wanted to keep the changes to the decoder minimal. It would also make the interface to the decoder much more complicated.

Instead we chose to move all buffering out of the decoder and into the application. The decoder needs to use three buffers in decoding a frame: one buffer to decode data into, and two buffers with decoded frame data representing the last two decoded reference frames. The SPEG decoder manages the memory for those buffers itself. To implement our changes to the decoder, we moved the responsibility for managing the buffers used internally by the decoder into the application.

Before decoding, we pass the decoder references to the buffers we want it to use internally. We modified the decoder to use buffers passed to it as parameters rather than use its own internal buffers. Each time we invoke the decoder, we pass it one buffer to use as its internal buffer to decode into. We also pass it pointers to buffers representing the reference frames it relies on. If the frame to be decoded only relies on one frame or on none, we pass null values for those buffers.

Because we need to hold onto the decoded buffers from reference frames, we must keep the decoded frame data after the decoding of each reference frame. Since the application is supplying the decoder with its own memory to decode into, once the decoder has decoded a frame, we make sure to hold onto the buffer that was used internally to decode it. Because we provided the decoder with the memory to decode into, we still have access to it after the decoder returns. In a worst case scenario, the maximum number of buffers needed is equal to the sums of the number of I and P frames in the current window. The number of frames in the windows we are currently using is between 16 and 32. The number of those which are reference frames and need to be buffered will be considerably more...
than two.

3.4.3 Decoder state

We tried to limit ourselves to changing as little as possible of the decoder's normal work flow. We wanted to trick it into thinking it was being used normally. When decoding an arbitrary frame in priority order, if the state of the decoder and all its buffers were the same as if the decoder was decoding in offset order, then very little of the decoder's internal workings would have to be changed. Much of our changes to the decoder involved saving its state after the decoding of each frame and restoring state before the decoding of each frame.

There are several pieces of state which the decoder needs to have available before decoding a frame. If it's decoding a B-frame which relies on two reference frames, it needs access to the buffers holding the decoded forms of those two frames. There is also other state such as the amount of time since the last reference frame, which is needed for decoding a frame. Of all the state which the decoder keeps between invocations, we determined what state was vital to being able to decode properly, and made sure it was made available to the decoder before it decodes each frame. We make this state available using two techniques. Some state must be extracted from the decoder after decoding a frame to be made available for the decoding of future frames. Other state can be calculated prior to decoding a frame. Normally the computation of such state would take place in the decoder itself, but because we are decoding in priority order, we must move the calculations outside of the decoder and pass the results to it.

We added extra parameters to the main decode function in the decoder which allowed us to save and restore state between calls. The parameters were either input parameters or output parameters. The output parameters were used to extract state from the decoder after each decode operation, to be held onto by the application. The decoder was modified so that after each frame decoding, it would copy all the relevant variables to the output parameters passed to it by the application. After each call to the decoder, the application would save those values to possibly be used on subsequent invocations of the decoder.

The input parameters served the opposite purpose. The extra input corresponds to previous decoder state, which was extracted from the decoder after a previous invocation. The application passes that data to the decoder each time it is called. The decoder was modified so that each time it is called it would extract all the input parameters given to it and set its internal state accordingly. Because all of the decoder state represents the state of the decoder after decoding a particular frame, we chose to store the state in the frame objects.

\[\text{Motion vectors are scaled proportionally to the temporal distance to reference frames.}\]
3.4.4 Frame re-ordering

The SPEG decoder works on the principle that it will be given video data in decode order and will output frames in display order. The user of such a decoder can rely on the fact that the frames which will be output will be in the order in which they will be displayed and will not have to re-order the frames before displaying them. The re-ordering of the frames from decode order to display order will take place in the video decoder itself. This can make life easier for the user of the decoder.

The SPEG decoder implements something similar to a one-in-one-out protocol with frames. In most cases, when a frame is decoded, a frame will be output. Due to the fact that the order in which frames are given to the decoder is not the same as the order in which they are output, when a frame is decoded the frame which is output may not be the same frame. There will also be times when a frame is input to the decoder but there is no output. This usually happens at the beginning of a stream. This introduces a one frame latency, as the number of frames which have been output will be one less than the number of frames input. Since only one frame is typically output when one frame is input, at some point the decoder will need to be flushed. This is done by feeding it an empty input at the end of the stream, at which point the decoder will output the one frame it has yet to output.

As an example of how the re-ordering works, we'll discuss a sequence of frames which are to be displayed as I,B1,B2,P. The order which they will be given to the decoder is I,P,B1,B2 because the P-frame must be decoded before the frames which rely upon it. This is a typical mapping from decode order to display order. The first frame to be given to the decoder is the I-frame. Since it is also the first frame in sequence, it can be output immediately. Next, the P frame is given to the decoder. It is the second frame to be given to the decoder, but is meant to be displayed fourth. The decoder decodes it but does not output it immediately. It cannot be output until the two B-frames have been decoded and output, in order to ensure the frames are output in display order. At this point, a one frame latency has been introduced into the decoding cycle. This latency will typically continue for the duration of the video. Next the first B-frame is given to the decoder. Because both of its reference frames have been decoded already (and are thus stored in buffers), the B-frame can be decoded immediately. It cannot be output until the two B-frames have been decoded and output, in order to ensure the frames are output in display order. At this point, the P-frame is finally ready to be output. When the next frame is given to the decoder (which follows the last frame of the above sequence), the P-frame will be output.

In summary, with a standard decoder, when we feed the decoder the second reference frame in a video, we begin a cycle during which there will always be a one frame latency until the decoder is flushed. There will always be one frame from the video which has been decoded, but cannot be
output since it needs to be re-ordered first. This is done so that the user of the codec does not have
to re-order frames after they have been output from the codec. This gives us something similar to a
one frame in, one frame out model, but not quite that.

When we decode frames in priority order instead of decode order, we do not want the decoder
to re-order the frames that it outputs into display order. With offset order decoding there can be
at most a one frame latency between between the time the first frame is given to the decoder and
when the decoder produces the first frame of output. After that frame has been output, the decoder
will output a frame every time it is given one to decode. However, with priority order decoding the
latency could vary. We might need to provide several frames to the decoder before the first display
order frame has been decoded. This would happen in the case where in the decode window there are
several frames of higher priority than the first display order frame. Conversely, sometimes sending
a frame to decoder may result in many frames simultaneously being ready for display. However the
decoder API does not allow for multiple frames to be output at once. Moving frame re-ordering
outside of the decoder allows us to stick to the simple one frame in, one frame out model of
decoding, avoiding the need to change the decoder API to allow a variable number of frames to be output each
time a frame is given to it to be decoded. We judged that this approach would require the least
modifications to the decoder.

It is worth noting that XviD supports a mode called packed mode. In this mode, the model of one
frame in, one frame out is enforced by ensuring that the decoder always produces an output when
it is given a frame to decode. Because of re-ordering, there will be times when it is not possible to
output a frame, so in packed mode the decoder outputs a dummy frame with no data. At other times
the decoder will output two frames packed into a single buffer so that the decoder does not need to
be flushed. However, we found that packed mode was not suitable for the purpose of priority based
decoding.

3.5 Data dropping

We have already outlined 2 major steps in what is required for priority driven video decoding:
prioritizing the order that the data should be given to the decoder, and having the decoder be able
to decode data given to it in priority order. The final issue is the actual dropping of data when there
is not enough CPU to decode it all.

There are two scenarios that we can encounter. For a given decode window, there may be enough
CPU to decode the entire window in the time allotted to it. In that case, no data needs to be dropped.
The other case is that while decoding, we run out of time and cannot decode all the frames in the
window. In that case, all undecoded frames are dropped.

We use timers as a notification of deadlines. We maintain a pair of timers for each video that is being played. One timer is used to start decoding a decode window and another is used to stop it. Only one of these timers will be used, depending on whether dropping is required. When we begin to decode a decode window, we set the stop timer to go off when we should finish decoding that window. Whether that timer goes off depends on whether we have enough resources to decode the window in the allotted time. If the decoding of the window completes before the timer goes off, the stop timer is canceled and the start timer is initialized to go off when the decoding of the next window should begin. On the other hand, if the stop timer does go off, all undecoded frames in the current window are dropped and the decoding of the next window begins.

3.6 Summary

This chapter gave a high level view of what was necessary to implement priority order decoding in QStream. The streaming client needed to be modified, with the modifications being grouped into three main categories. The handling of video data and its submission to the decoder had to be modified so that the data could be submitted in priority order. The decoder was modified so that it could decode prioritized data, rather than data given to it in the canonical offset order. Finally, a mechanism was designed to drop data of low priority when the required computational resources were insufficient.
Chapter 4
Implementation

The previous chapter provided a high level overview of what was involved in implementing priority progress video decoding using the PPD algorithm. This chapter presents a concrete description of the implementation of the algorithm along with all the required scaffolding. The implementation using priority based decoding is also contrasted to the implementation of the system using the standard decoder to illustrate the differences along with the overhead required for decoding based on priority. It should be noted that although the comparison is to the standard version of QStream, most video players work in the same way.

The implementation of the PPD algorithm consisted of modifications to the client application in the QStream system. There were three main parts to the implementation. The first was to modify QStream to prioritize video data and send that data to the decoder in priority order rather than offset order. The second part involved modifying the SPEG decoder to accept data given to it in priority order rather than offset order. The last part of the implementation involved implementing a mechanism for dropping data during times when there is not enough processing power to decode an entire interval of video. This chapter discusses the specific details of the implementation and shows relevant code if necessary. All code in this section is pseudo-code to abstract away low level details which would be distracting.

4.1 Changes to the streaming client

The streaming client in QStream is event driven. The decoding of video is done through the scheduling of a number of events to carry out the work. The main entry to the decoding subsystem is the function \texttt{sp\_decode} which decodes audio and video. Handling of audio gets priority because it is important not to let the hardware audio buffer underflow as interruptions in audio are generally unacceptable to users. Since we are only concerned with video decoding, we will not mention audio decoding further.

4.1.1 New data structures required

Before discussing the decoding subsystem, we must first describe the new data structures to be used for priority decoding. Because we are now sending data to the decoder based on frame priorities, we
store the video ADUs in frame structures called QFrames. The following shows the format of the QFrame structure:

```c
struct QFrame {
    heap adus
    int priority
    enum type
    QFrame ref_frames[2]
    Time pts
    int ref_count

    IMAGE image
    MACROBLOCK* mbs
    int last_non_b_time
    int time_pp
    int time_bp
}
```

The `adus` field is a heap which holds the ADUs for the frame sorted by offset order. The `priority` field represents the priority of the `QFrame` which is a number between 0 and 15, with 15 being the highest priority. The `type` field tells what type of frame it is (I-frame, B-frame or P-frame). `ref_frames` is an array of pointers to two other QFrames. These are the reference frames upon which this frame relies. If the frame is not a difference frame or only relies on one frame, one or both of the pointers will be null. `pts` represents the presentation time stamp of the frame, which is the time relative to the start of the video that the frame should be displayed on screen. The `ref_count` field is used for garbage collection of the QFrame. The remaining fields are used to store decoder state. They will be explained more fully later. In brief, with the modifications made to the decoder, some its internal state had to be saved and restored between invocations and the QFrame objects are used to hold the state.

We also created a PPD structure to encapsulate the state of the decoder instance for a video decode window. Each decode window has its own associated PPD structure. When the SDUs are decoded into ADUs, QFrames are created to hold the ADUs and references to the QFrames are stored in the decode window’s PPD object. There will only be one decode window actively being decoded at a time. It may be the case that while a decode window is being decoded, the parsing of SDUs into ADUs has resulted in several other decode windows worth of ADUs to be created. In this case, their associated PPD objects are put on a queue to be decoded later. The following shows the structure of a PPD object.

```c
struct PPD {
    tree frames_by_offset
    tree ref_frames_by_offset
```
The PPD object contains three ordered collections of the frames in its decode window. `frames_by_priority` is a heap of frames sorted primarily by priority. Frames with the same priority are further sorted by offset. This heap is used to choose the next frame to be decoded when decoding in priority order. `ref_frames_by_offset` is a tree ordered by frame offset that holds references to all the I-frames and P-frames in the decode window. It is used to find which frames a difference frame relies on, since when decoding by priority, this must be specified to the decoder. `frames_by_offset` is a tree of frames ordered by offset. This is used for deciding which QFrame an ADU belongs to. `vid_start` and `vid_end` are time values which represent the times when the decoding of the PPD should begin and when it should end.

### 4.1.2 Finding reference frames

Each difference QFrame contains pointers to the reference QFrames it relies on. This is done for two reasons. The first is for garbage collection. Since C does not provide its own garbage collection mechanism, we must perform it ourselves. As for the QFrames, we need to know when to free them. We do so by keeping a reference count in each QFrame and freeing a QFrame when its reference count becomes zero. We may also end up dropping a QFrame without decoding it, in the case where there is not enough time to decode all the QFrames in its decode window. In that case each of the QFrames to be dropped will be dereferenced until their reference counts reach zero and will then be freed immediately.

In the case where we do not drop any data in the decoding phase, we make use of the reference counts. We want to make sure that we do not free a reference QFrame until all difference QFrames which rely upon it have been decoded, because the decoder needs that decoded data from the reference QFrames to decode the difference QFrame. On creation of a QFrame we set the QFrame's reference pointers to point to the up to two reference QFrames it relies on and we increment each of their reference counts. At the point of a QFrame's creation we are assured that any reference frames it relies on will have already been created because they will always be of higher priority. On the creation of a QFrame, we call the function `find_ref_frames` to set its pointers to any reference frames it relies on.
function find_ref_frames(frame)  
1     frame.ref_frames[0] = NULL 
2     frame.ref_frames[1] = NULL 
3     tree = frame.ppd.ref_frames_by_offset 
4     if frame.type == I_TYPE 
5         return 
6     frame.ref_frames[0] = 
7         lookup_previous_frame(tree, frame.offset) 
8     if frame.type == P_TYPE 
9         return 
10    frame.ref_frames[1] = 
11         lookup_previous_frame(tree, frame.ref_frames[0].offset)

This function is called by the QFrame constructor. It looks at the tree of reference frames contained in the frame's PPD object, `ppd.ref_frames_by_offset`. Because of the format of the SPEG bitstream, any frames that a difference frame references will immediately precede the difference frame in the bitstream. So when we want to find these frames, all that needs to be done is to traverse the tree of reference frames to find the immediately preceding frames. In the case of a B-frame, we need to find the two preceding reference frames and for a P-frame we need to find the one preceding. We do nothing for I-frames since they are not coded in terms of other frames.

4.1.3 Handling of ADU data

One of the more substantial changes to QStream was to how it handles ADUs. Before priority based decoding was implemented, ADUs were sent to the decoder in offset order. The ADUs were stored sequentially in the adaptation window and there was no need for a structure representing a frame of video. Before priority order decoding, the function `sp_win.decode_sdus` would loop over all the SDUs with the same timestamp (comprising all the SDUs in a map window), extract the ADUs from each SDU and would store the ADUs in a heap in the adaptation window, sorted by offset. With priority decoding, we now store the ADUs in QFrames. The function `decode_sdus_helper` is called by `sp_win.decode_sdus` for each ADU and is responsible for creating QFrames and using them to store the ADUs. Knowing what QFrames to create was facilitated by header information embedded in some of the ADUs in the form of start codes\(^1\). Each frame in the SPEG bitstream has a start code as its first group of bytes, followed by metadata about the frame. When the bitstream has been converted to ADUs, the first ADU of each frame will contain a start code. When storing the ADUs in QFrames, each ADU is checked to see if it contains a start code to determine if a new QFrame needs to be created. The following code shows the `decode_sdus_helper` method. It takes a reference to the PPD representing the current decode window and an ADU to be stored in a frame in that

\(^1\)All start codes in MPEG are 4 byte sequences that begin with the three byte sequence 0,0,1.
Chapter 4. Implementation

PPD.

function decode_sdus_helper(ppd, adu)
1  need_to_create_frame = false
2  frame = lookup_previous_frame(ppd.frames_by_offset, adu.offset)
3  if has_start_code(adu)
4    start_code = get_start_code(adu)
5    coding_type = get_coding_type(adu)
6    need_to_create_frame =
7      !frame ||
8      start_code in [GOP_START, B_TYPE, P_TYPE]
9  if need_to_create_frame
10     frame = create_qframe(ppd, adu, coding_type)
11    heap_insert(frame.adus, adu)

On line 1, the variable need_to_create_frame is used to represent whether or not a new QFrame needs to be created to hold the current ADU. On line 2, the frame is set to point to the QFrame into which the ADU should be inserted. If ppd.frames_by_offset is empty, frame will be set to NULL. Note, that if the ADU contains a picture start code, it will not necessarily be stored in frame. Lines 3-8 check if a new QFrame needs to be created for the current ADU. A new QFrame needs to be created if the current ADU has a start code and the start code represents a B-frame, P-frame or is the header for a group of pictures (GOP). Since I-frames always follow a GOP header, we decided to put the GOP header in the same QFrame as its following I-frame. Thus, when the current ADU contains an I-frame header, we do not create a QFrame, but instead put the ADU into the same QFrame as the preceding GOP header. Lines 9-10 create a QFrame if one needs to be created. Finally the current ADU is inserted into the heap of ADUs in the QFrame.

In addition to being responsible for dealing with the ADUs, sp_win_decode_sdus also schedules a PPD to be decoded at a later time. One can think of that as the entry to the decoding of the PPD. The following code illustrates this.

function sp_win_decode_sdus(PlaySession pps, AdaptationWindow win)
1  timestamp = min_sdu(win.sdus).timestamp
2  ppd = create_ppd(pps, timestamp)
3  for each sdu in the current map window
4    for each adu in sdu
5      decode_sdus_helper(ppd, adu)
6  if is_empty(win.sdus)
7    duration = win.vid_end - timestamp
8  else
9    duration = min_sdu(win.sdus).timestamp - timestamp
10   ppd.vid_end = duration + timestamp
11  ppd_start_decoding(ppd)
On line 1, `timestamp` is a reference to the timestamp of the current map window. This is taken from the first SDU in the map window, which is also the next ADU in the adaptation window\(^2\). On line 2 `ppd` is created, and it will hold all the ADUs for the current map window. On lines 3-5, all the ADUs in the current map window are extracted and passed to the helper method to be put in QFrames. On lines 6-9, `duration` is set to the length of the map window, which is the same as the duration of the decode window. If the adaptation window `win` has no more SDUs left, the duration is calculated as the difference in time between the end of the adaptation window and the beginning of the map window. Otherwise, the beginning of the next map window is taken from the next SDU in the heap of SDUs, and the duration is set to be the difference in start times of the two map windows. On line 10, the end time of the PPD is set. Finally, on line 11 a video decoding event is scheduled for `ppd` to begin its decoding in the future.

### 4.1.4 Sending data to the decoder

PPD decoding changed how ADUs were stored, so that now they are stored in QFrames. Submitting the ADUs to the decoder had to change accordingly. `sp.decode` is the main function in QStream responsible for sending video to the decoding subsystem. It starts the chain of event scheduling which results in video being decoded. Each time it is called, it will call `sp_win_decode_sdus` (described above) to create the next PPD and populate it, or will free the current adaptation window if no SDUs remain in it. The last thing that `sp_win_decode_sdus` does is to call `ppd_start_decoding` to schedule a timer to go off when the decoding of the newly created PPD is to begin.

`ppd_start_decoding` takes a PPD as a parameter and schedules a timer for the decoding of that PPD. When that timer goes off, the function `ppd_start_timeout` gets called. That function will schedule an expiry timer for the PPD to go off when the PPD is meant to be done decoding. The expiry of PPDs will be discussed shortly. The other thing done by `ppd_start_timeout` is to call `ppd_schedule_decode` to schedule the decode event.

`ppd_schedule_decode` is used to schedule decode events, each of which is responsible for a chain of events which will deal with retrieving one ADU and sending it to be decoded. In addition to scheduling the decode event, `ppd_schedule_decode` also has the job of setting the priority of that event. It sets the priority to be proportional to the priority of the next QFrame to be decoded. The reason that it does this is for situations when there are more than one video being played by the application at the same time. To achieve equal quality across videos, the video whose next QFrame has the highest priority should be decoded next. The event scheduled by `ppd_schedule_decode` to deal with each ADU will call the `handle_decode_event` function. This function is shown below.

---

\(^2\)Recall that adaptation windows are made of one or more map windows
function handle_decode_event(PlayVideo pv)
1     ppd = get_ppd(pv)
2     ppd_submit_adu(ppd)
3     if ppd.has_adus_left()
4         ppd_schedule_decode
5     else
6         cancel_timeout(pv.decode_timeout)
7     ppd_delete(ppd)
8     ppd = get_ppd(pv)
9     if ppd
10         ppd_start_decoding(ppd)

On lines 1-2 the next PPD object is retrieved and ppd_submit_adu is called to submit a single ADU further down the decoding pipeline. On lines 3-4, if there are still ADUs in the PPD, another decode event will be scheduled and the function returns. Otherwise, no ADUs remain in the PPD and the next PPD will be scheduled for decoding if possible. On line 6, the expiry timer for the current PPD is canceled since it is no longer needed. Next the current PPD is freed and ppd is set to be the next PPD in the queue (lines 7-8). Finally, if there was indeed a PPD on the queue, it is scheduled to be decoded.

ppd_submit_adu is responsible for sending ADUs further down the pipeline in the decoding sub-system. Which ADU is next depends on whether we are decoding in offset or priority order. Either way, we retrieve the next ADU to be submitted by first getting a reference to its enclosing QFrame, and removing and submitting the ADU with the lowest offset from that QFrame. The code for ppd_submit_adu is shown below.

function ppd_submit_adu(ppd)
    if decoding_by_priority
        frame = min_frame_by_priority(ppd)
    else
        frame = min_frame_by_offset(ppd)
    adu = remove_min_adu(frame)
    sp_video_submit_chunk(ppd, adu)

The variable decoding_by_priority is set as when the program is started to represent whether we are doing offset based decoding or priority based decoding. Accordingly, frame is retrieved from the PPD and set to be the next frame in either offset order or priority order. Next adu is set to be the ADU of minimal offset in frame and is removed. Finally, sp_video_submit_chunk sends the data from the frame to be decoded. sp_video_submit_chunk does not actually send each ADU to the decoder individually. Each time it is called it adds the data from the ADU to a buffer which is used to collect the data for a single frame. Only after that buffer has been filled with all the data for a frame, does it get submitted directly to the decoder as one unit.
4.2 PPD expiry and data dropping

We have hinted at the existence of an expiry timer for each decode window but have not yet explained it fully. The expiry timer is our mechanism for dropping data if we do not have enough resources. An expiry timer is set to go off when we reach the time when the current decode window should finish. If all the QFrames in the window had been decoded, the expiry timer would have been canceled by the `handle_decode_event` function.

When the expiry timer goes off, the function `ppd_expire_timeout` will be called. This function stops the decoding of the current window and cleans up after it. As mentioned earlier briefly, the data from the ADUs is collected in a buffer and the buffer is submitted to the decoder as a whole. This buffer may be partially full so it will be cleared. Next, the PPD object for the current decode window is deleted by calling the `ppd_delete` function. This will free any undecoded QFrames. Finally the expiry for the next PPD is scheduled.

4.3 Changes to the decoder

To understand the changes that were made to the decoder, it is vital to first explain the inner workings of the decoder before any changes were made to it. We have already mentioned the general way that an MPEG decoder works, but here we will be more specific and focus solely on the SPEG decoder.

4.3.1 How decoding worked before our changes

The public API of the SPEG decoder consists of a main function called `xvid_decore` along with one other function called `xvid_global`. The `xvid_decore` function is used for all three tasks of creating an instance of an XviD codec, decoding data and encoding data. `xvid_global` is a function for setting and retrieving global options of the codec. In order to keep the changes to the decoder minimal, we made sure to keep the API the same. Here is what the `xvid_decore` function looks like, along with the parameters it takes.

```c
int xvid_decore(void *handle, 
    int opt, 
    void *param1, 
    void *param2)
```

`handle` is a reference to the SPEG codec. This may be either an encoder or a decoder instance. `opt` represents what type of action we are performing with the `xvid_decore`. In the case of decoding video, we specify the constant `XVID_DEC_DECODE`. The meanings of the next two parameters depends on the context in which the function is used. When decoding video `param1` represents the input
and output parameters we are passing to the decoder. They are packed into a structure of type `xvid_dec_frame_t` in order to keep the interface of the function simple. Packing the parameters into a structure ensures that the `xvid_decode` function in future versions of the decoder that require more parameters will have the same signature. The parameter `param2` represents an `xvid_dec_stats_t` object which is used for collecting stats about the decoding process. All of our work involves decoding video so we will only concentrate on that functionality of `xvid_decode`.

### 4.3.2 Internal buffer management

When the decoder is invoked, it consumes the data from the bitstream passed to it in the `param1` parameter, which is encoded as an SPEG bitstream. The data is then decoded into a buffer internal to the decoder of type `IMAGE`, named `decoder.cur`. This decoded data is never seen by the user of the decoder and only lives inside the decoder. `decoder.cur` is one of a number of buffers used internally by the decoder.

After the bitstream has been decoded into the `decoder.cur` buffer, the decoder may output a frame. As mentioned in the previous chapter, due to re-ordering of data done in the decoder, the decoder will not always output a frame after it has decoded one. The re-ordering also makes it possible that the decoded frame that is output is not the same as the frame that was just decoded.

When the decoder does output a frame, it will output a frame being held in one of its internal buffers. Before outputting the frame, colourspace conversion is done to it with the result being written to `frame.output` which is a user supplied buffer of type `xvid_image_t`, which is passed in through the `param1` parameter. After that has been written, the decoder returns and the user of the decoder can retrieve the data from `frame.output` in order to be displayed. We will explain more about `param1` shortly.

It is important to note that while both the `decoder.cur` buffer and the data written to `frame.output` both represent decoded frames, the data they contain is not of the same format. `decoder.cur` contains data before colour conversion has been applied to it, and `frame.output` contains the data after colour conversion. Buffers internal to the decoder such as `decoder.cur` are of type `IMAGE`, while the `frame.output` buffer used as an output parameter of the decoder is of type `xvid_image_t`. Both `IMAGE` and `xvid_image_t` are defined in XviD, although `xvid_image_t` is the only one of the two which is part of the public interface to XviD. The user of the decoder does not need to know about the `IMAGE` type. All output from the decoder is done through the user-supplied `frame.output` buffer. The memory used for the internal buffers in the decoder is allocated and managed by the decoder. The `frame.output` buffer on the other hand, is allocated by the user of the decoder.

After a frame has been decoded into `decoder.cur`, if that frame represents a reference frame,
the decoder needs to keep it buffered for the decoding of future frames. Because the `decoder.cur` buffer is always used for decoding the current frame, once a reference frame has been decoded into `decoder.cur`, the decoder will move it into another internal buffer before returning, so that upon the next call to the decoder, that data will not be overwritten. The decoder keeps two internal buffers which represent the decoded versions of the last two reference frames that were decoded. When the decoder is invoked, `decoder.refn[0]` will hold the last decoded reference frame and `decoder.refn[1]` will hold the reference frame decoded previously. The data stored in each of these buffers will persist until another reference frame is decoded. Once a new reference frame has been decoded, `decoder.refn[1]` will be dropped since it is no longer needed, `decoder.refn[0]` will be copied to `decoder.refn[1]` and `decoder.cur` will be copied to `decoder.refn[0]`. This is done because there is only ever a need to keep copies of the last two decoded frames.

### 4.3.3 Other structures internal to the decoder

The decoder also keeps two arrays of `MACROBLOCK` structures internally. Each array represents one frame, and has one `MACROBLOCK` for each macroblock in the frame. Similarly to the decoder's handling of buffers for decoded frames, the decoder contains one array called `decoder.mbs` which represents the macroblocks for the current frame, and one called `decoder.last.mbs` which represents the macroblocks from the last reference frame. As with the internal `IMAGE` buffers, the memory for the `MACROBLOCK` arrays is managed by the decoder. In addition to these two structures, the decoder keeps other state, mostly in the form of simple values. These values represent properties of the decoding such as the amount of time since the last reference frame was decoded.

### 4.3.4 Re-ordering frames

One of the functions of the decoder before it was modified was to re-order frames from decode order to display order. Each time a frame was given to the decoder it would be decoded into the `decoder.cur` buffer. If a frame was to be output, a frame would be colour converted and written to the `frame.output` parameter passed to the decoder by the user. After that, the internal buffers in the decoder would be juggled if the frame that was just decoded was a reference frame. Finally the decoder would return.

Re-ordering frames by the decoder is facilitated by the decoder's choice of which frame to output. The frame to be output is often not the frame which has just been decoded. After decoding a frame, the decoder has a choice of outputting one of two frames: `decoder.cur` or `decoder.refn[0]`. Which frame to be output depends on what type of frame was just decoded. The first frame of the stream,

---

3In MPEG, a frame is defined as a 2D array of macroblocks.
which is always an I-frame, may be a special case. If the low.delay \(^4\) decoder option is specified, it would always be output immediately after being decoded, because that first frame is usually the only frame in the stream which has the same position in both offset and display orders. For all other reference frames, as soon as they are decoded, the frame to be output is the previous reference frame, held in the decoder.refn[0] buffer. This is due to the relationship between offset and display orders. B-frames would always be output as soon as they were decoded.

The role of outputting a frame belongs to the decoder.output function in the decoder, which is called after a frame has been decoded but before the decoder returns. Among its parameters, it takes a reference to the internal buffer which is to be output. Which of the internal frames is passed to decoder.output depends on the type of frame just decoded. For the first I-frame and all B-frames, decoder.curt is passed to it. In all other cases, decoder.refn[0] is passed. In addition to the internal buffer which is passed to decoder.output one of the two MACROBLOCK arrays is passed as well.

As mentioned in the previous chapter, we would like the decoder to always produce an output not re-order the frames for us. We have achieved this by modifying the decoder to always produce an output when given an input. Each time a frame has been decoded, we ensure that the decoder.output function will be called. Additionally, we ensure that the frame that is output is the one which was just decoded. Rather than passing one of decoder.curt and decoder.refn[0] to decoder.output, we only pass decoder.curt. Since this buffer always holds the decoded form of the frame just decoded, this ensures that the frame output by the decoder always corresponds to the encoded frame given to it. Correspondingly, we also always pass the decoder.mbs macroblock to decoder.output.

### 4.4 Overall functionality

The use of the xvid.decire function for video decoding has remained very similar after our modifications to the codec. We still call it once per frame to be decoded, passing all relevant parameters in its param1 parameter. The only difference is that the xvid.dec.frame.t structure has had extra fields added to it so that we can pass more information to the decoder. The main reason we need to pass more data to the decoder is that when we are decoding based on priority, the decoder is not able to deduce as much information about interframe dependencies and other state which it would otherwise be able to persist internally between invocations. It has to be passed that information as parameters instead. The structure of the xvid.dec.frame.t structure is outlined below.

\(^4\)We do not use low.delay in QStream.
The `version` field represents the version of XviD we are using. This is used to ensure compatibility between XviD and the application using it. `general` is a bitfield representing binary options we want to pass to XviD, such as post-processing options. `bitstream` is a reference to a bitstream object which holds the video data to be decoded. `length` represents the number of bits in the bitstream. `output` represents a chunk of memory into which the decoder will output the decoded frame. `brightness` represents the video's brightness and is not used by QStream.

When decoding in offset order, the decoder only needs the above fields to be supplied to it because much of the other important data can either be deduced or persists in the decoder between invocations. For example, when decoding a B-frame, the decoder needs to know what reference frames to use in its decoding. In the case of offset order decoding this can be deduced. However when decoding in priority order, the decoder needs to be told which reference frames the frame it is decoding relies on. When decoding in priority order, the decoder needs to be passed more information. In addition to the fields that were part of the `xvid_dec_frame_t` already, more fields had to be added.

In the modifications to the decoder, we moved state normally held by the decoder into the application. That way, we could save the decoder’s state after each invocation and restore it before each subsequent invocation. The fields added to the `xvid_dec_frame_t` structure which act as parameters to the decoder, all serve this purpose. They are stored in QFrames in the application, and the relationship between these fields and those in the QFrame can be seen by noting the similarity in the names of the fields. The fields are divided into those which act as input parameters, those which act as output parameters, and those acting as both. Below is a description of the structure after we added more fields to facilitate priority based decoding. The fields listed in the previous definition of the same structure are left out of this definition for brevity.

```c
struct xvid_dec_frame_t {
    IMAGE image_out
    IMAGE image_refs[2]
    MACROBLOCK* mbs
    MACROBLOCK* last_mbs
    int last_non_b_time
}
```
image_out is an output parameter. It is a reference to an IMAGE buffer which will be used to store the frame after it has been decoded but before it has been colour converted. For each QFrame that we are about to decode, we allocate an IMAGE buffer and store a reference to it in the QFrame. We pass a reference to that QFrame's IMAGE buffer to the decoder as the image_out parameter. Before the decoder decodes a frame, it changes its pointer to the decoder.cur internal buffer it would normally decode into to point to image_out. After the decoder returns, the application maintains a reference to the decoded buffer to be used in the future.

image.refs is an input parameter which holds buffers of the last two decoded reference frames (in offset order). Before the application calls the decoder, it sets these to point to the image fields of the two reference QFrames that precede it in offset order. These buffers will have previously been passed to the decoder as the image_out parameter, and thus will be filled with decoded data. The decoder needs those for the decoding of difference frames. Before it begins decoding, the decoder changes its internal pointers to the two most recently decoded reference frames, decoder.refn[0] and decoder.refn[1], to point to the two image.refs buffers.

mbs is an input/output parameter and last.mbs is an input parameter. Each represents an array of MACROBLOCK structures, with each array being associated with one particular frame. They are stored in their associated QFrame objects.

The last three fields are all input/output parameters which represents units of time in the decoder. These values are used to keep track of the distances between reference frames and B-frames and are used in decoding B-frames. Like all the other added fields in xvid_dec_frame.t, they are stored in QFrames. Before the decoder is called, they are set. The decoder sets internal variables of the same name to be equal to the values passed to it, and upon the decoder returning, the application saves these fields to their associated QFrame.

4.5 Summary

This chapter outlined the important implementation details of priority progress decoding. We described the new data structures which needed to be created and how they were used. We also discussed the event driven infrastructure used to handle the decoding of video and how timers are used. We also discussed the changes we made to the decoder along with the new parameters which had to be passed to it.
Chapter 5
Evaluation

In this chapter we present an evaluation of Priority Progress Decoding (PPD) based on our implementation. Our results are based on a series of experiments we have conducted using QStream. Our experiments consisted of running the client program to play videos and collecting data on several aspects of the program as it decoded video.

5.1 Experimental results

We have performed measurements by monitoring the behaviour of the system during video display. We performed a number of experiments on the system using two versions of the streaming client. One client was compiled with the PPD decoder and one was compiled without the PPD decoder and all the related infrastructure. In the case of the client with priority based decoding implemented, it was run in two modes: priority order decoding and offset order decoding. The reason we measured the performance of the client with priority decoding implemented but not used is to demonstrate that the overhead involved with the infrastructure that was created to support priority decoding (such as using QFrames and PPD objects) is minimal.

5.1.1 Problems with current implementation

It should be noted that these results are preliminary. As of the time of this writing, priority order decoding is working properly during the playing of single videos. However, in the case of playing multiple videos under CPU overload there are some kinks to be worked out. The videos play and data is dropped, however the dropping is not uniform across the videos. This may be due to a bug or it may be that our approach is wrong.

The possible problem with our approach is that it is possible for some decode windows to be starved when we are playing multiple videos. Each time a new decode window begins to be decoded, it will contain all its data, while the other windows will likely to only be partially full, since they may have started their decoding earlier. When the new window begins its decoding, suddenly it will have the highest priority for decoding because it will still contain its highest priority frames, whereas the other windows will have had their highest priority data decoded already. On a similar note, if most
of the videos start a new decode window at roughly the same time, they will all have high priority while the windows which are midway through their windows will contain lower priority data. Thus those windows could get starved.

It is not immediately clear that this starvation scenario is truly what is going on. This needs to first be investigated. If the problems we are encountering do turn out to be due to the aforementioned scenario, there are other approaches we could take to balancing the quality between videos. One would be to make sure that all the decode windows that exist at any given time are aligned on their start and end times. This way there would be no overlap and no videos would be starved. However, when streaming from different sources simultaneously, one cannot assume that the map windows (and thus the decode windows) will be able to be aligned. They could be of different lengths. A way to remedy that would be to have the streaming client make the decisions about how to group frames into decode windows, rather than having the decode windows correlate to map windows. Another possibility for fixing this is to increase the sizes of the decode windows but not have them align with each other. This could possibly fix the starvation problem, but would add decoding latency.

5.2 Overhead of using the PPD decoder

We ran some simple experiments to calculate the overhead of the PPD decoder over the XviD decoder. We played videos that were simple enough that they could be decoded fully and wouldn't overload the CPU. Thus each time we played a video all the frames would be decoded and no data dropping would occur.

We used the streaming client to play videos and measured the CPU time required using `/usr/bin/time`. We streamed the videos over our local network from an instance of the streaming server residing on a different computer. Because of the high bandwidth of our network, no data was dropped during network transmission and the videos were all of maximum quality. We ran two different versions of the client: one version which uses the PPD decoder, and one version which uses the original decoder. The client which uses the PPD decoder was also run in two different modes: decode order decoding and priority order decoding. For all three situations, we ran each of the videos three times and took the average of the results.

Figure 5.1 shows the results of our experiments using two videos called `incredibles.db` and `winged.db`. Each of the three rightmost columns represent one of the three scenarios we tested. All the numbers represent the CPU time needed to play the videos. They are the sum of the `user` and `system` times as reported by `time`. Of interest to us is the difference between the CPU times required for each video. For the `incredibles.db` video, there was 9.5% overhead for the PPD decoder in
decode order and 9.1% for the PPD decoder in priority order. For the winged.db video, there was 1.1% overhead for the PPD decoder in decode order and 1.3% overhead for the PPD decoder in priority order.

The winged.db video is approximately twice the resolution of the incredibles.db video and thus more of the computation spent playing it is spent doing the decoding. Since the largest part of the work in the player is done in the decoder, a video with a higher resolution will lead to a greater proportion of the total work being done in the decoder. Conversely, a video with a lower resolution will spend a greater proportion of its computations outside of the decoder. Since our work involves adding extra work outside of the decoder, we believe that there should be a larger overhead to do priority decoding on lower resolution videos, which is what we have seen. Also, although the overhead using the PPD decoder in both cases is not very large, we believe that with some tweaking, the overhead can be decreased.

<table>
<thead>
<tr>
<th>video name</th>
<th>SPEG codec</th>
<th>PPD decode order</th>
<th>PPD priority order</th>
</tr>
</thead>
<tbody>
<tr>
<td>incredibles.db</td>
<td>20.53</td>
<td>22.48</td>
<td>22.39</td>
</tr>
<tr>
<td>winged.db</td>
<td>144.8</td>
<td>146.40</td>
<td>146.68</td>
</tr>
</tbody>
</table>

Figure 5.1: CPU time requirements (seconds)

5.3 Range of adaptability

One thing of interest to us is the range of adaptability that can be had with our system. We'd like to know what is the difference between the amount of resources needed to play a video at maximum quality and minimum quality. By minimum quality, we mean only playing the data of the highest priority. We compared two setups. In one setup, we used an instrumented version of the player which would drop all video data that was not of the highest priority. In the other setup we used the priority based decoder to decode data in priority order. We did not test any of the offset order decoding setups as we were concerned with adaptability, which is only achieved by adapting through priority decoding.

We ran experiments on videos and measured the total runtimes of each using /usr/bin/time. Figure 5.2 shows our results. For the incredibles.db video decoding only the top priority data reduced the amount of CPU time used by 81.8%. For the winged.db video there was a reduction of 79.8%. It should be noted that in our current implementation, the range of adaptability is a factor of the video as streamed, since our decode windows are based on the map windows which are assigned by the streaming server. Thus we are limited in our range of adaptability. However, in future versions of QStream it is likely that there will be more flexibility in choosing the sizes of decode windows. If
decode windows are allowed to grow larger than the map windows used by the streaming server, a greater range of adaptability can be achieved.

<table>
<thead>
<tr>
<th>video name</th>
<th>Decoding all data</th>
<th>Only top priority data</th>
</tr>
</thead>
<tbody>
<tr>
<td>incredibles.db</td>
<td>22.39</td>
<td>4.08</td>
</tr>
<tr>
<td>winged.db</td>
<td>146.68</td>
<td>29.57</td>
</tr>
</tbody>
</table>

Figure 5.2: CPU time requirements (seconds)
Chapter 6

Conclusions and Future Work

In the first section of this chapter we describe the contributions of this work. In the next section we discuss possible future directions in which this work can be taken.

6.1 Conclusions

This thesis has presented a system for decoding video in the absence of sufficient CPU availability.

6.1.1 Motivating arguments

In this thesis, we argued that an adaptive approach to video decoding is necessary for a number of reasons. One reason is the increasing variability among both video devices and video content. Because of this variability many devices are unable to meet the computational needs required to meet the demand of decoding certain content. Another motivation for taking an adaptive approach is for playing multiple videos on the same system. The variability in the computational requirements for the videos results in an imbalance of video quality across videos. Allocating a proportional amount of the processing power to each of the videos cannot result in equal quality across the videos because the relative proportion of the computational resources needed by each video fluctuates over time.

6.1.2 Conceptual contributions

This thesis presented the concept of priority based video decoding. With this approach, video is decoded in priority order rather than the traditional offset order. In order to accommodate this, a video’s timeline is divided into intervals called decode windows. In each decode window, the data to be decoded is prioritized and then decoded in offset order. Although the video is decoded in priority order, it is re-ordered so that the results are displayed in the proper order. If not all the video in the window can be decoded, all undecoded video is dropped. Decoding video in such a manner is useful in situations where there are not enough computational resources to decode a video (or several videos) in real-time. Also, by using such an approach when displaying multiple videos on a single system, we can achieve balanced quality across videos.
6.1.3 Implementation

We have built a decoding module as a part of the QStream video streaming system. This module, called the PPD decoder, is responsible for decoding video bitstreams in priority order. This required both changes to the client application in QStream as well as modifications to the XviD decoder module used by it. One major element of the implementation was to modify the streaming client to prioritize data and submit it to the decoder in priority order. Another element was the modification of the decoder to be able to decode the data given to it in that order. The last element was to implement frame dropping.

6.1.4 Evaluation

We evaluated several aspects of the performance of the priority based decoder. We ran experiments by streaming videos over our local network from a server on a different host. We found the computational overhead of using the priority based decoder over the standard SPECK decoder to be between 1.1% and 9.5%. Although this is not a huge overhead, we believe that with some tweaking that it can be reduced significantly. We also examined the range of adaptability of the system and noted that with our videos the amount of CPU used to decode a video could be reduced by about 80% and still have video be shown at its minimal quality level. We also noted a current problem with balancing quality between multiple videos and listed some possible solutions.

6.2 Future Work

6.2.1 Decreasing latency

Because the order in which frames are decoded in priority order is different than the order in which they are displayed, the PPD decoder introduces a latency in displaying frames up to the duration of its decode window. Having such a latency poses no problem for non-interactive video applications such as streaming over the web. However, for use in interactive applications such as video conferencing, this latency is not ideal. Interactive applications require low latency in order to feel responsive.

As previously mentioned, the PPD decoder introduces a latency to the video decoding process which is bounded by the length of the video decoding window. The longer the window, the more latency it causes. At 24 frames per second (a common rate), this means that each frame of decode latency adds 42 milliseconds to the overall latency, which is quite significant as this latency itself is only part of the overall latency which we are trying to minimize. Other factors which contribute to the overall latency include time spent encoding the video on the server side, and time spent being transmitted over the network, as well as other smaller contributing factors. Thus if we are to have
a low overall latency, it is imperative to try to keep the latency introduced by the decoder to a minimum. Taking a cue from PPS, PPD could mitigate this in some cases by adapting the window sizes as the stream plays. This extension is left to future work.

6.2.2 Fast-forward and rewind

Our work makes the possibility of implementing fast-forward and rewind much easier. For fast-forward, the naive implementation would be to decode the whole video and end up using more CPU than would be used to play the video at normal speed. However, with our decoder we could decode the video in priority order, and only display video of sufficiently high priority. Rewind is a little more tricky. However, much of the infrastructure laid down for priority order decoding could be used for rewind. Priority decoding and rewind have the similarity that they both decode data in an order which is not the order normally used by a video decoder. Implementing rewind on top of our code would be made easier by the fact that the implementation of priority decoding has already decoupled the order frames are decoded from the order they are displayed. We also have implemented code to calculate dependencies between frames which can be used for rewind.

6.2.3 Incremental decoding

One limitation of the current PPD implementation is that it only has one way of adapting, which is through frame dropping. There are other ways to adapt including only partially decoding frames. This would be possible because SPEG is a scalable codec which means that it is possible to only partially decode a frame. This is how scalable codecs deal with data that has been dropped, but in the case where all the data has arrived but there is not enough CPU to decode it all, it is possible to drop partial frame data during the decode stage in order to spend less CPU cycles to decode each frame. This might require saving more decoder state outside of the decoder. Also, it is expected to have a narrower range of adaptation, since there is a high non-variable cost to decoding a frame whether it is of low quality or high quality. This could also be quite complex to implement.

6.2.4 Better priority mapping

As mentioned earlier, the ppd decoder assigns frame priorities based on the priorities assigned at the network level. We believe that this results in a good priority mapping, however it is not necessarily the best mapping. The priorities are assigned by the the priority mapper module in the server application and are based on both hard and soft dependencies. While the hard dependencies by definition constrain our mapping of priorities to frames, the soft dependencies are more amenable to change. One approach would be to ignore the priorities from the network level and create a priority
mapping module as part of the PPD decoder. This module would act in a similar way to the mapper in the server. It has yet to be seen whether this would result in better performance.

6.2.5 Priority based decoding using other video formats

Modifying other MPEG-based decoders to use the PPD algorithm should be straightforward. MPEG-1, MPEG-2 and MPEG-4 all work fundamentally the same way, with reference frames and difference frames. All 3 standards use the same system of I, B, and P frames. The only challenges to adapting our approach to another MPEG decoder lie in the specifics of that decoder’s implementation. However, a new MPEG standard called H.264 is gaining a lot of publicity because it achieves significantly better compression rates. H.264 has similarities to the aforementioned MPEG standards in that it has reference and difference frames, with difference frames encoded as combinations of reference frames. However, there are also many differences, including the fact that instead of a difference frame being limited to referencing at most 2 frames, they can reference up to 5. At a high level, the PPD approach should also work with H.264. However, H.264 is much more complex than MPEG-1/2/4 and porting PPD to an H.264 decoder could pose unforeseen challenges. H.264 would be a good candidate to use PPD since it requires much more computational resources than MPEG-1/2/4 and thus there is a greater proportion of current systems which cannot cope with the demands of decoding it. With H.264 requiring more processing power, and PPD being targeted to systems with insufficient processing power, H.264 could benefit from our approach. It should also be noted that a high quality open source implementation of H.264 called x264 is available [6].

6.2.6 Optimizations

As of this writing, the priority based decoder developed for this thesis has only been recently completed. While it works correctly, little effort has gone into optimizing it for efficiency. There are several ways which it can be improved. Although the PPD decoder’s performance is almost as good as the standard XviD decoder, tweaking it could conceivably bring the efficiency difference between the two down to insignificant levels.
Bibliography


