Automatic error detection and switching of redundant video signals, with focus on loop detection

Aleksander Zdunek

Computer Game Programming, bachelor's level
2018

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
Sammanfattning

Denna rapport beskriver arbete utfört med att implementera automatisk detektion av loopande följder av bildrutor i en videosignal. Den centrala algoritmen för igenkänning av loopar beskrivs. För att förbättra beräkningshastighet tillämpas hashning av bildrutor. Två videosignaler jämförs med avseende på deras innehåll av loopar, och växling av uppspelad signal sker baserat på utvärdering av signalernas kvalitet. Upprepade följder, skilt från loopande följder, diskuteras. Tankar för vidare utveckling av en mer heltäckande paketlösning för detektion av olika typer av fel i videosignaler presenteras kort.
Abstract

This report describes work done on implementing automatic detection of looping frame sequences in a video signal. The central loop detection algorithm is described. Hashing of video frames is used as a means of improving computational performance. Two video signals are compared with respect to containing loops, and switching of displayed stream is done based on evaluated stream qualities. Repeating sequences—distinct from looping sequences—are also discussed, as well as cursory thoughts for further work on implementing a more comprehensive error detection package for video signals.
# Contents

1 Introduction .................................................. 1
   1.1 Background .............................................. 2
   1.2 Specification ........................................... 2
   1.3 Direkt Link system ....................................... 3
   1.4 Social, Ethical and Environmental Considerations ....... 4

2 Loop detection ............................................... 5
   2.1 Strings .................................................. 5
      2.1.1 String compare ..................................... 6
      2.1.2 Substring enumeration ................................ 6
      2.1.3 Substring search .................................... 6
   2.2 History buffer .......................................... 7
      2.2.1 Circular buffer ...................................... 7
   2.3 Repeated substrings ..................................... 8
      2.3.1 Element stream and history buffer ................. 8
   2.4 Loops .................................................... 9
      2.4.1 Loop detection ...................................... 9
      2.4.2 Algorithm complexity ............................... 10
      2.4.3 Reporting loop size ................................. 10

3 Frame hashing ................................................ 12
   3.1 Performance timing ...................................... 12
   3.2 xxHash .................................................. 13
   3.3 Cryptographic hashing ................................... 13
   3.4 Perceptual hashing ...................................... 13

4 Stream switching ............................................. 15
   4.1 Stream quality classification ............................ 15
   4.2 Stream switching strategies ............................. 16

5 Result .......................................................... 17
   5.1 Implementation .......................................... 17
   5.2 Further work ............................................ 17
      5.2.1 Real world testing ................................. 18
      5.2.2 Effects of video compression ...................... 19
      5.2.3 Perceptual hashing ................................. 19
      5.2.4 Statistical quality analysis ....................... 19
      5.2.5 Audio ............................................... 19
5.2.6 Signal loss and data corruption ........................................ 20

6 Conclusions .................................................. 21
Acknowledgement ........................................... 22
Bibliography .................................................. 23
Chapter 1

Introduction

The industry standard for digital video signalling within a television studio or production pipeline, before broadcasting, is the serial digital interface (SDI)\(^4\). SDI transmits uncompressed digital video as a stream of digital bits. The original SDI standard limits data transfer rate to a maximum of 360 Mbit/s and is often now referred to as SD-SDI\(^3\) (for “Standard Definition”). Several other standards based the original SDI standard have been developed\(^3\) to facilitate transfer of higher definition video which requires higher data transfer rates; among these HD-SDI\(^5\) with a transfer rate of slightly less than 1.5 Gbit/s, and Dual Link HD-SDI and 3G-SDI\(^6\) with transfer rates of slightly less than 3 Gbit/s.

The digital image data carried over an SDI link is usually encoded using some variant of YUV coding\(^7\). YUV coding describes each pixel of an image with three colour components, usually of 10 bits per component. The Y component, called *luma*, describes the light intensity of the pixel, essentially making up the greyscale component of the image. The U and V components are called the *chroma* components and describe the colourisation of the pixel and image. In an end device like a television screen, or for display on a computer monitor, the YUV encoded colour data is transformed to RGB colour coding, but any processing or analysis of the image data within the video distribution pipeline is usually done on the YUV coded signal. YUV encoding has become the standard video image encoding in television networks due to historical reasons from the early days of colour television\(^7\): Transmitting chroma component analog signals using the analog luma signal as a carrier allowed older black-and-white television sets to receive a colour picture signal as if it was a black-and-white picture signal. The positions of the YUV components within the bitstream, relative to the video frame delineations, translates to pixel positions in the displayed image. YUV encoding also allows for some reduction in image data size by subsampling chroma data in lower image resolution than luma data, taking advantage of the propensity of the human eye\(^8\) for being more sensitive to variations in light and dark than to variations in colour.
1.1 Background

Intinor\textsuperscript{1} is a company in Umeå, Sweden, that develops and provides distribution solutions for digital television and video streaming. Their Direkt family of products—Direkt Link, Direkt Receiver and Direkt Router—are built to handle encoding, decoding and transmission of digital video over the Internet. On their wishlist of features to add to their products is the functionality to detect a faulty input video signal and automatically switch over to a backup signal. Fault scenarios include:

- total loss of signal
- frozen image
- repeating loop of a few image frames
- blacked out image (or other colour)
- no sound
- crackling/popping sound
- electrical/connectivity issues causing signal CRC errors

**Total loss of signal**: Total loss of signal is characterized by an input channel not receiving any data. Detection of total signal loss is already present in Intinor’s systems.

**Frozen image / repeating loop**: Every video frame in the stream being the same results in a frozen video image. Similarly, a repeating loop of a few image frames results in the visual equivalent of an old vinyl record hangup. A frozen image is a special case of a repeating loop.

**Blacked out image**: A zeroed frame data stream results in results in a blacked out image when using RBG color representation. With YUV color encoding a zeroed data stream would typically result in a uniformly green video image. Similar to a frozen image, blacked out video is also a special case of a repeating frame loop.

**Sound**: Video is usually accompanied by audio. Problems with missing or distorted sound negatively impacts the video stream viewing experience.

**CRC errors**: Connectivity issues can cause signal data to be corrupted in transmission. Data corruption is detected through cyclic redundancy checks of checksums for each frame line, attached to frames in transmission. Integrity of audio data is not protected by checksums.

Including automatic signal monitoring and switch-over functionality in Intinor’s products would alleviate the end user from manual monitoring duties and would be an additional selling point for the products.

1.2 Specification

This thesis project focuses on loop detection of video frames, and implementing a proof of concept implementation that will run on Intinor’s Direkt Link hardware. Since frozen and blacked out image are both special cases of repeating frames,
focusing the work on detecting repeating frames covers a significant portion of the listed fault scenarios.

The project will hopefully be able to serve as a solid starting point for a more complete implementation of an automatic video signal monitoring system.

1.3 Direkt Link system

Organisation of relevant subsystems of a Direkt Link is illustrated in Figure 1. It is within the context of this system that implementation work is carried out.

![Figure 1: System overview for a Direkt Link](image)

SDI 1 video signals from external video playout systems are received on video capture interfaces, and buffered in circular frame buffers. Frame buffers are read by the videomixer subsystem which does some video processing of the signals. The resulting processed video stream is sent to an output hardware interface for distribution.
1.4 Social, Ethical and Environmental Considerations

Throughout the project I have been mindful of rising to the trust given to me by Intinor letting me have access to their office environment, system libraries and source code. Letting an unknown person into such an environment opens the risk of product source code being stolen, or other sensitive business information being publicly disclosed due to either malice or carelessness. Care needs to be taken not to abuse such trust. These concerns extend beyond completion of the work.

Even schematically outlining the organisation of a system, like done in this report, or revealing parts of a private library API can potentially tell a malicious third party something significant about a system and thus open up for potential security risks. To what level such descriptions of a system is acceptable needs to be determined in consultation with the responsible party for the system.

Conversations have taken place, and agreements made, with regards to shared ownership of the final results of the thesis work.
Chapter 2

Loop detection

This chapter documents the work done on detecting repeating or looping sequences of video frames in a stream. A distinction is made between loops and repeats.

Video frames of incoming or outgoing video streams are often stored in a short circular frame buffers, either on a hardware level or a software level. A circular buffer is essentially a fixed size array where buffer elements are written sequentially. When the buffer is full the buffer circles back to the oldest element in the buffer which is overwritten by the next incoming element. The frames of a video stream are accessed through the frame buffer. Typical lengths of a frame buffer are 4 or 8 frames. Problems in video transmission may cause a frame buffer on either the transmitting or receiving side to not get timely input, which results in frames already stored in the circular frame buffer to be read again, resulting in a playout that appears jittery or frozen. This is an observed failure mode both in Intimor's own equipment as well as in equipment from other manufacturers.

For the purpose of detecting these stream errors of repeating video frames, we abstract the video stream into a sequence of abstract elements. The problem then becomes a problem of detecting repeating substrings.

Analysing a sequences of elements in a stream requires that the elements are stored in a buffer representing the element history of the stream. Since it is neither efficient or of interest to analyze the complete history of a stream we limit the problem further to only searching the immediate history of a few frames for repeating subsequences.

2.1 Strings

A string is a sequence of elements. Usually the elements are assumed to be characters, but for the purpose of this text this is not necessary. In fact the first implementation of the concepts in this chapter was made to work on strings of characters. Only after this had been shown to work were the same algorithms applied in an implementation that worked on video stream. A substring is a contiguous subsequence of elements from a string, meaning that adjacent elements of a substring are also adjacent elements of the original string. Detecting repeating sequences of video frames within a stream can thus be abstracted into
finding repeated substrings within a string. To do this we shall need to be able
to compare strings for equality and to be able to list the substrings of a string
for comparison.

\section*{2.1.1 String compare}

Comparing strings for equality is done by iterating over the elements of one
string and comparing each element to the element at corresponding position in
the other string. If elements at corresponding positions are not equal then the
strings differ for at least one element and the strings are not equal. Otherwise,
if the elements of the strings are equal for each position the strings are equal.
This assumes that the comparing strings are of equal length. The algorithm for
comparing strings is shown in Algorithm 1.

\begin{algorithm}
\caption{String compare}
\begin{algorithmic}
\For {all position $p$ in string1}
\If {string1[$p$] \neq string2[$p$]}
\State return not equal \Comment{terminate algorithm early}
\EndIf
\EndFor
\State return equal
\end{algorithmic}
\end{algorithm}

\section*{2.1.2 Substring enumeration}

Substrings can be specified by a pair of indices $(i, j)$ where $i$ and $j$ are the posi-
tions within the original string of the first and last elements of the substring, respectively. Therefore all substrings of a finite string of length $n$ can be enu-
merated by generating every pair $(i, j)$ where $0 \leq i \leq j < n$. This can be done
with nested loops as shown in Algorithm 2.

\begin{algorithm}
\caption{Enumerate substrings}
\begin{algorithmic}
\For {$i = 0$ to $n - 1$}
\For {$j = i$ to $n - 1$}
\State $(i, j)$
\EndFor
\EndFor
\end{algorithmic}
\end{algorithm}

The length $l$ of a substring given by an index pair $(i, j)$ is $l = j - i + 1$. Therefore all substrings with a given length $l$ can be given by a single index $i, 0 \leq i < n - l + 1$, which is shown in Algorithm 3 for completeness sake.

\begin{algorithm}
\caption{Enumerate substrings of given length}
\begin{algorithmic}
\For {$i = 0$ to $n - l$}
\State $i$
\EndFor
\end{algorithmic}
\end{algorithm}

\section*{2.1.3 Substring search}

To determine if a pattern string appears as a substring in another string, enumerate
substrings of length equal to pattern string length according to Algorithm
3, and compare each substring to the pattern according to Algorithm 1. This is
summarised in Algorithm 4.
Algorithm 4 Detect occurrence of substring within string

for all substrings s of length equal to pattern length in string
if s equals pattern
return pattern found
return pattern not found

2.2 History buffer

In the context of video transmission the string elements represent video frames. A video transmission stream can then be seen as an ever growing string. With typical video streams running for hours or even years, at 25 or more frames per second, the string of video frames representing the entire history of the transmission stream quickly grows to be extremely long. The string analysis algorithms presented in this chapter are very simple brute force algorithms. They are easy to understand and to implement, but their asymptotic operation complexity makes them entirely unsuitable for processing long strings. Several improved string search algorithms suitable for processing long strings are widely known and asymptotically faster algorithms for detecting repeating substrings are also likely to exist. Fortunately, the type of video transmission errors we aim to detect consist of short repeated or looping substrings within close temporal (or positional, as it were) proximity. Therefore processing the entire stream history is unnecessary and undesirable. Instead we limit analysis to a history buffer containing only the latest few elements appearing in the stream. To achieve this the history buffer is a first in first out (FIFO) buffer where the latest video frame (or more generally string element) is pushed into one end of the buffer while the oldest element in the buffer is removed and forgotten. Detection of repeating substrings is thus limited the string stored in the stream history buffer.

The longest possible substring that can be repeated in a finite string is half the length of the string. The size of the history buffer must therefore be twice the length of the longest desirable detectable substring repeat. Typical use case scenarios in streaming video applications involve repeated substrings of about 8 frames or less, which would dictate a typical size for the stream history buffer to be 16 elements.

2.2.1 Circular buffer

The stream history buffer is implemented as circular buffer. A circular buffer is a contiguous sequence of memory, and two markers denoting the start and end positions of the data stored in the buffer. The position markers follow a modular arithmetic on the capacity of the buffer, allowing the stored data to be split up into two parts that may stored out of order in the physical memory. Subsequent elements of the stored data sequence can be stored before earlier elements in physical memory. In other words, a stored data sequence can begin somewhere in the middle of the allocated memory and then wrap around. This allows for efficient insertion and deletion of elements at the the beginning and end of buffered sequences, while keeping sequential access of buffered data intact using a modular arithmetic iterator. This makes the circular buffer an excellent data structure for buffering data streams.
2.3 Repeated substrings

A repeating substring is a substring that appears twice or more within a string or a stream of elements. Depending on application it may be relevant to specify whether or not overlapping substrings are disallowed or counted as repeated substrings. To illustrate, consider the string ABABAB. The substring AB is repeated within the original string. The substring ABAB is repeated within the original string if overlapping substrings are considered but not if they are disallowed. As a counterpoint to overlapping substrings, substrings do not need to appear contiguously in a string. For instance the substring AB is repeated within the string ABCAB.

To detect if a substring \((i, j)\) is repeated within a string \((0, n - 1)\) where \(n\) is the length of the original string, the substring search algorithm (Algorithm 4) can be adapted to count the number of times the substring appears in the string, instead of terminating at the first substring match. Because a substring will always match with itself at least once, the substring appears multiple times in a string if the string search count returns 2 or more. This count does however include overlapping substrings.

Slightly more efficiently a substring repeat can be detected by partitioning the original string into a preceding and a subsequent string to the pattern substring, show in Algorithm 5.

Algorithm 5 Detect repeated substring without overlapping

```
search for pattern \((i, j)\) within the substring \((0, i - 1)\)
if pattern found
   return pattern found
search for pattern \((i, j)\) within the substring \((j + 1, n - 1)\)
return search result
```

2.3.1 Element stream and history buffer

Detection of repeated substrings within an element stream is done by analysing a limited buffer of stream elements, containing the immediate history of the stream. Algorithm 6 shows this buffering of the stream, and searches the buffer for repeating substrings by using enumerated substrings (Algorithm 2) as pattern inputs to Algorithm 5. Limited constant size of the history buffer means that repeated substrings that are too long or too far apart to not fit completely within the history buffer are not detected.

Algorithm 6 Detect any repeated substrings

```
for each element in stream
   if history is full
      history.pop()
      history.push(latest stream element)
   for each substring \(s\) in history
      search history for repeats of \(s\)
```

One observation to make about the history buffer is that because of continu-
ing input from the element stream, substrings move in position from one end of the buffer to the other end, and every substring will at some point be located at the other end of the buffer. This means that it is possible to simplify searching of repeating substrings in the history buffer by only searching the partition following the pattern substring, because earlier repeats will have been detected at an earlier point of time in the element stream. This in turn means that it is not necessary to enumerate strings that are longer than the resulting subsequent history partition.

One avenue for improvement of the string repeat detection, that we have not explored is the observation that iterating over pattern substrings combined with substrings traversing the history buffer leads to many superfluous comparison operations that have already been performed earlier. This aspect is taken into consideration for the simplified detection case of loops, detailed in the next section.

Something else that has not been taken into consideration so far is the potential ability to define a maximum allowable gap between repeated substrings. It is conceivable that for some real world scenarios it is of interest to not detect repeated substrings that are sufficiently far apart but short enough to both fit within the history buffer.

Ultimately, repeat detection was decided to be an unnecessarily general solution, based on intuition about what kind of stream errors would be likely to appear in real world applications. Instead stream error detection was focused and simplified to detect looping streams.

2.4 Loops

For the purpose of this work we make a distinction between a looping and a repeating substring. A repeating substring is as a substring that appears more than once in a superstring*. A looping substring is a repeating substring with the additional requirement that there exist no other substring between two occurrences of the substring. For example the substring AB is looping and repeating in the string ABABC but only repeating in the string ABCAB.

Following the proof of concept implementation of repeat detection on character strings, we decided to focus development on loop detection. It would seem likely that in the majority of error cases involving repeating video frames in a stream, the repeating sequences are indeed loops. Constraining repeat detection to loop detection simplifies implementation, should improve computation efficiency, and potentially makes it easier for the end user to understand proper usage and configuration of the feature in a product.

2.4.1 Loop detection

Determining if any given substring is looping is simply a case of comparing the substring to the immediately following substring of the same length. For substring \((i, j)\), compare to substring \((j + 1, 2j - i + 1)\) for equality.

To determine if a string contains any loops we need only enumerate all substrings of the first half of the string. Because substrings traverse through the history buffer as described in section 2.3.1, every substring will at some

*A superstring is a string that contains a given substring.
point in the stream appear at the front of the history buffer. Therefore to detect loops in an element stream, for each update of the history buffer we only need to enumerate the substrings at the front of the buffer, of length up to half the buffer size. Algorithm 7 uses string comparison (Algorithm 1) to detect looped substrings on a continually updated history buffer.

**Algorithm 7** Loop detection in a stream

```plaintext
for each element in stream
    if history is full
        history.pop()
        history.push(latest stream element)
    let n be size of history
    for i = 0 to (n - 1)/2
        if substrings (0, i) and (i + 1, 2i + 1) are equal
            report loop detected. break
```

The push and pop operations on a FIFO buffer can be conceptualised and implemented in two ways relative to the indexing. If we say that index 0 is the front of the buffer then the push operation can insert an element at either the front or the back of the buffer. The pop operation operates on the opposite end of push. Depending of how the history buffer is conceptualized Algorithm 7 technically analyses reversed substring. This does not however have any significance for determining if a substring is looping.

### 2.4.2 Algorithm complexity

For each element in the input stream the loop detection performs at most \( \frac{n+1}{2} \) substring comparisons, where \( n \) is the size of the history buffer. Fewer substring comparisons may be required when a loop is detected and iteration over substrings can terminate early. Each substring comparison requires a number of element comparisons equaling at most the length of the substring. The substring lengths increase by 1 for each substring comparison, forming an arithmetic progression. The number of element comparisons per iteration of the loop detection forms an arithmetic sum

\[
\sum_{j=1}^{m} j = \frac{m(m+1)}{2},
\]

where \( m = \frac{n+1}{2} \). Thus the maximum number of frame comparisons per frame in a video stream is

\[
\frac{(n+1)(n+3)}{8} = \frac{n^2 + 4n + 3}{8}.
\]

In the worst case the complexity of Algorithm 7 is quadratic on the size of the stream history buffer. In the best case when the stream contains no loops, the complexity is linear on the size of the history buffer.

### 2.4.3 Reporting loop size

Since a looping string must appear twice contiguously in a stream before a loop can be detected, the frame count of the loop at first detection is twice
the length of the looping substring. Any immediately subsequent positive loop matches add 1 to the loop frame count. Let $n$ be the number of contiguously detected loops and $l$ be the length of the looping substring, then the total frame count of a looping section of the stream is $2l + n - 1$. The looping frame count is reset when a loop is no longer detected.
Chapter 3

Frame hashing

Hashing is the process of reducing a chunk of data into a smaller piece of data meant to represent the original data. The smaller piece of data is called a hash, or sometimes a digest. A function that generates a hash from a chunk of data is called a hash function. Informally a hashing algorithm is also sometimes referred to as a hash. Hashing should not be confused with data compression. A hash is intended to function as a type of identifier, or a fingerprint, of some data and it is generally not possible to recreate the original data from a generated hash. Hashes are usually much smaller than the data they were generated from and are not unique, meaning that several distinct sets of data can generate the same hash. When the same hash is generated from two different sets of data this is called a hash collision. Part of the difficulty with designing a good hash function is to reduce or minimize the probability of hash collisions in a given application.

Full HD video with resolution $1920 \times 1080$ pixels per frame with 2 bytes per pixel at a typical chroma subsampling level comes to almost 4 megabytes per video frame. Instead of storing the full video frame in the history buffer and comparing the entire frame data during the element comparison of the loop detection, only a hash of each frame is stored in the history buffer. Since an MD5 hash implementation already existed in the Intinor standard library, this was the most convenient first choice for a hashing algorithm.

3.1 Performance timing

To be practically useful the implementation of the loop detection must process video in real time. This means that hashing and comparing frames must not take more time than the time between two consecutive frames, defined by the frame rate of the video stream. Since the loop detection is intended to analyse redundant streams, hashing and frame comparison on one stream must in fact not consume more than half the time between consecutive frames, to allow processing time for analysis of both streams. The loop detection must also leave processing time on the hardware for other tasks considered more principal to the Intinor product.

Performance of the implementation is estimated by measuring the time it takes to process an archive of video frames from a RAM disk and comparing
that to the time the same amount of frames would play out at, at the given frame rate of the archive. If the implementation is able to process a frame archive faster than the nominal time it should also be able to process a video stream in real time, on comparable hardware.

By inserting timing markers around the frame hashing procedure call in the test implementation, it is possible to sum the time required to hash the video frames and compare this to the total archive processing time. In the case of the MD5 hash implementation it turned out that the processing requirements of the loop detection itself, on a history buffer with size 16 elements, is practically insignificant compared to the time spent hashing frames. Thus, any performance optimisations must be looked for in the hashing.

### 3.2 xxHash

In an effort to improve hashing performance an alternate open source implementation of MD5 hashing\(^{12}\) was tried. This however only produced a marginal speed improvement.

Instead another hashing algorithm, named xxHash\(^{13}\), was found. xxHash claims to be "an extremely fast non-cryptographic hash algorithm, working at speeds close to RAM limits". Timing tests showed that the open source implementation of xxHash performed almost eight times faster than the baseline MD5 implementation. xxHash did also not appear to alter the correctness of the loop detection algorithm. xxHash therefore seems an excellent choice for hashing video frames for loop detection.

### 3.3 Cryptographic hashing

Hash functions in cryptographic applications need to have a couple of properties in order to be cryptographically secure. It should be difficult to find a data set that hashes to a given hash, and it should be difficult to find two data sets that hash to the same hash\(^{14}\). Hash functions intended for cryptographic applications are called cryptographic hashes. MD5 was designed to be cryptographically secure\(^{11}\) but has since been broken\(^{15}\). Regardless, MD5 is still used in non-cryptographic applications.

xxHash is a non-cryptographic hashing algorithm\(^{13}\). Non-cryptographic hashes forgo the requirements of cryptographic hashes, and can therefore presumably be made faster.

### 3.4 Perceptual hashing

Hashes are usually arbitrary mappings from data sets, meaning that one or more data sets map to a hash but there is no semantic information that can drawn from a hash about the generating data. For cryptographic applications this is a necessity and for non-cryptographic applications this can be an effect of achieving good distribution of hashes. In contrast to this is the concept of perceptual hashing\(^{16}\). The idea with perceptual hashing is that, unlike with cryptographic hashing, small differences in input data results in small differences in the hashes. This makes it possible to compare two hashes for similarity.
instead of only equality, and have the comparison say something meaningful about the originating data. The exact meaning behind small differences is left unspecified and is dependent what the hashed data is meant to represent (such as a picture, a text or an arbitrary binary file for example), and the context of usage.

A limitation of the loop detection implementation of this thesis work is that it only compares frame hashes for equality. This means that two frames are only detected as equal if the are pixel perfect identical (or in the unlikely case of a hash collision). For instance if two consecutive frames are identical but for one pixel a human viewer would perceive the frames as equal but they would not be identified as looping by the loop detection algorithm. This is potentially a problem since it is conceivable that a stream may contain frames that are qualitatively identical not pixel perfect. In such cases the loop detection will fail to detect an indeed faulty stream. Particularly concerning is the effect video compression has on image quality. Video compression introduces compression artefacts which might negate pixel perfect equality of perceptually identical frames. If a video stream is compressed somewhere down the line from where the loop error is produced, perceptually identical frames might not produce identical frame hashes. Using perceptual frame hashing would likely solve this problem. Frames that are practically equal should then generate similar hashes. Instead of comparing the hash values of two frames for strict equality the frames can be defined as essentially equal if the difference between the hashes is below some defined threshold.

The problem with using perceptual hashing is performance concerns. The image processing required for perceptual hashing of video frames is likely to be costly in terms of processing time. Hashing of frames is by far the most costly part of the loop detection. If hashing is too slow it will be unsuitable for processing video streams in realtime. Use of perceptual hashing would also increase the probability of false positives. Sections of intentional still images a video production would be more likely to be detected as looping.
Chapter 4

Stream switching

The intention with being able to automatically detect loops, and faults in general, on a video stream is to be able to automatically switch play out to a alternate redundant stream if the primary stream goes bad. If a looping sequence of frames is detected in a video stream, the immediate response could be to switch play out to an alternate stream at first sign of trouble. This might however not necessarily be the most desirable course of action. A short temporary hiccup in the stream might not need to trigger a switch of streams. Unless play out of a stream is delayed by a number of frames equal to the size of the history buffer, in anticipation of the results of the stream analysis, the loop detection will be performed after the fact. An audience will see the same looping sequence of frames as the loop detection algorithm. If this fault quickly resolves itself, switching to an alternate stream may be undesired. The quality of the alternate stream must also be taken into account. There is no point in switching streams if both streams exhibit faults.

In the implementation of this work stream switching is done by internally setting a variable marking the active stream and calling an external script that edits the configuration of the separate video mixer program running on an Inter- nor Direkt Link.

4.1 Stream quality classification

To decide if a switch of active streams should take place, both redundant streams must be compared to each other in some way. In this thesis work we choose to do this by classifying the quality of each stream into broad categories of good, bad and dubitable, depending on recent prevalence of loops. To avoid switching active stream at first detected loop we introduce a user configurable grace time. A stream is qualified as dubitable when a loop is detected. If the stream has been continuously looping for the duration the grace time, the stream is qualified as bad. Likewise, if a bad stream stops exhibiting loops the stream is again reclassified as dubitable and then as good if no loops are detected for the duration of the grace time.

Information about detected loops is recorded for each stream. The record contains the latest detected state (either looping or non-looping) of the stream, the stream position of the latest state change, and the run length of the current
state. Stream position is given by a stream frame index and run length is given as a frame count. From this recorded information a current quality classification of the stream is derived. This presumes that loop detection is run, and the stream state information is updated, for each frame in a video stream.

4.2 Stream switching strategies

Switching between two streams can be done according to two different prioritisations. Preference can be given either to a designated primary stream or to the currently active stream. The latter strategy essentially values both streams equally high, while the former strategy gives priority to the primary stream.

When prioritising the primary stream, the primary stream is always made active if it is classified as good. Switching of active stream from the primary to the secondary stream occurs only if the primary stream is bad and the secondary stream is good. If the secondary stream is already active and the primary stream is not good, no stream switching occurs.

When preferring the active stream a switch occurs if the active stream is bad and the alternative stream is good.

In both strategies a switch of active stream will never occur to a dubitable stream. A switch from a dubitable stream can only occur in the case of strategic preference being given to the primary stream and the secondary stream being the active stream. The quality classification method is simple but could also be described as naïve. Not being able to switch away from a dubitable stream is potentially a problem since it is conceivable that in a more complex fault scenario a stream may experience looping subsequences interleaved with non-looping subsequences. In such cases when a stream exhibit looping behaviour but never loops continuously for an entire duration of the grace time the stream will not be classified as bad and will therefore not trigger a switch despite that a switching of streams may in fact be desirable in such a circumstance.

Since video streams are compared to each other only through their quality classifications, and the classifications are evaluated solely on the merits of each stream respectively, the streams do not in fact need to represent redundancy. An alternate stream may contain entirely different video. The contents of each stream is up to the discretion of the user.
Chapter 5

Result

The result of the thesis work is a proof-of-concept implementation of loop detection and automatic switching between video streams, that runs on an Intinor Direkt Link. Insight has been made about the overwhelming relative processor time consumption required for hashing frames, and the importance of computational efficiency in real-time video data processing. The project can serve as a starting point for further work on a more comprehensive fault detection package.

5.1 Implementation

Implementation work resulted in a largely self contained independent subsystem with no tight integration with the Direkt Link system as a whole. Loop detection and stream switching decision can be dropped in or removed from the system affecting other subsystems, other than needing to share processing power and sending the signal to switch streams. Figure 2 schematically illustrates how the implementation fits into the system. Frame buffers are read by the implementation, analogously to the videomixer as described in section 1.3. Switching between active video streams is done by the loop detection subsystem calling a script that alters the configuration values of the videomixer.

To quickly recapitulate the working flow of the implementation: One frame from each frame buffer is read and hashed. The frame hashes are stored in stream history buffers. History buffers are scanned for looping subsequences of hashes, and loop info per stream is updated. Loop info is summarised into a quality classification for each stream. Stream qualities are compared to each other and a decision whether to switch active stream is made.

5.2 Further work

Much work remains do be done in order to have comprehensive fault detection that can be used in professional video distribution settings for automated monitoring of video signals. This thesis work can only serve as a stepping stone to a more full fledged application. This section collects some thoughts for further work, that has resulted from working on this thesis project.
5.2.1 Real world testing

The decision to focus on loop detection was based on a theoretical intuition of the effects circular frame buffers might have on how faulty video signals might commonly present. A more structured investigation of what types of errors are common or likely occur in reality could be informative. Insight into how common errors are generated would be beneficial for more informed decisions on what fault detection strategies to focus work on to complement loop detection. To what degree loop detection needs to be complemented with other fault detection strategies could also be investigated. Conceivable but in reality unlikely errors may not motivate implementation. Knowledge of what causes common errors, or how they are produced, is also important for simulating or reproducing said errors in a controlled manner for testing purposes. Developing a testing framework for development of a comprehensive fault detection application seems like a project in its own right.
5.2.2 Effects of video compression

The project was originally undertaken under the premise of analysing uncompressed SDI video signals. Distribution of digital video over IP networks typically involve video compression, however. It is worth investigating what errors video compression can introduce on a video signal. Notice that in addition to errors during the compression itself, compressing an already faulty stream might alter the presentation of the error. For example, if an already looping stream is compressed, it is conceivable that two identical frames will not remain identical on a bit-level, which would render the current implementation of the loop detection ineffectual. Vice versa, it is also worth considering whether introducing errors on a compressed stream might have any significant consequences compared to introducing errors on an uncompressed stream.

5.2.3 Perceptual hashing

Tying into the previous section, the potential effects of video compression on image quality could motivate or even necessitate the use of perceptual hashing, as described in section 3.4. Perceptual hashing is a broad general concept. What available options of perceptual image hashing algorithms exist should be explored, and the algorithms evaluated from both the perspectives of functionality and processing performance.

5.2.4 Statistical quality analysis

The stream quality classification described in Chapter 4 might be improved with more intelligent statistical analysis, to cover patterns of interspersed looping. Statistical analysis could also lend itself to produce more fine grained classifications than three used in this work. In particular this could be important in an application that must make stream switching decisions based on more factors than only the presence of looping picture sequences.

5.2.5 Audio

While sound issues are on the list of fault scenarios to handle, no audio work has been done for this thesis work. Raw digital audio is encoded in sequences of samples, sampling the momentary amplitude of an audio waveform at discrete time point. Sample in the context of digital audio encoding/playback should not be confused with sample as it is used in audio production, where a sample often means a section of recorded audio.

When the difference between two waveform samples is too large, this tends to produce an audible pop. Presumably many such pops in close time proximity would result in a crackling sound. When the relative amplitude of a sampled waveform is so large as to be described within the bit depth of a sample, the digital representation of the waveform will appear with truncation of waveform peaks. This is called clipping and appears as a distorted sound with some crackling qualities. Detection of crackling or popping in an audio stream could therefore begin with measuring the difference between consecutive samples and detecting consecutive samples with maximum or minimum sample representation, which would indicate clipping.
Quiet or no sound is the result of contiguous sequences of samples with very small or no difference between samples. Detection of sound loss could again then be done by measuring the difference between samples.

5.2.6 Signal loss and data corruption

Intinor’s system is already capable of detection a total loss of video signal. This still needs to be incorporated into the stream switching decision process.

The signal receiving SDI hardware is also capable of detecting and reporting data corruption resulting from cable faults, by way of checksum comparison. This should also be included in the total stream evaluation.
Chapter 6
Conclusions

The three largely independent aspects loop detection, frame hashing and stream switching described in Chapters 2, 3 and 4 form the theoretical basis for the resulting implementation. Loop detection in particular entirely abstracts away the problem domain of SDI video signals into strings of arbitrary elements, where the only requirement on the elements is that they can be compared for equality. The purpose of frame hashing is to define this comparison operator on video frames as elements. Of course, defining a comparison operation can be done in other ways than hashing; hashing is used for the purpose of computational speed. The loop detection algorithm is entirely independent of the frame hashing as long as the comparison operator exists. Vice versa, frame hashing is entirely unconcerned with the use of the comparison operator within the loop detection algorithm. The stream switching is independent of the operations of loop detection and frame hashing. Stream switching only requires the resulting output of the loop detection as input to determine, in some more or less arbitrary manner, the quality of an element stream. The decision of whether to switch between streams then becomes a problem entirely separate from the other aspects of the whole. These three problem aspects come together to form the whole initial problem of automatically switching between redundant streams.

The initial thesis problem was specified for SDI video signals, which turns out to not have been very important. After some time spent familiarizing with the problem domain and the software environment where the resulting implementation would eventually fit in, the SDI specification of the problem was abstracted away. Both loop detection and stream switching strategy are problems independent of the underlying signal coding. The exact coding of a video signal might conceivably be more relevant for the data processing involved in frame hashing, but even here the specifics of the SDI signal is abstracted away by Intinors system. This means that SDI signals and YUV coding mentioned in Chapter 1 are largely unimportant for the final resulting implementation.

One should consider that it is not necessarily obvious what information from the problem domain is relevant for solving a given problem. Abstracting the facts of the context of the problem into information relevant for solving the problem must be considered part of the project. Therefore much work, especially initially, goes into what does not appear in the final result, in addition to the work that goes into what does appear in the result.
Acknowledgement

I wish to thank Björn Weinehall for patient assistance and advice, and everyone at Intinor for being welcoming.
Bibliography


[2] Umut Acar, 
http://www.cs.cmu.edu/afs/cs/academic/class/15210-s15/www/
lectures/mcs-notes.pdf

https://www.smpte.org/sites/default/files/
2013-09-10-3GSDI-Hudson-V3-Handout.pdf


