Practical and Scalable Transmission of Segmented Video Sequences to Multiple Players using H.264

Peter Quax, Fabian Di Fiore, Panagiotis Issaris,
Wim Lamotte, and Frank Van Reeth

Hasselt University - tUL - IBBT
Expertise Centre for Digital Media
Wetenschapspark 2
BE-3590 Diepenbeek
Belgium
{peter.quax, fabian.difiore, panagiotis.issaris,
wim.lamotte, frank.vanreeth}@uhasselt.be
http://www.edm.uhasselt.be

Abstract. We present a practical way to distribute viewports on the same video sequence to large amounts of players. Each of them has personal preferences to be met or is limited by the physical properties of his/her device (e.g., screen size of a PDA or processing power of a mobile phone). Instead of taking the naïve approach, in which sections of the video sequence are decoded and re-encoded for each of the clients, we have exploited advanced features offered by the H.264 codec to enable selection of parts of the video sequence by directly manipulating the encoder-generated bitstream. At the same time, we have overcome several practical issues presented by the fact that support for these features is sadly lacking from the state-of-the-art encoders available on the market. Two alternative solutions are discussed and have been implemented, enabling the generation of measurement results and comparison to alternative approaches.

Key words: Video coding, Transmission, Remote Rendering

1 Introduction

Motivation. The integration of multimedia streams of various nature is an important feature of many of today's games. While audio communication is widely supported — at least for some genres — the same is not yet true for video. The causes for this are many, but the required processing power and efficient distribution methods are probably the main culprits. Nevertheless, it opens the door for many interesting applications. The successful integration of video sequences for games is not limited to inter-person communication: various other applications, such as omnidirectional video [1], the replacement of traditional computer-generated background images by real-life captured sequences

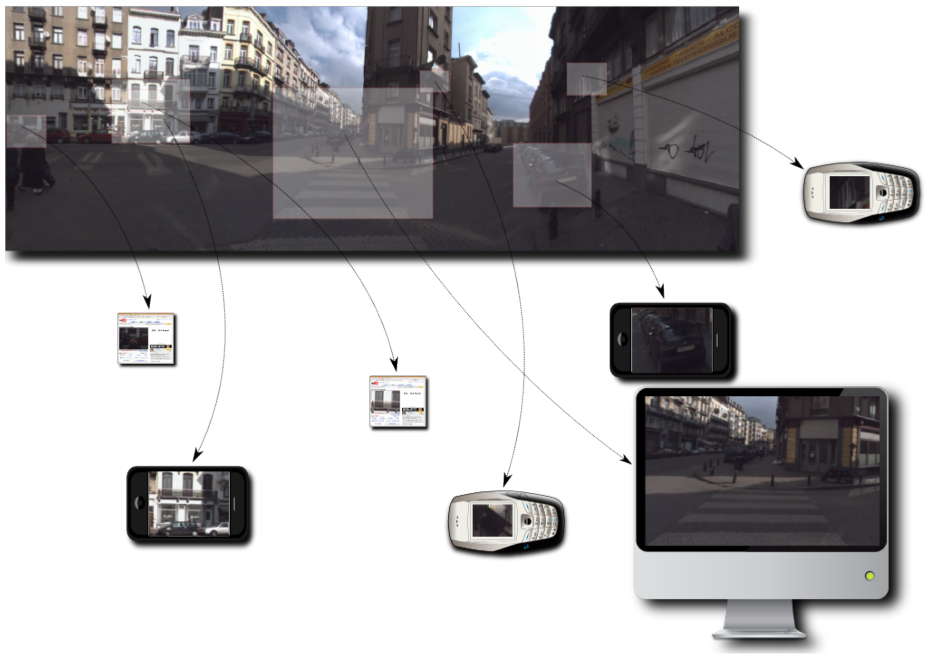


Fig. 1. Concept of distributing viewports on the same video sequence to large amounts of players.

or enabling low-power devices to play state of the art games through remote rendering can be envisaged. This paper considers the case in which parts of the same video sequence(s) need to be transmitted to a number of players (see figure 1). Such a condition exists, for example, in remote rendering setups [2,3,4] — in which a high number of individual viewports is generated on-the-fly by a high-performance server setup. It would be beneficial if a single encoding step could be used to generate a (large) stream, instead of maintaining multiple instances of the video encoding software to generate the individual viewports. A similar setup can be created for video communication purposes, in which a video conference is used during gameplay (e.g., for FPS or MMO games). In case a centralised server is responsible for generating the video sequences for individual participants (depending on their bandwidth capacity or display size for example), it would again be beneficial to generate a single video sequence that can be segmented at a later time. It should be clear that a system that takes a single (large) sequence and segments it through encoding/decoding parts of the frames would not scale, thus the need arises for being able to crop the desired regions directly from the encoded bitstream.

Contribution. As stated before, this paper focuses on sending parts of the same video sequence to a multitude of devices, each having their own set of characteristics and player preferences. Our technique focuses on selecting areas in a video sequence for individual transmission directly in the encoded bitstream, without requiring a separate decoding/encoding phase for each of them. In this paper we elaborate on exploiting advanced features that exist in the H.264 standard (e.g., slices) in order to target our applications. In addition, as there is currently little to no support for these features in the available codecs we present a way to work around these practical limitations and to enable an investigation into the efficiency of the approach.

Approach. In this section, we will briefly explain the general outline of the approach. For a more detailed description, we refer to the appropriate parts of the paper. Consider the case in which a single large video sequence is generated by an application, which is subsequently provided as input for a viewport selection service (based on player preferences or device limitations). A naïve way to perform the segmentation would be to decode a specific part of the sequence and provide it as input for an encoding instance. It should be clear that such an approach would not scale to the number of participants that is representative for today's games. Instead, one could provide some additional information in the bitstream representing the large video sequence. Through the use of this information, specific sections can be cropped simply by selecting those elements of the bitstream that are useful, eliminating the need for separate decoding/encoding phases. Of course, there are some requirements that need to be fulfilled in order to be able to just cut out specific sections of the bitstream, one of them being the requirement to remove dependencies between candidate viewpoint areas. This is accomplished by using slices, one of the features offered by the H.264 standard. In case we limit the motion estimation to these slices, they become self-contained and can be cut from the encoded bitstream in an efficient manner. By following through on this approach, we can combine several slices to re-generate complete video sequences, representing different viewports.

Paper Organisation. In section 2, we provide a short introduction to the features of the H.264 standard that are relevant for this paper. Some of the applications are also described in more detail. Section 3 details our specific approach to the problem and discusses two alternative solutions. The advantages and disadvantages of both are presented. A comparison between our approach and others is presented in section 4. Some pointers to possible future work are presented in section 5.

2 Related Work

A video encoder takes a sequence of frames as its input and spits out a — typically much smaller — bitstream conforming to a certain specification. To do so, block-based video codecs first divide each input frame into a grid structure of which each individual area is called a macroblock. The resulting output bitstream consists of a highly compressed representation of these macroblocks. As the representation of each macroblock in the compressed form is of variable length, a bit error in for example the first macroblock leads to the failure to decode any macroblock in the frame.

The H.264 specification [5] provides a feature called a “slice” which allows these macroblocks to be grouped. H.264, also known as MPEG-4/AVC, is the most recent ITU & MPEG video coding standard widely used (Blu-ray, DVB-S2, QuickTime, . . .). Because H.264 allows macroblocks to be grouped into multiple slices, each slide becomes individually en/decodable. A coded video bitstream can have a very simple structure using one slice per frame, however, having multiple slices is advantageous for parallel processing and network transmission where errors in a slice will not prevent the correct decoding of the other slices.

In our work, we exploit this feature in order to ignore unnecessary slices without having to re-encode the cropped region.

3 Approach

By regarding each frame of a video as being composed of many independent areas, we could – for each client – select the areas needed according to his viewport. As explained in section 2 the H.264 standard provides a way to divide frames in smaller units by means of slices which are groups of macroblocks belonging to the same frame and contain the actual encoded pixel data. In general, the H.264 specification describes the slice structure as being totally irregular, allowing each frame to be encoded in a different number of slices.

For our approach, we force our encoder to use a more regular slice subdivision with each frame containing the same number of slices, and each slice occupying the same area over all frames. Encoding a video stream using these constraints allows us to instantly know the visual area a specific slice represents, without needing to decode the actual contents.

Figure 2 illustrates several client requests for a particular view and the corresponding selected area of slices enclosing each view request.

Whenever a client wants to view just a particular part of the video, we somehow need to remove the areas of the image falling outside the selected area. The naïve approach of just sending only the selected slices imposes the client with a corrupt bitstream which regular decoders can not cope with. Also, cropping out pixel data as is common in known photo and videoprocessing tools would imply compressing the cropped out region for each client individually, since each client might be looking at a different segment of the video (i.e. a different area). Recompressing the cropped out region could be feasible for a few clients, but it cannot be considered scalable.

In the following subsections we elaborate on our specific approach to this problem and discuss two alternative solutions.

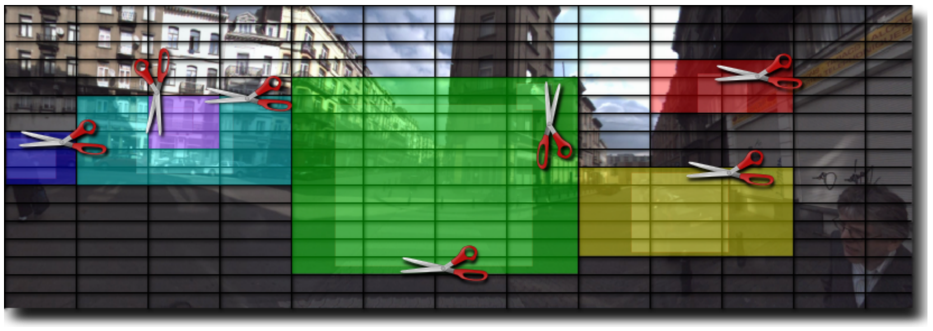


Fig. 2. Each frame is subdivided into a same number of slices. For each client, the areas depicted are needed to enclose the requested view (shown by the smaller area).

3.1 Removing Slices

If we want to view just a particular part of the video, we need to remove the areas of the image falling out the selected area. Ideally, this can be accomplished by removing all unneeded slices (See figure 3).

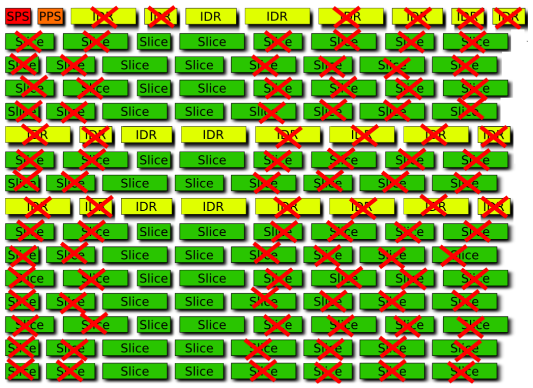


Fig. 3. Removing blocks removes part of a frame. In this case, each row depicts a single frame.

Unfortunately, deleting one slice affects all following slices. This is because each slice starts with a "first_mb_in_slice" field which contains the macroblock number of the first macroblock coded in this slice. Therefore, as we delete entire slices containing macroblocks, many macroblock numbers disappear and all slices following a deleted slice need to be altered. Moreover, H.264 uses variable bit length encoding meaning that different values are possibly encoded using codes of different length. As a result, changing one single value could imply changes in all bytes that follow.

One further problem is related to the fact that macroblocks are implicitly identified by their number. The data that motion compensation uses to reconstruct the frame is also identified by the same number. When the cropped out area is static (i.e. the client's view frustrum does not move) this causes no harm as the renumbering due to the cropping is the same for each frame, and motion vectors point to the correct data. However, when the cropping window moves (due to a moving view frustrum), problems arise as the numbering of macroblocks is different before and after the movement of the cropping window. Consequently, the motion compensation process in the decoder will start using the wrong image data to reconstruct the image. The only way to avoid this problem, is to stall the cropping window until the next intra-frame. This is a full frame used in MPEG-4 AVC encoding (comparable to I-frames) that does not need any information from the frame before it or after it to be played back properly.

Furthermore, moving the view frustum might cause the video frame size to change (see Figure 4). Although not restricted according to the specs, this causes problems with current decoders including VLC, Xine and MPlayer.

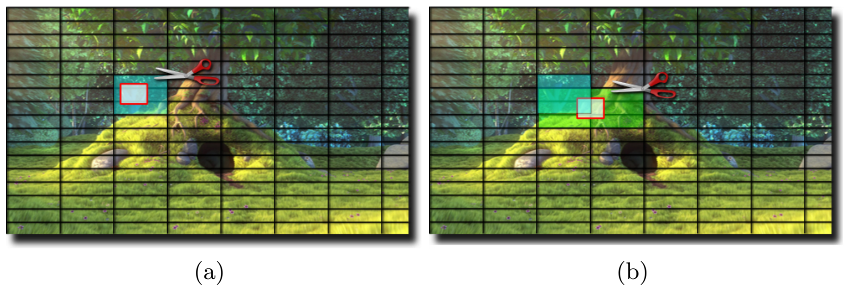


Fig. 4. Moving the view frustrum can cause the original (a) frame size to change (b).

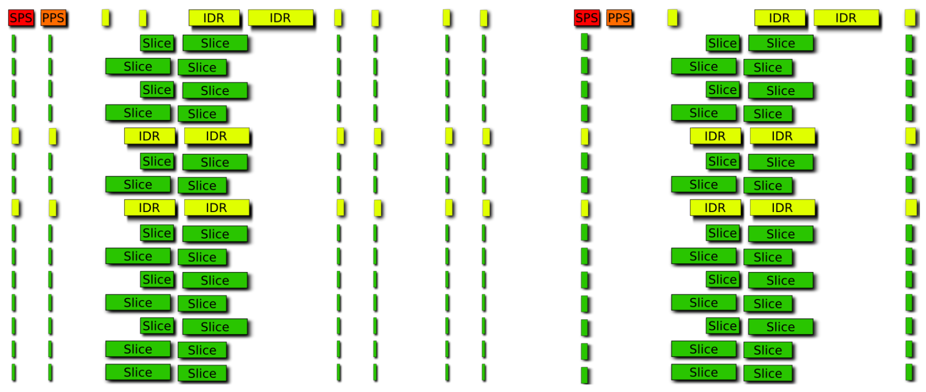
To summarise, removing the unnecessary blocks results in the smallest possible stream where the receiving end receives a simple stream with a resolution matching its request without any modifications needed on the client side. The mentioned issues, however, restrain this approach from being usable in real cases.

3.2 Replacing Slices

A different solution, which alleviates the aforementioned issues, uses replacement rather than removal of slices. Slices which would have been removed in the previous solution are now instead replaced by an artificially generated slice of minimal size. Because of this, no slice numbers need to be altered and motion compensation uses the correct numbers. Furthermore, no shifting of bits is needed as the existing and needed slices are sent as is. No header modification is needed either.

As the slices need no modification, this approach is more scalable and each client can be sent the slices it requires, unaltered. The only compensation of this approach is the larger decoded picture buffer size at the decoder side and the need for the decoder to be able to handle a larger number of slices.

The replacement slices can be generated on the fly (but can be cached too). For intra-frames (comparable to I-frames), the replacement slices contain the encoded slice data consisting of a number of gray blocks. Concerning inter-frames (comparable to B- and P-frames), the replacement slices contain encoded slice data using skip-bits, representing data which can be skipped as they are supposedly identical to the same data in the previous frame. This is also illustrated in Figure 5(a).



(a) (b)

Fig. 5. a) Replacing slices empties part of a frame. b) Grouping empty slices.

Replacing each unneeded slice with a generated slice is suboptimal and results in high overhead of headers. Furthermore, each slice might be sent individually which would add more overhead due to RTP, UDP, IP and data link layer headers. To this end, we also restructure the bitstream by generating slices containing spans of macroblocks from one unaltered slice to the next unaltered slice (see figure 5(b)).

We can conclude that this approach is very close to the space-efficiency-optimal case (when using the slice-deletion technique), but without the computational overhead and addressing issues.

3.3 General Concerns

In this section we elaborate on some general concerns and workarounds, applicable to both approaches.

Frame Dependency. Although slices are independently decodable, they do depend on the previous frame being entirely available. So, if we look at figure 6 we can see the motion vectors pointing out of various slices to the same slices in previous frames. However, as slices can be removed or replaced over frames motion vectors could arise which don't refer to the correct data in the previous frame. To this end, we modified our encoder to restrict motion vectors to the current slice area in the previous frame.

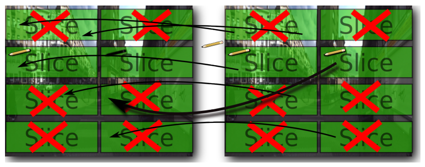


Fig. 6. Problematic motion vectors when slices are removed or replaced.

Grid Structure. Note that in the current approach, our slices are rectangular. As H.264 forces grouping of macroblocks to be consecutive in raster scan order our slices cannot contain more than one row of macroblocks. Furthermore, because each macroblock has a height of 16 pixels, the height of our slices is constrained to 16 pixels as well. Therefor, in order not to end up with too many slices each frame is divided into rectangular slices measuring 16 x 192 pixels. It might also be worth investigating the FMO (Flexible Macroblock Ordering) feature of the H.264 specification which allows ordering macroblocks in a more flexible way. This could allow us to implement the grid structure more efficiently by reducing the number of slices needed. Unfortunately, using FMO requires it to be supported by both the H.264 encoder at the server side and the H.264 decoder at the client side which is not the case at present. Figure 7 depicts our rectangular grid structure and the hypothetical case when employing FMO.

Moving View Frustrum. Moving the view frustum might also cause motion compensation to use data we cropped out in previous frames (see figure 8). The motion vectors could point to a area which is no longer within the cropped area. We can send more slices then strictly needed. If during movement we reach the border of the available area, we need to change the cropping area.

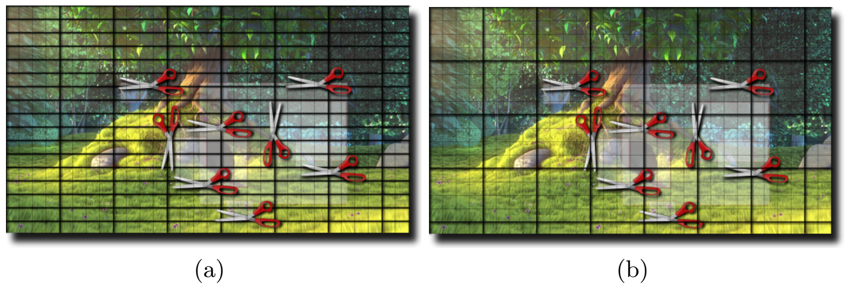


Fig. 7. Comparison of grid structures. a) Currently supported by H.264 encoders and decoders. b) Hypothetical case when employing Flexible Macroblock Ordering (FMO).

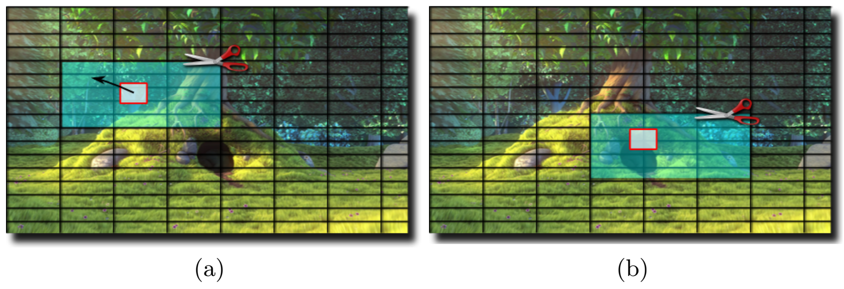


Fig. 8. Moving the view frustrum can cause motion compensation to use data we cropped out in previous frames.

Prediction Artifacts. If we would only send the slices needed to fill the current viewport, any sudden and/or large movement could cause visual artifacts. This is because we would be looking at an area for which we suddenly start receiving predicted slices for which we do not have the basis of the prediction. In other words, we start receiving predicted slices for that area, but we either never received the initial slices or either received artificial slices which are plain gray. As these artifacts are distracting, we send out more slices (i.e. located around the viewport) than strictly needed to fill the current viewport.

4 Measurement Results and Comparison

In this section we elaborate on some results measuring our optimal proposed approach of replacing unneeded slices by empty ones. All comparisons are made against the official reference encoder (called JM or Joint Model) [6]. Our implementation is also built on the reference encoder as this is the only encoder available that fully implements the H.264 standard. Concerning the clients, no specific decoder has to be employed.

All measurements were performed on the omnidirectional video sequence shown in the top row of figure 9. The medium row depicts snapshots of two clients with a different view on the input stream while the bottom row illustrates the actual videostream sent to a third client.

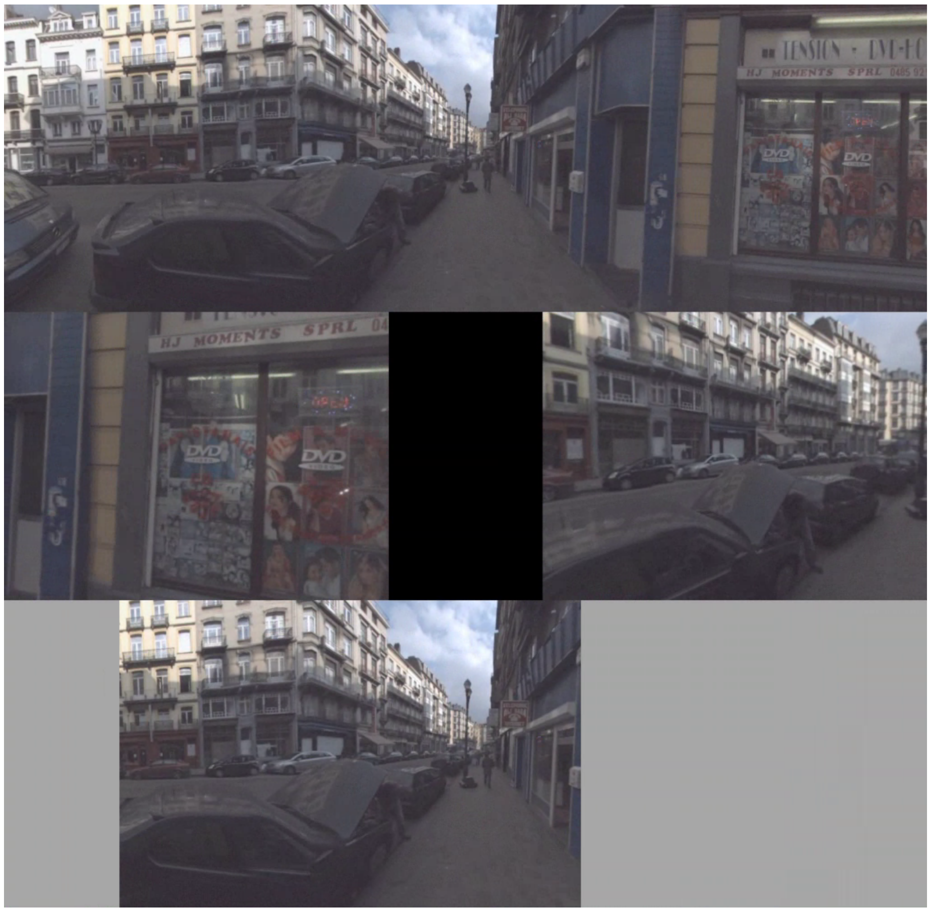
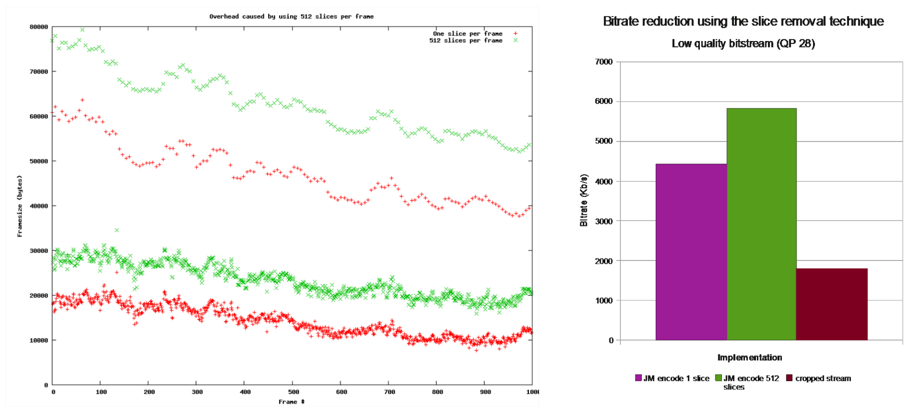


Fig. 9. Top row) Snapshot of the input stream. Medium row) Snapshots of two clients with a different view on the input stream. Bottom row) Snapshot of the actual videostream sent to a third client.

Figure 10(a) shows the grid structure overhead caused by using 512 slices per frame compared to 1 slice per frame (i.e. a full frame). Figure 10(b) depicts the same information (first two bars) but also the significantly lower bitrate (dark red bar) when applying our replacement approach to the 512 slices. Note that once H.264 encoders and decoders support the FMO (Flexible Macroblock

Ordering) feature, the overhead due to the amount of slices will be reduced and, hence, the final bitrate as well.



(a) (b)

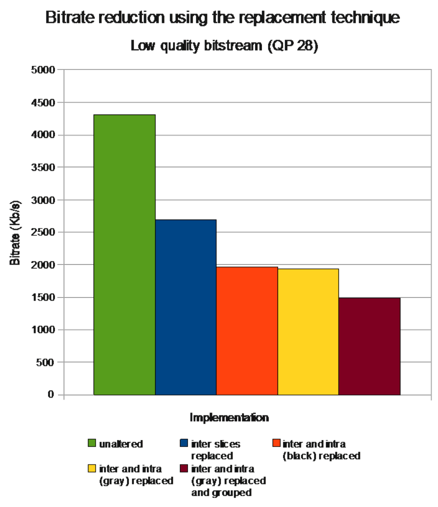
Fig. 10. a) Grid structure overhead caused by using 512 slices per frame. b) The purple bar shows the size of an encode with JM using 1 slice per frame (i.e. a full frame). As indicated by the green bar, using JM to encode using 512 slices per frame causes considerable overhead to represent the same frame. However, the data our cropping server actually sends is significantly lower, which is shown in the dark red bar.

Starting again from an unaltered stream (Figure 11(a), green bar) and applying our approach to just the inter slices (blue bar) already a reduction of 1.6Mbit for a 4.3Mbit stream is achieved. For the used samples, intra slices only represent 1/7 of the data but they are relatively large compared to inter slices. Additionally, using the replacement technique (replacing slices with black colour) on both inter and intra slices yields a considerable advantage (orange bar). The advantage of using the slightly more compact gray block representation is negligible (yellow bar). Finally, regrouping the replaced slices partially alleviates the overhead of the slices' headers (dark red bar).

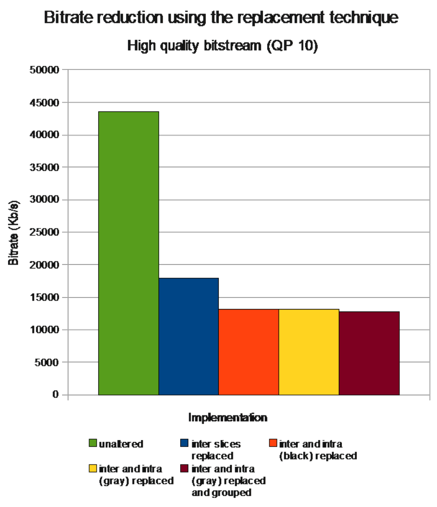
For higher quality streams (Figure 11(b)), our technique gives even better results. Now, with only replacement of inter-slices we see a reduction of a 43Mbit stream to 18Mbit (blue bar). Further replacement of intra-slices results in a 13Mbit stream (orange and yellow bar). Regrouping results in a stream just below 13Mbit (dark red bar).

5 Conclusion and Future Work

We have shown a practical way to distribute parts of a video sequence to large amounts of users. By adapting existing techniques from the H.264 standard, we



(a)



(b)

Fig. 11. Bitrate reduction using the replacement technique. Green bar = unaltered stream; blue bar = inter slices replaced; orange bar = inter and intra slices replaced (black); yellow bar = inter and intra slices replaced (gray); dark red bar = inter and intra replaced and regrouped. a) Low quality stream. b) High quality stream.

have shown that it is possible to perform the frame extraction directly from the compressed bitstream, eliminating the need for additional decoding and encoding phases. Test results have shown that the overhead associated with the slice structures is mitigated by the bandwidth reduction in transmitting high-quality video streams.

Additional benefit would be gained if FMO was practically usable and performable with respect to both the encoder and decoder. As this is not yet the case, we will look at the integration in open-source codecs (e.g., libavcodec).

Acknowledgements

Part of the research at EDM is funded by the ERDF (European Regional Development Fund) and the Flemish government. The research presented in this paper is part of the IWT-Teleon project.

References

1. T.E. Boult, R. J. Micheals, M. Eckmann, X. Gao, C. Power, and S. Sablak. Omnidirectional video applications. In *Proceedings of the 8th International Symposium on Intelligent Robotic Systems*, 2000.

2. Daniel Cohen-Or, Yuval Noimark, and Tali Zvi. A server-based interactive remote walkthrough. In *Proceedings of the sixth Eurographics workshop on Multimedia 2001*, pages 75–86, New York, NY, USA, 2002. Springer-Verlag New York, Inc.
3. Joachim Diepstraten and Thomas Ertl. Remote Line Rendering for Mobile Devices. In *Proceedings of Computer Graphics International (CGI2004)*, pages 454–461. IEEE, 2004.
4. Peter Quax, Bjorn Geuns, Tom Jehaes, Gert Vansichem, and Wim Lamotte. On the applicability of remote rendering of networked virtual environments on mobile devices. In *Proceedings of the International Conference on Systems and Network Communications*, pages 16–16. IEEE, 2006.
5. H.264 : Advanced video coding for generic audiovisual services. World Wide Web, `http://www.itu.int/rec/T-REC-H.264`, 2009.
6. H.264/AVC JM Reference Software. World Wide Web, `http://iphome.hhi.de/suehring/tml/`, 2009.