THREE-DIMENSIONAL VIDEO CODING ON MOBILE PLATFORMS

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND

ELECTRONICS ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCES

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Can Bal

September 2009

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Levent Onural (Supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Nail Akar

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Tolga Çapın

Approved for the Institute of Engineering and Sciences:

Prof. Dr. Mehmet Baray
Director of Institute of Engineering and Sciences

ABSTRACT

THREE-DIMENSIONAL VIDEO CODING ON MOBILE PLATFORMS

Can Bal
M.S. in Electrical and Electronics Engineering
Supervisor: Prof. Dr. Levent Onural
September 2009

With the evolution of the wireless communication technologies and the multimedia capabilities of the mobile phones, it is expected that three-dimensional (3D) video technologies will soon get adapted to the mobile phones. This raises the problem of choosing the best 3D video representation and the most efficient coding method for the selected representation for mobile platforms. Since the latest 2D video coding standard, H.264/MPEG-4 AVC, provides better coding efficiency over its predecessors, coding methods of the most common 3D video representations are based on this standard. Among the most common 3D video representations, there are multi-view video, video plus depth, multi-view video plus depth and layered depth video. For using on mobile platforms, we selected the conventional stereo video (CSV), which is a special case of multi-view video, since it is the simplest among the available representations. To determine the best coding method for CSV, we compared the simulcast coding, multi-view coding (MVC) and mixed-resolution stereoscopic coding (MRSC) without inter-view prediction, with subjective tests using simple coding schemes. From these tests, MVC is found to provide the best visual quality for the testbed we used, but MRSC without inter-view prediction still came out to be promising for some of

the test sequences and especially for low bit rates. Then we adapted the Joint Video Team's reference multi-view decoder to run on ZOOM™ OMAP34x™ Mobile Development Kit (MDK). The first decoding performance tests on the MDK resulted with around four stereo frames per second with frame resolutions of 640×352. To further improve the performance, the decoder software is profiled and the most demanding algorithms are ported to run on the embedded DSP core. Tests resulted with performance gains ranging from 25% to 60% on the DSP core. However, due to the design of the hardware platform and the structure of the reference decoder, the time spent for the communication link between the main processing unit and the DSP core is found to be high, leaving the performance gains insignificant. For this reason, it is concluded that the reference decoder should be restructured to use this communication link as infrequently as possible in order to achieve overall performance gains by using the DSP core.

Keywords: three-dimensional video, 3D video, mobile platform, video coding, H.264, MPEG-4 AVC, multi-view coding, MVC, mixed-resolution stereoscopic coding, MRSC, DSP, OMAP

ÖZET

MOBİL PLATFORMLAR ÜZERİNDE ÜÇ-BOYUTLU VİDEO KODLANMASI

Can Bal
Elektrik ve Elektronik Mühendisliği Bölümü Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Levent Onural
Eylül 2009

Kablosuz iletişim ağlarının ve cep telefonlarının çoğulortam özelliklerinin gelişmesi ile, yakın zamanda üç-boyutlu (3B) video teknolojilerinin, öncellikle sadece yeniden oynatma biçiminde ve daha sonra 3B görüntülü konuşma olarak cep telefonlarına uygulanması beklenmektedir. En güncel 2B video kodlama standardı olan H.264/MPEG-4 AVC'nin önceki standardlara göre daha etkili kodlama sunması nedeniyle, en yaygın olarak kullanılan 3B video veri biçimlerinin kodlanma teknikleri bu standardı baz almaktadır. En yaygın 3B video veri biçimleri arasında çok-bakışlı video, video-artı-derinlik, çok-bakışlı video-artı-derinlik ve katmanlı derinlikli video bulunmaktadır. Bulunan en basit 3B video veri biçimi olması nedeniyle, mobil platformlarda kullanmak amacıyla, çok-bakışlı videonun bir özel durumu olan geleneksel stereo video veri biçimi seçilmiştir. Geleneksel stereo video için en iyi kodlama tekniğini belirlemek amacıyla eş-anlı kodlama, çok-bakışlı kodlama ve bakışlar arası tahmin olmadan karışık-çözünürlüklü stereoskopik kodlama teknikleri basit kodlama düzenleri kullanılarak öznel sınama yöntemi ile karşılaştırılmıştır. Yapılan öznel sınamalarda, kullanılan sınama ortamı için çok-bakışlı kodlama en iyi görsel

başarımı sağlarken, bakışlar arası tahmin olmadan karışık-çözünürlüklü stereoskopik kodlama da bazı sınama dizilerinde ve özellikle düşük bit hızlarında tatmin edici sonuçlar vermiştir. Bu sınamalar sonrasında *Joint Video Team*'in örnek çok-bakışlı kodçözücüsü, ZOOM™ OMAP34x™ *Mobile Development Platform* üzerinde çalıştırılmak üzere uyarlanmıştır. Yapılan kod çözümü sınamaları 640×352 çözünürlüklü videolarda, birim saniyede ortalama dört stereo çerçeve kod çözümü ile sonuçlanmıştır. Başarımı artırmak amacıyla, kod çözücü yazılımının profili çıkarılmış ve en talepkar algoritmalar bütünleşik sayısal sinyal işleme biriminde (DSP) çalıştırılmak üzere uyarlanmışlardır. Yapılan sınamalar sonucunda DSP üzerinde %25 ile %60 arasında başarım kazancı elde edildiği gözlemlenmiştir. Ancak mobil platformun tasarımı ve yazılımın yapısından dolayı, ana işlem birimi ve DSP arasındaki iletişim için gereken sürenin yüksek olduğu ve elde edilen başarım kazançlarını etkisiz bıraktığı belirlenmiştir. Bu nedenle, DSP üzerinde başarım elde etmek için yazılımın yapısı değiştirilerek bu iletişim bağının olabildiğince az kullanılacak biçime getirilmesi gerektiği sonucuna varılmıştır.

Anahtar Kelimeler: üç-boyutlu video, 3B video, mobil platorm, video kodlanması, H.264, MPEG-4 AVC, çok-bakışlı kodlama, karışık-çözünürlüklü stereoskopik kodlama, DSP, OMAP

ACKNOWLEDGMENTS

I would like to express my gratitude to Prof. Levent Onural, my professor and then my supervisor for his support and invaluable guidance. It has been a great honor for me to be a student of him.

I would also like to thank Assoc. Prof. Dr. Nail Akar and Assist. Prof. Dr. Tolga Çapın for serving as members of my thesis committee and accepting my invitation without hesitation.

I want to thank my parents and my grandmother as well, for being so understanding and supportive throughout all my life. I definitely would not be able come to the point I have without them.

Furthermore, I am grateful for the support and encouragement of my lovely girlfriend Yasemin, who was my sanctuary through the stressful times. I also want to thank my selfless friend Üstün for his extensive help during my research.

Finally, I would like to acknowledge the European Commission's Seventh Framework Programme project, the 3DPhone for the support of this work and all of its partners for providing me a collaborative working environment. In particular, I would like to thank Fraunhofer HHI for helping me with the subjective tests carried out in Chapter 4 of this thesis. I want to thank TÜBİTAK as well for supporting me financially through my MS degree program.

Contents

List of Figures

List of Tables

Dedicated to My Grandfather, Süper Dede

Chapter 1

INTRODUCTION

Today, with the evolution of the wireless communication technologies and the multimedia capabilities of the mobile phones, mobile phones started to serve many other purposes than just providing telephony services. Nowadays, people use their mobile phones for listening to music, watching video or TV, browsing the Internet, video conferencing and much more. On the other hand, the three-dimensional (3D) video technologies started to get commercialized, mostly for the cinema technologies, but also for TV and even for the internet. Therefore, 3D video technologies will soon get adapted to the mobile phones as well, first to provide 3D video playback but ultimately to support 3D video telephony. However, the available computational power and the power consumption are the bottlenecks for delivering 3D video technologies on the mobile phones. To overcome these bottlenecks, developers need to highly improve the video processing steps for the specific platforms they are working on. Additionally, choice of the 3D video representation and the associated coding method is crucial to provide satisfactory video playback to the consumers.

A few years back, when the 2D video services started to be delivered on mobile phones, different approaches were proposed to provide efficient video coding performances on mobile phones in the literature. These approaches vary from each other in terms of design methods, but they all try to optimize the macroblock-level operations such as motion compensation/estimation, quantization, transform operations, etc. and also variable length encode and decode operations. Some of these approaches focus only on software design on readily available general purpose processors, in order to provide flexibility for future modifications and enhancements [1], [2], [3], [4]. These approaches attempt to optimize the most demanding operations for their specific hardware by software means. Some others focus on designing dedicated hardware by using VLSI technologies and implement the whole video codec within a chip. These approaches usually provide better encode/decode performances over software optimization approaches with the cost of loosing flexibility [5], [6]. On the other hand, some other approaches try to keep a balance between the flexibility and performance and use a software-hardware co-design approach [7], [8], [9], [10]. These generally use a general purpose processor to manage the high-level operations and management of the additional hardware modules. Next to this, they design hardwired hardware modules which implement the demanding macroblock-level algorithms.

As mentioned, all of the previously referred work focus on the implementation of 2D codecs. Since the standardization process of 3D video coding is still ongoing, the problem of delivering 3D video services on mobile phones is quite new. We are not aware of any publications on performance analysis of 3D video codec implementations at this time. However 2D video codec implementation approaches are also related to our problem, since the 3D video coding methods are mostly based on the available 2D video coding standards.

For the choice of the 3D video coding method, recent studies show that simulcast, multiview video (MVC), mixed-resolution stereo video (MRSC) and video-plus-depth (V+D) coding methods yield promising results for being used on mobile environments [11],[12] in terms of visual quality, but their suitability in terms of computational power needs to be investigated on the selected hardware platform.

In our project, the 3DPhone[1], the hardware platform is selected with an intention to design and implement a complete 3D mobile device, with a 3D user interface and 3D video playback capabilities. Therefore, the hardware platform needed to be chosen so that it could serve for different kinds of applications. For that reason the consortium selected the ZOOM™ OMAP34x™ Mobile Development Kit (MDK) for development as it features the OMAP3430™ System-on-a-chip (SoC), which is equipped with an ARM main processor that most mobile or smart phones use today, a dedicated graphics processor for the 3D graphics rendering and a DSP chip for signal processing. The software environment of the OMAP34x™ MDK also played a significant role in its selection as the development platform. It features a Linux distribution called "Poky Linux" as its operating system; and this provides the opportunity to easily adapt the available Linux based software to run on the OMAP34x™ MDK.

In this thesis, we study and compare the performances of possible 3D video coding methods (simulcast, MVC and MRSC without inter-view prediction, with basic coding schemes specified specifically for this thesis) to be used on the OMAP34x™ MDK. In addition to the analytical comparison of simulcast and multiview coding, we provide the results of subjective tests conducted for comparing the performances of the three possible 3D video coding methods. We also provide the implementation and testing of a reference multiview decoder on the OMAP34x™ MDK. For the implementation, we take the software improvement

[1] http://the3dphone.eu/

approach since our hardware platform is fixed, but we also utilize the idea behind the software-hardware co-design approaches and try to optimize the most demanding video processing steps for the embedded DSP core.

Our contributions in this thesis work can be summarized as follows:

- According to the results of our subjective tests, MVC yielded the best visual quality over simulcast and the compared MRSC methods for the testbed and coding schemes we used, but MRSC without inter-view prediction still came out to promising for some of the test sequences we used, especially for low bit rates. However, since we did not have an embedded 3D screen ready for the OMAP34x^{TM} MDK, the tests were conducted with large displays; and it is concluded that for more reliable results, further subjective tests need to be conducted when the embedded 3D screen is ready.

- In the implementation of the MVC decoder on the OMAP34x^{TM} MDK, the decoding tests on the ARM processor yielded a low number of frames per second. Therefore we profiled the decoder software to find out the most demanding algorithms and selected some of these most demanding algorithms to be ported to run on the embedded DSP chip. Then the selected algorithms are implemented for the DSP and they achieved performance gains ranging from 25% to 60%, depending on the type of the ported algorithm. However, the communication link between the ARM and DSP processors is found to be very slow, and the time required for the processors to communicate exceeded the time gained by running the algorithms on the DSP. Therefore, it is concluded that the structure of the decoder software needs to be altered so that this communication link is used as infrequently as possible.

To give an outline of the thesis, in Chapter 2 we provide a concise summary of the H.264/MPEG-4 AVC coding standard, as it is the basis of all the

3D video coding methods compared and implemented in the thesis. In Chapter 3 there are brief explanations of the available 3D video representations and the associated coding techniques, including the reasons why they can or cannot be applicable on a mobile platform. In Chapter 4, the performance of mixed-resolution stereoscopic coding is compared with the performances of simulcast and multiview video coding methods. In this chapter the results of the subjective tests conducted for comparing the performances of these coding methods are provided and evaluated. In Chapter 5, we give the details of our implementation of a multiview decoder on the OMAP34x^{TM} MDK and the performance tests of the implemented decoder. Lastly, we draw conclusions and list a few possible approaches for the future research in the project in Chapter 6.

Chapter 2

H.264/MPEG-4 AVC STANDARD

H.264/MPEG-4 Advanced Video Coding (AVC) is the most recent video coding standard. It is developed by the collaboration of ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG), under the partnership effort known as Joint Video Team (JVT). The standard is referred to as H.264 by ITU-T and as MPEG-4 Advanced Video Coding (AVC) by ISO/IEC, but they have identical technical content.

This chapter is intended to provide a brief summary of the coding standard, with an emphasis on the parts that will contribute directly to the understanding of the thesis. Therefore some parts of the standard are intentionally omitted in this chapter. A detailed overview of the standard is available in [13]. Additionally in [14], the H.264/AVC standard is provided in detail with various examples, illustrations and figures.

2.1 Video Representation

The YCbCr color space represents a scene with a brightness component (luma) and two color-difference components (chroma). Since the human visual system is more sensitive to the luma component, such a representation allows downsampling the chroma components without losing much from the visual quality. Therefore H.264/AVC uses YCbCr color space representation and a sampling structure in which each of the chroma components are downsampled to one-fourth of the resolution of the luma (one-half in both horizontal and vertical directions).

2.2 Macroblocks

As mentioned before, H.264/AVC is a block-oriented video coding method, and handles the video frames by partitioning them into smaller elements called *macroblocks*. A macroblock is basically a fixed-size rectangular area in a video frame that consists of 16×16 samples of the luma component, and 8×8 samples of each of the chroma components.

Slices

Slices are groups of macroblocks. The macroblocks can be distributed into slices in a raster scan order or in a custom way by *Flexible Macroblock Ordering (FMO)*. FMO is going to be further discussed in the next paragraph. A video frame can consist of one or more slices. Each of the slices in a video frame is self-contained in the sense that it is possible to decode the samples contained in a slice without the use of data from other slices. Although this statement is valid, some information from other slices might sometimes be necessary in order to apply the *deblocking filter* (will be explained later) across slice boundaries.

With the use of FMO, the video frame can be partitioned into slices and macroblocks in any way desired by the use of the concept of slice groups. It is achieved by including a macroblock to slice group map in the generated bit-stream.

Each of the slices of a video frame can be predicted with a different prediction method. The labels of the slices according to the possible prediction methods, which are going to be discussed later, are as follows:

- I slice: In these slices all macroblocks are coded using intra-frame prediction.
- P slice: Macroblocks of P slices are coded using inter-frame prediction with only one motion-compensated prediction signal per prediction block. P slices can also use prediction modes of I slice.
- B slice: Macroblocks of B slices are coded using inter-frame prediction with two motion-compensated prediction signals per prediction block. B slices can also use prediction modes of P slice.
- SP slice: Intentionally left unexplained as it does not contribute directly to the understanding of the thesis. For details please refer to [13].
- SI slice: Intentionally left unexplained as it does not contribute directly to the understanding of the thesis. For details please refer to [13].

2.3 Handling of a Macroblock

The encoding process of a macroblock is crudely as follows: First every luma and chroma sample of the macroblock is predicted, either spatially or temporally. Then, the predicted version of the macroblock is subtracted from the original one and the residual is encoded using transform coding. For transform coding, the residual is subdivided into 4×4 blocks and each of them is transformed with

an integer transform. The generated transform coefficients are then quantized and encoded using entropy coding. These steps are explained in detail in the following sections.

2.3.1 Prediction

Each of the macroblocks can be predicted with one of the several possible choices of prediction modes, depending on which type of slice it belongs to. In the most general sense, there are two different prediction methods, which are *Intra-Frame Prediction* and *Inter-Frame Prediction*.

Intra-Frame Prediction

In H.264/AVC, intra-frame prediction is conducted in the spatial domain, meaning that it does not allow temporal prediction of the samples. In all types of slices, intra-frame prediction modes are allowed, which are Intra_4×4, Intra_16×16 and I_PCM. As the name implies, Intra_4×4 is a mode where each of the 4×4 luma blocks in a macroblock is predicted seperately. In this mode, the samples of a block are predicted using the neighboring samples of previously coded blocks to the left and/or above of the block to be predicted. This mode is good for predicting parts of a video frame with significant detail. Intra_16×16 mode works in a similar manner with Intra_4×4, only with the difference that it performs prediction over the whole 16×16 luma block. Therefore, this mode is best for predicting smooth areas within a video frame. On the other hand, I_PCM mode allows the encoder to temporarily disable the prediction and transform coding steps for a macroblock and directly put the sample values of a macroblock in the bitstream. This feature is good as it allows representing problematic areas accurately and puts an upper limit on the number of bits representing a macroblock.

For further details of intra-frame prediction, please refer to [13].

Inter-Frame Prediction

In H.264/AVC for inter-frame prediction, partitioning the macroblocks with luma block sizes of 16×16, 16×8, 8×16, 8×8, 8×4, 4×8, and 4×4 is supported. Unlike intra-frame prediction, inter-frame prediction allows both spatial and temporal prediction of the partitioned blocks. In P slices, the partitions are predicted by referring to a block of the same size in a reference frame. This referencing is achieved with a prediction signal, which includes a translational motion vector and the index of the reference frame. For the B slices, inter-frame prediction is very similar to the P slices, with the main difference that partitions in B slices can also be predicted as a weighted average of two distinct reference blocks.

The motion compensation in inter-frame prediction has an accuracy of one-fourth of the distance between the luma samples. When a motion vector points to a non-integer sample position, the luma samples at half or quarter sample positions are calculated by interpolation. The interpolation of the values at half sample positions is done by using a one dimensional 6-tap FIR filter. Then the values at quarter sample positions are found by averaging the values of the nearest integer and/or half sample positioned samples. For the details of the filtering operations, Figure 2.1 provides the necessary labels of the samples that will be used through the following mathematical expressions. Note that in this figure, samples at integer, half and quarter sample positions are labeled with capital, lower-case and double lower-case letters respectively.

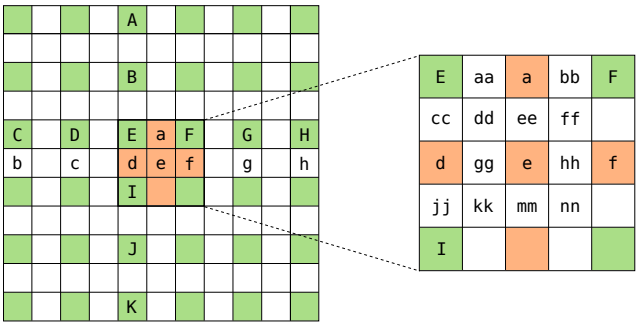


Figure 2.1: Interpolation of the samples at half and quarter sample positions.

The samples with labels a and d are located at half sample positions and their values are calculated by using the intermediate values a' and d'. a' and d' are found by applying a 6-tap FIR filter as follows:

$$a' = (C - 5D + 20E + 20F - 5G + H)$$
$$d' = (A - 5B + 20E + 20I - 5J + K).$$

Then the values of the samples a and d are computed from the values of a' and d' as follows, and then they are clipped to the range of 0–255:

$$a = (a' + 16) \gg 5$$
$$d = (d' + 16) \gg 5.$$

The value of the sample labeled as e is derived by calculating the intermediate value e' first as follows:

$$e' = b - 5c + 20d' + 20f' - 5g + h$$

where the values of b, c, g, f', and h are obtained in a way very similar to the calculation of d'. Once the value of e' is determined, e is derived from e' as follows and then it is clipped to the range 0-255:

$$e = (e' + 512) \gg 10.$$

The values of the samples at quarter sample positions with labels aa, bb, cc, jj, ee, gg, hh, and mm are computed by averaging and upward rounding the values of the nearest integer and half sample positioned samples. For example, the value of the sample labeled as aa is found as follows:

$$aa = (E + a + 1) \gg 1.$$

The values of the samples at quarter sample positions with labels dd, ff, kk, and nn are calculated by averaging and upward rounding the values of the nearest half sample positioned samples on the diagonal direction. For example, the value of the sample labeled as dd is found as follows:

$$dd = (a + d + 1) \gg 1.$$

For the associated chroma blocks of the predicted luma blocks, as expected, the accuracy is one-eighth of the distance between the luma samples. For interpolating the subsamples of chroma blocks, in H.264/AVC, always the bilinear interpolation technique is used.

For further details of inter-frame prediction, please refer to [13].

2.3.2 Transform, Scaling, and Quantization

As previously mentioned, in H.264/AVC, at the encoder side a transform operation is conducted on the residual blocks. Unlike the previous block-oriented video codec standards, H.264/AVC does not use 8×8 discrete cosine transform (DCT) for this transform operation, but defines a separable integer 4×4 transform. The defined transform matrix provides an approximation of DCT, with the following coefficients [15]:

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix}.$$

The selected coefficients allow the transform to be implemented with only addition and bit-shift operations. Another feature of this transform is that, with the selected coefficients encoder and decoder mismatches are prevented.

Once the transform is applied on a 4×4 block, the obtained transform coefficients are scaled and rounded. This quantization step is going to be further discussed later. The quantized coefficients are then arranged into a sequence by a technique called *zig-zag scanning* and this sequence is coded by entropy coding methods. At the decoder side, these steps are simply reversed.

For the residual chroma component of a macroblock, an additional 2×2 Hadamard transform is applied to the DC coefficients of the four 4×4 chroma blocks.

It is worth mentioning that, another 4×4 Hadamard transform is defined in the standard as well, specially for the Intra_16×16 prediction mode. As Intra_16×16 mode is intended for predicting the smooth areas of the video frame, when this mode is used, the 4×4 Hadamard transform is applied additionally on the DC coefficients of the sixteen 4×4 luma blocks of the residual macroblock.

Returning back to the quantization step, the scaling operation is controlled by a variable called the *quantization parameter*. It can take 52 values, ranging from 1 to 52. One step increase in this value corresponds to about 12% increase in the quantization step size (6 step increase corresponds to doubling the quantization step size). Therefore higher values of the quantization parameter result in a coarser quantization of the transform coefficients.

For further details of the transform and quantization steps, please refer to [13] and [15].

2.3.3 Entropy Coding

In H.264/AVC standard the syntax elements and the quantized transform coefficients are compressed by entropy coding methods. For the syntax elements a simple entropy coding scheme is used, with a single predetermined exp-Golomb codeword table.

For coding the quantized transform coefficients there exists a method called Context-Adaptive Variable Length Coding (CAVLC). In this method, there are a few different predefined VLC look-up tables. The choice of the VLC look-up table while coding the transform coefficients depend on the syntax elements that are already been coded. Since the VLC look-up tables are generated to match the corresponding conditional statistics, it provides a better compression over using a single VLC look-up table.

In H.264/AVC there also exists an entropy coding method called Context-Adaptive Binary Arithmetic Coding (CABAC), which can be used instead of CAVLC. As its name implies, it features an arithmetic coding method and it generates its alphabet adaptively according to the already coded syntax elements. Since it estimates the conditional probabilities in an adaptive manner, it provides even better coding efficiency over CAVLC.

Since lossless coding is not in the focus of this thesis, details for these entropy coding methods are not provided. For further details of the entropy coding methods, please refer to [13] and [16].

2.3.4 In-Loop Deblocking Filter

In all of the block-oriented video codecs, the decoded video sequences may include unintentionally created block-like defects, which are called *blocking artifacts*. In H.264/AVC, to overcome blocking artifacts an adaptive filtering operation, which is called the in-loop deblocking filter, is defined. The reason it is called adaptive is that whether the operation is going to be conducted or not depends on the values of the samples to be filtered. The filtering operation is applied on a block edge and affects up to three samples on either side of the block boundaries. Figure 2.2 provides the labels of the affected samples, as they will be referred while the filtering operation is being explained. Please note that filtering operation is conducted only on a certain direction (either horizontal or vertical), and the only reason for using the same labels for the samples along both directions is to provide a simpler mathematical explanation of the filter.

The decision of whether the filtering operation is applied on a block edge or not depends on two threshold values, which are α and β. The samples p_0 and q_0 are filtered if and only if the following conditions are satisfied:

1. $|p_0 - q_0| < \alpha$
2. $|p_1 - q_0| < \beta$
3. $|q_1 - q_0| < \beta$.

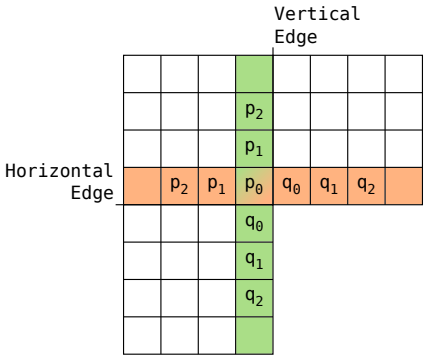


Figure 2.2: Labels of the samples affected by the deblocking filter.

Similarly, the sample p_1 is filtered if $|p_2 - p_0| < \beta$, and q_1 is filtered if $|q_2 q_0| < \beta$ holds.

These threshold values α and β increase with the quantization parameter (QP). Hence, if the QP is low, deblocking filter is not applied most of the time, since the differences between the sample values along the block boundaries are more likely due to the actual video content. However when QP is high, deblocking filter is applied more frequently, as it is expected to have a coarser and smoother decoded video and if there exist a high difference between the sample values along the block boundaries, it is most probably due to a blocking artifact.

For further details of the in-loop deblocking filter, please refer to [13] and [17].

Chapter 3

THREE-DIMENSIONAL VIDEO REPRESENTATIONS AND CODING METHODS

Various choices, depending on the application, are available for representing a three-dimensional (3D) video. In this chapter, some of these representations are going to be briefly described, without going into much detail on their coding steps. References are provided for further details of these representations and their associated coding methods.

In this section, the "ballet" 3D video sequence, which is used for the illustrations of this chapter, is provided by the courtesy of Interactive Visual Media Group at Microsoft Research (© 2004 Microsoft Corporation) and the associated depth data of the sequence is generated for the research provided in [18].

3.1 Conventional Stereo Video (CSV)

Conventional stereo video (CSV) is the least complex 3D video representation among the ones that are going to be explained in this chapter. In CSV, the 3D video is represented by two color videos (views) of the same scene shot with a certain difference in the angle of view. Figure 3.1 illustrates the CSV representation.

Figure 3.1: CSV representation ("ballet" sequence is used by the courtesy of Interactive Visual Media Group at Microsoft Research)

Usually, coding of CSV data includes steps very similar to the ones explained previously in the H.264/AVC standard. It is possible to encode the separate views as different video contents with a 2D video codec like H.264/AVC, and this method is called *simulcast coding*. However, since the views exploit correlation with each other, it is possible to encode CSV data more efficiently. There exists a coding method called *multiview coding (MVC)*, which allows inter-view prediction next to the temporal and intra prediction modes [19], [20], in order to reduce the total bit rate of the 3D video while maintaining the same visual quality. These coding methods are designed to encode even more than two views but are applicable to CSV data as well. There are also some methods which use view interpolation techniques to compensate for camera geometry [21].

Another possible solution for coding CSV data is *mixed resolution stereoscopic coding (MRSC)*. This method encodes a CSV data after downsampling one of the views to one fourth of its original resolution (one half in both horizontal and vertical

directions), and achieves a further reduction in bit rate by simply reducing the input data. This does not result in much loss of the overall visual quality since the human visual system is not very responsive to such an operation, and can compensate for it from the information coming from the full resolution view. MRSC can be coded in a simulcast manner or with inter-view prediction just like MVC, with the only difference that the right view gets upsampled back to its original resolution at the decoder side. Therefore, MRSC achieves a further reduction in bit rate when compared to the full resolution simulcast coding and MVC methods, with the trade-off of additional computational complexity coming from the downsampling and upsampling operations. Figure 3.2 provides an illustration of the downsampling operation for MRSC. Since MRSC is going to be explained in further detail in Chapter 4, its discussion at this point is intentionally limited.

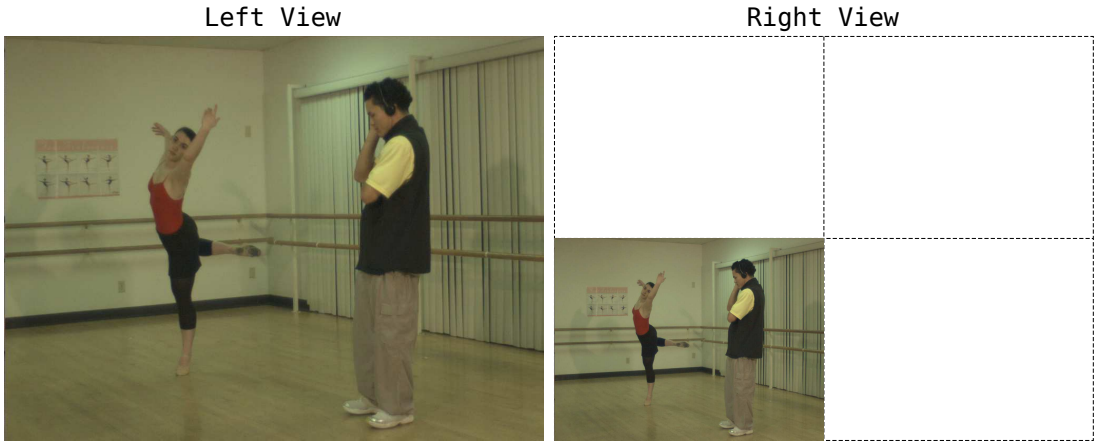


Figure 3.2: Downsampling of the right view for MRSC ("ballet" sequence is used by the courtesy of Interactive Visual Media Group at Microsoft Research)

Due to its low complexity, CSV is expected to be an applicable and promising 3D video representation for mobile platforms, and it forms the focus of this thesis.

3.2 Video plus Depth (V+D)

In the video plus depth (V+D), a 3D scene is represented by a color video and an associated depth map data. The color video is just like any views of CSV, and the depth map is a monochromatic video, with a luma component only. In the depth

map, the near objects appear brighter, whereas far objects appear darker. The V+D representation is illustrated in Figure 3.3.

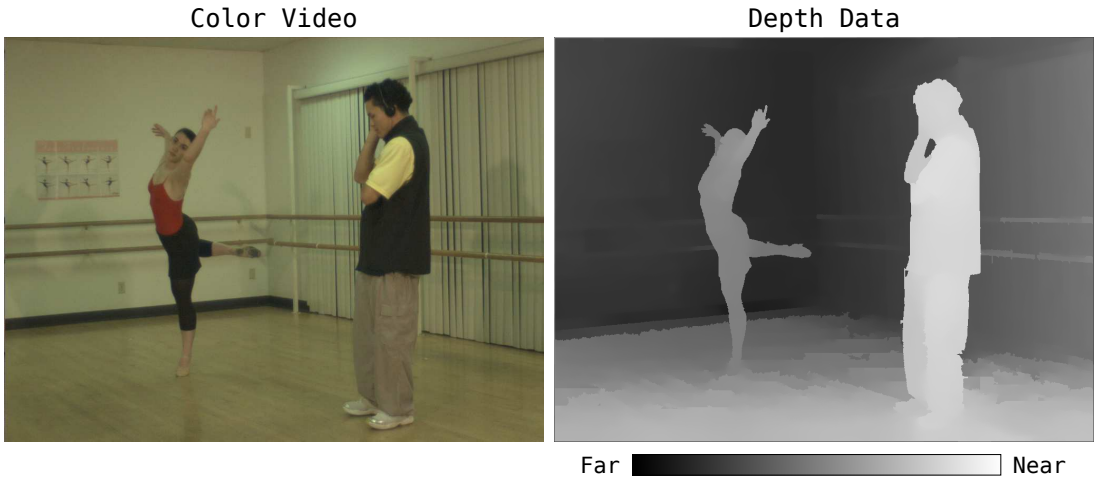


Figure 3.3: V+D representation ("ballet" sequence is used by the courtesy of Interactive Visual Media Group at Microsoft Research and the associated depth data of the sequence is generated for the research provided in [18])

V+D data can be coded much like a 2D video, where the color video can be fed as the input to a 2D video codec, like H.264/AVC; and the depth map can be separately coded by feeding into the luma channel of the same codec.

This representation requires an additional processing step at the decoder for rendering a stereo video pair from the color video and the associated depth map, by using the camera geometry information [22]. Also an additional processing is required at the encoder side, to generate the depth map possibly from a multiview color video data. As these steps may be demanding depending on the algorithms selected, its suitability for mobile platforms need to be investigated. This video representation is also considered as a possible choice for mobile platforms in the 3DPhone project and is being investigated by Fraunhofer HHI.

3.3 Multiview Video plus Depth (MVD)

Multiview video plus depth (MVD) representation is an extension of V+D. It features more than two views, with a color video and an associated depth map for each of the

views. Therefore it provides a number of different viewpoints to the observer and for this reason is mostly used for 3D television or free viewpoint video applications. Figure 3.4 provides an illustration of the MVD representation.

As in the case of V+D, for this representation, the depth map has to be generated for each view at the encoder side. Also at the decoder side, depending on the application, additional virtual views might need to be rendered. There are proposed algorithms for generating and coding MVD data [23], [24]; however they require highly demanding processing steps, making this representation a poor candidate for mobile platforms.

3.4 Layered Depth Video (LDV)

Layered depth video is also an extension of V+D. In this representation, again there is a color video and an associated depth map. However, in LDV there also exists an additional component called the *background layer* and another additional depth map component associated to it. As its name implies, the background layer provides the color video content that is covered by the foreground objects in the color video component of LDV.

LDV is also a good candidate for 3D television and free viewpoint applications as it allows rendering of several different virtual views. An illustration of LDV representation is provided in Figure 3.5. Algorithms for rendering virtual views from LDV data, generating LDV data out of MDV data and coding LDV data are proposed in [18] and [25], but these algorithms require high computational power, making LDV also a poor candidate for mobile platforms.

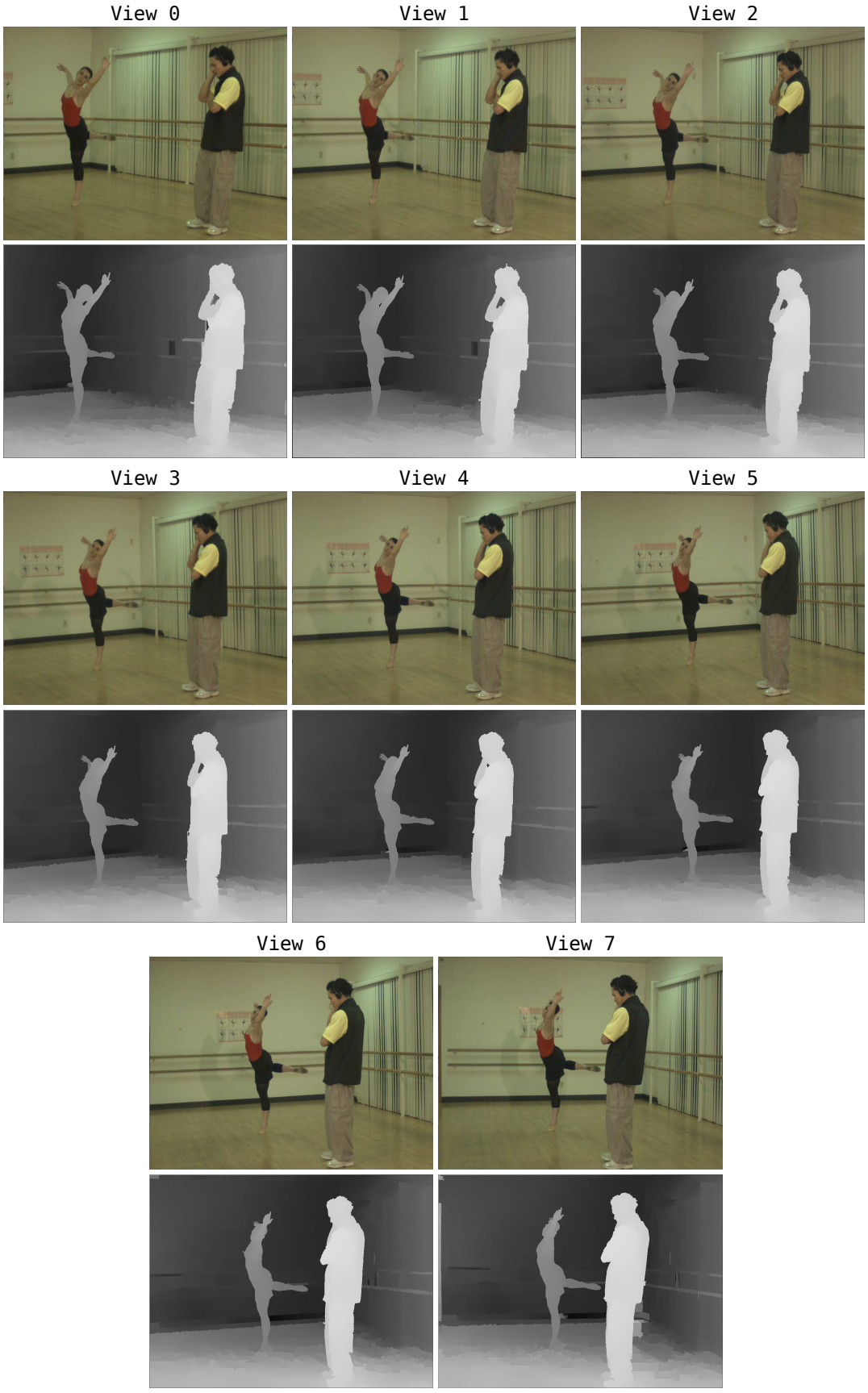


Figure 3.4: Multiview Video Plus Depth ("ballet" sequence is used by the courtesy of Interactive Visual Media Group at Microsoft Research and the associated depth data of the sequence is generated for the research provided in [18])

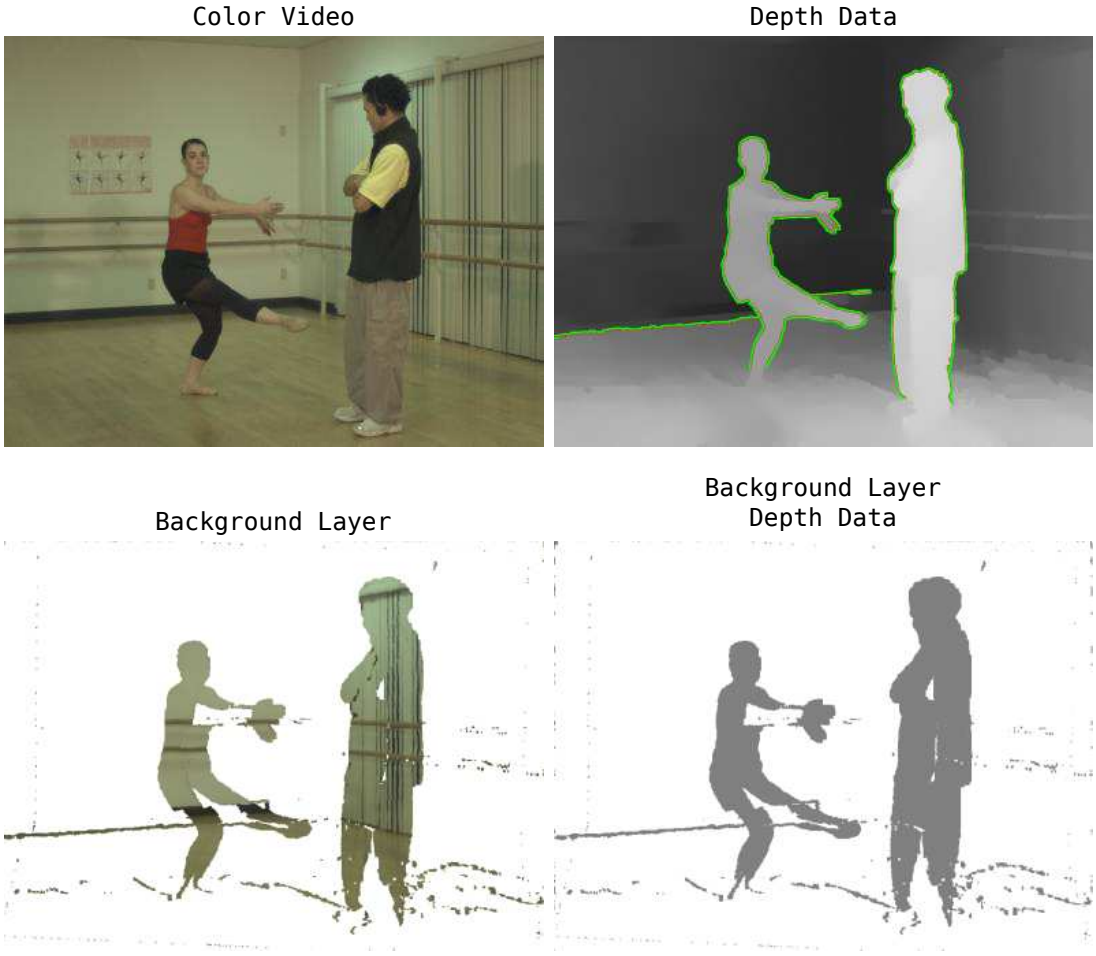


Figure 3.5: LDV representation. From A. Smolic, K. Müller, P. Merkle, P. Kauff, and T. Wiegand, "An Overview of available and emerging 3D video formats and depth enhanced stereo as efficient generic solution," in *Proc. PCS 2009, Picture Coding Symposium,* May 2009. [26] Reprinted with permission.

Chapter 4

MIXED-RESOLUTION STEREOSCOPIC CODING (MRSC)

4.1 Description of Work

There are a few alternative methods for coding stereoscopic video. Simulcast, multi-view (MVC) and mixed-resolution stereo coding (MRSC) are among them and are the ones considered primarily as possible solutions for mobile applications. In simulcast coding, both of the views are coded as two completely independent 2D videos (with no referencing between views). It is exactly the same as coding the two views of the 3D video in two separate steps with a conventional 2D video codec. This method yields the highest bit rate for a 3D video compared to the other solutions, but is the least complex. The MVC differs from simulcast coding since it allows referencing between the two views. In most cases, MVC outperforms simulcast coding, yet requires more computation as expected [19]. Its computational complexity is directly related to the complexity of the referencing scheme between the views. On the other hand, the MRSC method tries to take advantage of the human visual system in order to

reduce the bit rate, and is potentially a very promising method for the mobile applications [12]. MRSC depends on the *binocular suppression theory*, which implies that the overall 3D perception quality is dominated by the highest quality view of a stereo pair [12],[27],[28],[29],[30]. Therefore it is possible to downsample one of the views (and upsample back to the original resolution at the decoder) and code different views with different resolutions, without loosing much from the overall 3D perception quality. For MRSC, there also exist some proposed algorithms allowing inter-view prediction [27],[31],[32].

In this chapter, performances of these coding methods are compared and their usability for mobile platforms are investigated. Note that the MRSC solutions with inter-view prediction are not investigated through this work; instead the two views of MRSC are coded in a simulcast manner. Thus, for the sake of simplicity, in this chapter MRSC without inter-view prediction is referred directly as MRSC. The experimental results related with MRSC are provided and discussed in the following section.

4.2 Performance Analysis

4.2.1 Software Environment

For the experiments the Joint Multiview Video Model (JMVM) software is used. It is the reference software for the Multiview Video Coding (MVC) project of the Joint Video Team (JVT) of the ISO/IEC Moving Pictures Experts Group (MPEG) and the ITU-T Video Coding Experts Group (VCEG) [33]. It is written in C++ and includes about 100000 lines of code. The initial commit of the software to the CVS server it is maintained in, is on September 21, 2006; and the last commit is on November 4, 2008. The details of how to access this CVS server is provided in [33]. The experiments are conducted with the version 8.1 of the JMVM software, which is the latest version as of August 12, 2009.

As downsampling operation is the core of MRSC, it is also necessary to provide the details of downsampling steps used for the tests. For all the downsampling operations conducted in this section, the bundled downsampling tool of the JMVM software (DownConvertStatic) is used and the User's Manual of the tool is available in [34]. In the default mode, this tool features seven different 12-tap downsampling filters, and the decision of which filter is going to be used depends on the scaling ratio. These filters are defined by JVT within [35] and they are based on the Sine-windowed Sinc-function, which can be represented with the following formula:

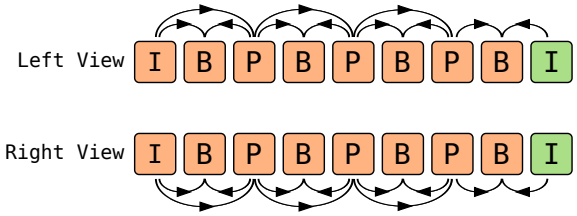
$$f(x) = \begin{cases} \frac{\sin\left(\pi \frac{x}{D}\right)}{\pi \frac{x}{D}} \cdot \sin\left(\frac{\pi}{2}\left(1 + \frac{x}{N \cdot D}\right)\right) & |x| < N \cdot D \\ 0 & \text{otherwise} \end{cases}$$

where D is the decimation parameter and N represents the number of lobes for the Sinc function on each side. For the filters of the DownConvertStatic tool, the variable N is fixed to 3, and the variable D gets chosen according to the scaling ratio. With this software and the implemented filters, any scaling ratio is allowed and can also be different in horizontal and vertical directions [34]. For further details of the filters please refer to [35].

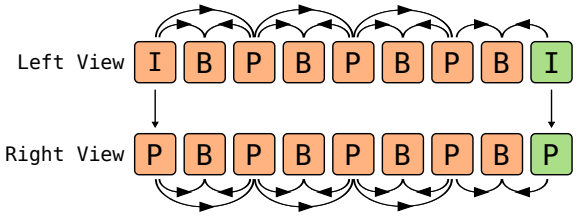
4.2.2 Experimental Results

In the experiments we used the least complex coding scheme that the software allows since the codec will be implemented on a mobile device. The coding schemes for simulcast and MVC are provided in Figure 4.1. In this figure, the arrows are directed towards the predicted frames from the frames used as references. For MRSC, the scheme is just the same as the simulcast, only the right view is downsampled to one fourth of the original resolution (one half in each direction).

Initially, to compare the performances of simulcast and MVC with each other, three test sequences of different complexities are chosen and they are coded with JMVM with the configuration file given in Figure A.1. Through the experiments, the only variable



(a) Simulcast



(b) MVC

Figure 4.1: Coding schemes

changed in this configuration file is the basis *quantization parameter* (QP) for a coding method (note that both views have to be coded with the same QP due to software restrictions). With this, for both of the methods, various bit rates are achieved and the corresponding PSNR values are recorded. The overall PSNR values of the 3D videos are calculated by averaging over the individual PSNR values of the two views. From the results of these experiments, MVC achieves less than 0.5 dB gain in quality compared to simulcast. Due to the very simple prediction scheme we used, this is reasonable and expected.

On the other hand, for the MRSC there is no objective quality measure like PSNR for the overall 3D perception quality, so it is not possible to compare MRSC to simulcast or MVC using a mathematical formula. However, to gain some insight about its potential, we may assume that the overall PSNR will be highly dependent to the PSNR of the left view (high resolution view). Therefore, the PSNR values of the left-view may yield meaningful results as a preliminary study.

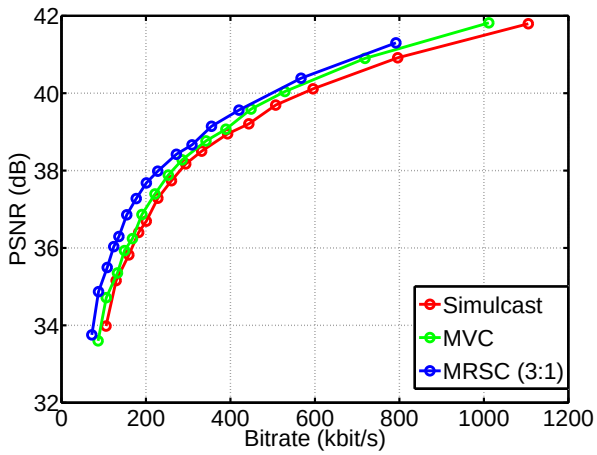
For comparing the performance of MRSC with the performances of simulcast and MVC, we assume that the overall PSNR value for an MRSC coded video is exactly the same with the PSNR value of its left view. Since we do not include the quality of the right view in the calculation of the overall PSNR value, we are free to choose the

right-view to be quantized as coarse as possible to achieve lower total bit rates with the same overall PSNR value. However, this would lead to completely wrong predictions since having a very coarse right-view should result in considerable degradation in the overall 3D perception quality. Hence, for the bit rates of the MRSC coded videos, we fix the bit rate ratio of the left-view to right-view to 3:1, which we expect it to perform comparable to simulcast and MVC methods with an educated guess. But this ratio is also a variable and the one that would yield the best 3D perception quality needs to be determined with further subjective tests. With the bit rate ratio fixed to 3:1, for these comparison tests the total bit rates of MRSC videos are also directly calculated from the left view just by multiplying its bit rate with a coefficient of $\frac{4}{3}$. Since the screen of the mobile device will support a low resolution, we used downsampled versions (640×352) of the original sequences for our tests. The comparison of all these methods over Bullinger, Car and Hands sequences are shown in Figure 4.2.

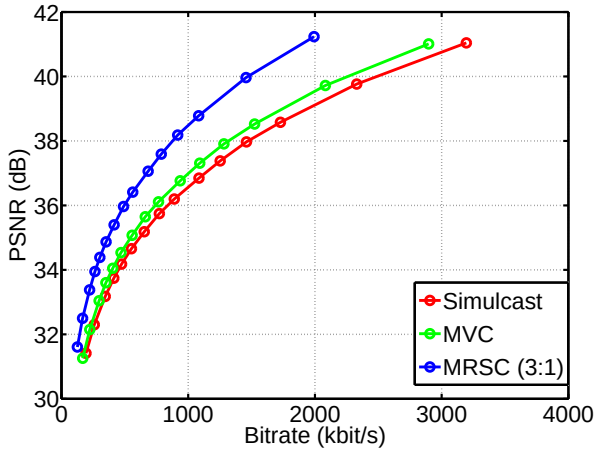
Assuming the explained calculation of PSNR for MRSC video quality is valid, it is possible to conclude that MRSC method is promising for mobile applications. However, this assumption is not correct since it is expected to have some additional effect from the right view on the overall 3D perception quality, as well. Hence, some subjective tests needed to be conducted before concluding the performance of this method.

Subjective Tests

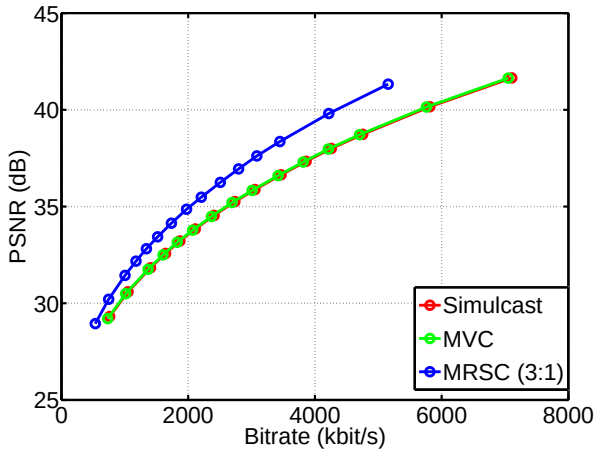
In order to understand the performance of MRSC over conventional stereoscopic coding methods, a subjective test is conducted on 16 people. For the tests a Sharp Actius AL3DU laptop with an embedded 15" 3D parallax barrier based LCD screen is used. For diversity in the complexities of the test sequences, Bullinger, Car and Hands sequences are selected to be used. The aim of these tests was to find the best perceptual performance among the following coding methods:



(a) Bullinger



(b) Car



(c) Hands

Figure 4.2: Rate-distortion comparisons for simulcast, MVC, and MRSC

A	Simulcast	Full Resolution Coding (FRC)
B	Multiview	FRC
C	Simulcast	MRSC - 2:1 bit rate ratio between views (2:1)
D	Simulcast	MRSC(3:1)
E	Simulcast	MRSC(4:1)

To generate the MRSC data, we also simulcast coded the downsampled versions of the selected sequences using JMVM with almost the same configuration file as A.1, just with one fourth of the original resolution. Then we combined the full-resolution simulcast coded left-views with quarter-resolution simulcast coded right-views so that they would have the predetermined bit rate ratios between their views and satisfy the total bit rate constraints (i.e. for a predetermined ratio of 3:1 and a total bit rate of 4 kbit/s, a 3 kbit/s left-view is matched with the right view having a bit rate closest to 1 kbit/s).

Through the subjective tests, the *A-B preference* test method is used. For this test method, the procedure for one comparison test is as follows: The videos to be compared are shown to the observers one after the other (twice each as 1,2,1,2). Just before the videos, the corresponding labels of the videos are shown with a white font over a black background, to indicate which video is going to be played. For example, if it is the fourth comparison test, before first video *4A* and before second video *4B* is shown. Once the videos are played to the observers, they are asked for their preferences. The preference choices are labeled as $A/B/Same$; where A corresponds to the first video shown and B corresponds to the second one. The observers are explicitly asked to select *Same* if and only if they can not perceive any difference in the overall 3D perception quality of the compared videos at all. After they indicate their choices, they move on to the next comparison test. The timing scale of one comparison test is provided in Figure 4.3:

In our case, the videos to be compared have the same content, but they are coded with different methods. For each of the selected test sequences, the sequence is coded with the methods to be compared resulting in five different compressions of the same video. In order to compare all the methods with each other, the videos are coupled to span all possible combinations ({A,D}, {B,E}, {B,C}, {C,E}, {A,B}, {B,D}, {D,E},

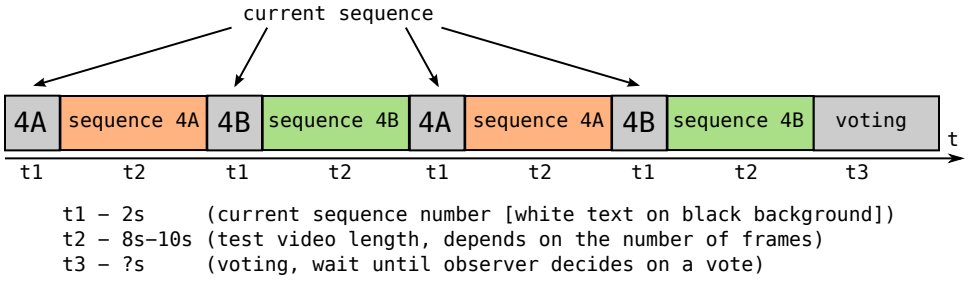


Figure 4.3: Timing of one comparison test [36]

{C,D}, {A,E}, {A,C}). Therefore for each test sequence ten different comparison tests are conducted.

In order to understand the effect of MRSC at different bit rates, for each of the test sequences, two bit rate levels are determined. These levels are labeled as *HIGH* and *LOW*, indicating the relative amount of bit rates of the videos to be compared. Since the bit rate is dependent on the video content, these levels are varied across the test sequences. The *HIGH* and *LOW* levels for a sequence are determined by taking the simulcast coded video with QP30 (quantization parameter) and QP36 as reference points. For example, the total bit rates of simulcast coded Car sequence with QP30 and QP36 are 1253 kbit/s and 474 kbit/s respectively; and these are selected to be the *HIGH* and *LOW* bit rate levels for Car sequence. The reason for selecting the bit rate levels by taking QPs as reference points is that the JMVM software does not have an option to determine the desired bit rate and the overall bit rates get determined only by adjusting the QPs. Since the QPs can be changed by a step of one, it was not possible to achieve exactly the same bit rate for each of the coding methods to be compared. Because of this, the QPs of the other coding methods are adjusted so that the total bit rates for all the methods to be compared are as close to these levels as possible. For example, for the Car sequence and the *HIGH* bit rate level, we are required to allocate 835 ($1253\times\frac{2}{3}$) kbit/s for the left view and 418 ($1253\times\frac{1}{3}$) kbit/s for the right view for the MRSC(2:1) method ideally. However, by adjusting the QPs, the closest bit rates we can achieve for the right and left views are 812 and 443 kbit/s respectively, so we choose the QPs achieving these bit rates to be used for the MRSC(2:1) method. In Table 4.1, test video parameters (bit rates of each view & QPs) that are used in the subjective tests are given. Another option for adjusting

the bit rates could be keeping the QPs constant and adjusting the frame rates of the coded videos, but in our subjective tests we fixed the frame rates of the coded videos to the original frame rates of the used sequences and have not considered this option in our tests.

With all these parameters determined, for each test sequence and for each bit rate level the planned ten comparison tests are conducted. For evaluating the preferences of the observers, a fixture-like model is used. In this model, the comparison tests for one test sequence and for one bit rate level form a tournament, making 6 tournaments in total. In these tournaments, each comparison test is considered as a game and the preferred video is assumed to win against the other. When a video wins a game, it is given two points. If the compared videos get selected to be *Same*, the match results with a draw and both videos are given one point each. According to this model, the preferences of different observers are taken as different results of the same tournaments. Therefore, the final results of the tournaments are calculated by averaging over all 16 observers. The subjective test results for the model explained are listed in Table 4.2.

According to the results listed in Table 4.2, at first sight, MVC can be said to be the most voted for most of the sequences and for both bit rate levels. However, for each section, all the methods have very close vote averages and their standard deviations are high. So, it is not possible to conclude a global ranking among all the methods. When the results are investigated in more detail, even some unexpected and conflicting results exist. For example for low bit rate Bullinger sequence MRSC(3:1) and MRSC(4:1) use the same left views while MRSC(4:1) uses a lower PSNR right view. So it is expected to have MRSC(3:1) to be voted better than MRSC(4:1), however observers favored MRSC(4:1) according to the results. Such a situation also exist for low bit rate Hands sequence for MRSC(3:1) and MRSC(4:1) methods. This leads us to think that it may have been difficult for the observers to assess the 3D perception quality when the explained set up and procedures are used.

Therefore, the results of these tests are left as inconclusive and a cross-check of the same tests with different observers and a different 3D display is conducted. The reason for changing the display was that on the previous tests, observers informed us

Table 4.1: Test video parameters (bit rates and QPs of each view)

Videos \ Methods		A		B		C		D		E	
		Left (kbit/s)	Right (kbit/s)	Left (kbit/s)	Right (kbit/s)	Left (kbit/s)	Right (kbit/s)	Left (kbit/s)	Right (kbit/s)	Left (kbit/s)	Right (kbit/s)
Bullinger	HIGH Bit Rate	232 (QP30)	212 (QP30)	266 (QP29)	182 (QP29)	315 (QP28)	150 (QP24)	315 (QP28)	117 (QP26)	371 (QP27)	93 (QP28)
		Total: 444		Total: 448		Total: 465		Total: 432		Total: 464	
	LOW Bit Rate	102 (QP36)	99 (QP36)	116 (QP35)	75 (QP35)	133 (QP34)	67 (QP31)	151 (QP33)	52 (QP33)	151 (QP33)	41 (QP35)
		Total: 201		Total: 191		Total: 200		Total: 203		Total: 192	
Car	HIGH Bit Rate	591 (QP30)	662 (QP30)	688 (QP29)	594 (QP29)	812 (QP28)	443 (QP24)	955 (QP27)	331 (QP26)	955 (QP27)	251 (QP28)
		Total: 1253		Total: 1282		Total: 1255		Total: 1286		Total: 1206	
	LOW Bit Rate	227 (QP36)	247 (QP36)	264 (QP35)	206 (QP35)	312 (QP34)	164 (QP31)	367 (QP33)	119 (QP33)	367 (QP33)	102 (QP34)
		Total: 474		Total: 470		Total: 476		Total: 486		Total: 469	
Hands	HIGH Bit Rate	2097 (QP30)	1765 (QP30)	2097 (QP30)	1718 (QP30)	2585 (QP28)	1305 (QP23)	2878 (QP27)	983 (QP26)	3164 (QP26)	802 (QP28)
		Total: 3862		Total: 3815		Total: 3890		Total: 3861		Total: 3966	
	LOW Bit Rate	1005 (QP36)	863 (QP36)	1005 (QP36)	821 (QP36)	1302 (QP34)	645 (QP30)	1481 (QP33)	448 (QP33)	1481 (QP33)	391 (QP34)
		Total: 1868		Total: 1826		Total: 1947		Total: 1929		Total: 1872	

Table 4.2: Subjective test results - with Sharp Actius AL3DU laptop

High Bit Rate			Low Bit Rate		
Bullinger			Bullinger		
Coding Methods	Mean	Std.Dev.	Coding Methods	Mean	Std.Dev.
Simulcast - FRC	4.19	1.38	Simulcast - FRC	4.56	1.75
MVC - FRC	4.94	1.06	MVC - FRC	4.38	2.06
MRSC(2:1)	4.25	2.11	MRSC(2:1)	4.06	0.85
MRSC(3:1)	3.25	1.24	MRSC(3:1)	3.38	1.96
MRSC(4:1)	3.38	1.36	MRSC(4:1)	3.50	1.59
Car			Car		
Coding Methods	Mean	Std.Dev.	Coding Methods	Mean	Std.Dev.
Simulcast - FRC	4.81	1.28	Simulcast - FRC	3.75	1.53
MVC - FRC	4.88	1.67	MVC - FRC	4.50	1.90
MRSC(2:1)	3.25	1.91	MRSC(2:1)	3.75	1.34
MRSC(3:1)	3.25	1.69	MRSC(3:1)	4.06	1.53
MRSC(4:1)	3.81	1.38	MRSC(4:1)	3.94	1.57
Hands			Hands		
Coding Methods	Mean	Std.Dev.	Coding Methods	Mean	Std.Dev.
Simulcast - FRC	4.00	1.10	Simulcast - FRC	3.88	1.75
MVC - FRC	4.00	1.75	MVC - FRC	4.69	2.02
MRSC(2:1)	3.94	1.77	MRSC(2:1)	4.50	1.93
MRSC(3:1)	3.25	1.69	MRSC(3:1)	3.44	1.36
MRSC(4:1)	4.81	1.11	MRSC(4:1)	3.50	1.59

about the difficulty to sense and watch 3D videos on the parallax barrier based screen. Therefore in these tests a Miracube G320S monitor is used and the observers wore special polarized glasses for the 3D sensation. With this, it is expected to reduce the effect of the type of display on the votes. The tests are prepared by us, and then conducted by Fraunhofer HHI, which is a partner of the 3DPhone project as well, as the display belonged to them. Test results averaged over seven observers are listed in Table 4.3.

From these tests, MVC came out to be the most preferred again, almost over all sequences and bit rates, which supports the tests conducted with the Sharp 3D laptop. However this time, the rankings of the rest of the methods are also more consistent over different sequences, and it is possible to derive some conclusions.

For Bullinger and Car sequences (low and medium depth and detail), full resolution coding methods (both simulcast and MVC) are preferred over the compared MRSC

Table 4.3: Subjective test results - with Miracube G320S monitor

High Bit Rate			Low Bit Rate		
Bullinger			Bullinger		
Coding Methods	Mean	Std.Dev.	Coding Methods	Mean	Std.Dev.
Simulcast - FRC	4.57	2.51	Simulcast - FRC	4.43	1.13
MVC - FRC	6.00	2.58	MVC - FRC	5.43	2.76
MRSC(2:1)	3.71	2.43	MRSC(2:1)	5.14	1.57
MRSC(3:1)	2.29	1.80	MRSC(3:1)	3.29	1.50
MRSC(4:1)	3.43	1.90	MRSC(4:1)	1.71	1.80
Car			Car		
Coding Methods	Mean	Std.Dev.	Coding Methods	Mean	Std.Dev.
Simulcast - FRC	5.71	1.38	Simulcast - FRC	4.71	1.50
MVC - FRC	6.14	2.04	MVC - FRC	5.29	2.98
MRSC(2:1)	3.19	1.50	MRSC(2:1)	5.14	2.27
MRSC(3:1)	2.86	2.27	MRSC(3:1)	2.43	1.62
MRSC(4:1)	2.00	1.63	MRSC(4:1)	2.43	1.99
Hands			Hands		
Coding Methods	Mean	Std.Dev.	Coding Methods	Mean	Std.Dev.
Simulcast - FRC	3.71	2.69	Simulcast - FRC	4.00	2.00
MVC - FRC	4.29	2.14	MVC - FRC	3.71	2.69
MRSC(2:1)	4.14	1.46	MRSC(2:1)	4.29	2.14
MRSC(3:1)	3.86	2.48	MRSC(3:1)	3.43	0.98
MRSC(4:1)	4.00	3.17	MRSC(4:1)	4.57	2.23

methods. On the other hand for low bit rates, MRSC(2:1) method seem to perform better than it did for high bit rates, and got selected to perform better than simulcast and close to MVC.

For the Hands sequence (high depth and detail), at high bit rate MVC is again selected to be the best and this is consistent with the rest of the results. However the votes for each of the methods are very close to each other and deriving any other conclusion out of these results would be biased.

On the other hand, the results for low bit rate Hands sequence came out to be inconsistent again. MRSC(4:1) outperformed all of the methods leaving MRSC(3:1) to be last, which has a higher PSNR value than MRSC(4:1). So, for these results it is not possible to derive solid conclusions.

Lastly, when MRSC methods are compared to each other, there is a general inclination towards MRSC(2:1) over the rest. When MRSC(3:1) and MRSC(4:1) are compared to each other, they usually performed very close and there is no obvious preference of one over the other.

Summing up the findings of the subjective tests, MRSC(2:1) method or perhaps a lower ratio MRSC still can be promising for mobile applications since it is expected to deal with low bit rate videos most of the time. It is worth mentioning again, that in this chapter we only considered the MRSC methods without inter-view prediction. Since MVC came out to be the best among the compared methods and MRSC(2:1) performed better than simulcast coding in some cases, MRSC with inter-view prediction might outperform MVC in some cases. Hence, deploying both MVC and MRSC decoding features on the mobile device can be a promising approach. Additionally, the test results are found to be highly dependent on the used display. Therefore, MRSC requires further investigation, both in terms of 3D perception quality and computational complexity, when the first mobile hardware prototype is ready with an embedded 3D display. Until then, investigations on the video decoding performance of the selected mobile platform are continued with the MVC method.

Chapter 5

IMPLEMENTATION AND TESTING OF MVC ON THE MOBILE PLATFORM

5.1 Hardware Platform

As the mobile hardware device the "Logic Product Development"s, "ZOOM^{TM} OMAP34x^{TM} Mobile Development Platform (MDK)" is used. The ZOOM^{TM} OMAP34x^{TM} MDK features the following hardware specifications [37],[38]. Note that only the specifications that contribute to the understanding of the thesis are provided in the following list:

- Texas Instruments OMAP3430^{TM} System-on-a-chip (SoC)
  - 550MHz ARM® Cortex^{TM}-A8 processor (main processing unit)
  - 400MHz TMS320C64x+ digital signal processor (additional processor for imaging, video and audio algorithms)
  - PowerVR SGX530^{TM} GPU (dedicated graphics processor)
- 128MB Mobile DDR memory - 32-bit memory bus with 166MHz clock speed

- 256MB NAND memory - 16-bit memory bus with 166MHz clock speed
- 3.7" VGA TFT touchscreen display
- 10/100 BASE-T ethernet port
- MicroSD/MMC card slot
- Serial port

The OMAP34x^{TM} MDK runs a Linux distribution called "Poky Linux" as the operating system with a kernel version "2.6.24-7-omap1-arm2". The boot loader, kernel and the file system get installed on a MicroSD card with the help of a personal computer (PC), and the MDK boots up from the MicroSD card. Note that storing the kernel and boot loader on the NAND memory of the MDK and accessing the file system, which can be stored on the personal computer, via ethernet port is also possible. Once the MDK is booted, the communication with the MDK is established over the serial port, and the user can send keystrokes from the keyboard of the PC directly to the MDK. Figure 5.1 shows the MDK running a notepad application.

Figure 5.1: The ZOOM^{TM} OMAP34x^{TM} MDK runnning a notepad application.

5.2 Preliminary MVC Tests on OMAP34x™ MDK

In this chapter, from now on the ARM® Cortex™-A8 processor will be referred as the MPU as an abbreviation of *main processing unit* and the TMS320C64x+ digital signal processor will be referred as DSP, for the sake of simplicity.

For multiview video coding performance on OMAP34x™ MDK, JMVM software [33] is compiled for the MPU and tested. For initial tests, four sequences with different complexities and depth are chosen and are encoded using a personal computer. Then the decoding performance of JMVM on the MPU is examined with these coded videos in order to understand the computational power of the device. The decoding performance of JMVM on the MPU for these videos are listed on Table 5.1.

Table 5.1: Decoding performance of JMVM on ARM® Cortex™-A8 processor

Sequence name	Resolution	Stereo frames per second
Bullinger	640×352	4.8
Hands	640×352	3.6
Car	640×352	3.8
Pantomime	PAL	2.2

As it can be seen from Table 5.1 the initial MVC performance tests resulted in a low number of frames per second. Therefore, to better use the hardware resources, it was decided to take advantage of the DSP, as well, while decoding the videos. To understand the capabilities of the DSP, some of the algorithms of the JMVM software are planned to be ported to run on the DSP. To find out which algorithms would be the most beneficial to port, first a profiling on the JMVM software is done and the most computationally intensive algorithms are found. Then, some of these most demanding algorithms are selected and implemented on the DSP, and their performances on the DSP are examined. The profiling information, implementation steps and the experimental results are provided in the following sections.

5.3 Multiview Codec (JMVM Software) Profiling

To profile the JMVM software, the profiling information is generated for each of the selected videos. The most demanding functions of JMVM and their profiling results for Bullinger, Car, Hands and Pantomime videos are provided in the Tables B.1, B.2, B.3, B.4, respectively.

According to the profiling results, the most demanding functions are found to be consistent across different test videos. Since JMVM software is based on the H.264/MPEG-4 AVC standard, JMVM's most computationally intensive functions are found to be the quantization operations, transforms and filters, as expected.

To test the gain with the DSP, as mentioned before, a few of these functions are selected to be ported to and tested on the DSP. The names of the selected functions are "MbDecoder::xScale4x4Block", "LoopFilter::xFilter", "QuarterPelFilter::xPredElse" and "Transform::xInvTransform4x4Blk", and their codes are provided in Appendix C. The reason behind this selection was to see the DSP performance on different types of operations. The characteristics of the selected functions are listed in Table 5.2.

Table 5.2: Characteristics of the functions selected to be ported to DSP

Function name	Characteristics
MbDecoder::xScale4x4Block	Dequantization operation on a 4x4 macroblock *Multiplication intensive*
LoopFilter::xFilter	Adaptive Deblocking Filter *Nonlinear filtering*
QuarterPelFilter::xPredElse	Quarter-pel Filtering operation *Convolution type of filtering*
Transform::xInvTransform4x4Blk	IDCT on a 4x4 macroblock *Transform*

5.4 DSP Programming

The TMS320C64x+ is a fixed-point DSP using the VelociTI^{TM}, which is a high-performance, advanced very-long-instruction-word (VLIW) architecture designed by Texas Instruments [39]. By design, this architecture allows the execution of up to eight instructions in parallel, making it efficient for signal processing tasks such as filtering and transform like operations.

Because of its architecture, the DSP uses a technique called *software pipelining* for increasing the throughput of the *for loops*. This is achieved by scheduling the instructions of a *for loop* so that multiple iterations of the loop execute in parallel [39]. Figure 5.2 illustrates a software pipelined loop. In this figure, the stages of a loop are represented by A,B,C,D, and E followed by a number indicating the iteration count. In the colored area all five stages of the loop execute in parallel.

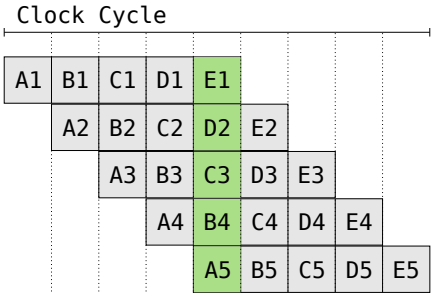


Figure 5.2: Software pipelined loop [39]

Therefore the DSP is capable of executing some types of algorithms faster than they would on the MPU. The implementation steps of a program running on the DSP are provided in the next section.

5.4.1 Implementation

On the OMAP34x^{TM} MDK, the developer does not have direct access to the DSP. Instead, the MPU has to be used for controlling it. There is a separate operating system running on the DSP, which is called the *DSP/BIOS^{TM}*. Therefore, on the Linux operating system there is a driver called *DSP/BIOS^{TM} Bridge (or DSP Bridge)*

and it maintains the communication link between the MPU and the DSP operating systems with an API available to the developers. The DSP Bridge provides the ability to:

- "initiate signal processing tasks on the DSP,
- exchange messages with the DSP tasks,
- stream data buffers to and from DSP tasks,
- dynamically map external memory into the DSP virtual space,
- pause, resume, and delete DSP tasks"

to the MPU [40],[41]. Hence, for running an application on the DSP, a DSP side program and an MPU side program need to be written and compiled specifically for the corresponding operating systems. The MPU side of a program gets compiled with an ARM® version of the GNU Compiler Collection (GCC), and this step is straightforward and is similar to compiling a C program with GCC on a PC. On the other hand the DSP side of the program gets compiled with "C6000 Code Generation Tools" and the compilation step is more complicated as it requires some preliminary configurations.

DSP side of a program

On the DSP side, all the codes coming from different applications are compiled into one *base image*, which is loaded by the DSP Bridge once the OMAP34x™ MDK boots. Since all the codes coming from different applications are compiled into one image, DSP Bridge uses an abstraction to address different groups of codes. In this abstraction, blocks of code belonging to the same application are called a *node*. Each node has a unique identifier number, and can be addressed through the DSP Bridge. Additionally, DSP Bridge allows loading new nodes dynamically even after the /textslbase image is loaded; however, this feature is not used for the tests of this chapter, therefore its further discussion is omitted. There are three different types of nodes which are "Task nodes", "XDAIS Socket nodes" and "Device nodes". The Task node is the most basic processing element of the DSP Bridge, and it runs as an independent execution thread in the DSP/BIOS™ . Since Task node is the only type that is used for the

implementation parts of the thesis, details of other nodes are omitted. The details of the other types of nodes are available in [41]. Note that from now on "node" will be used to refer to "Task node" for the sake of simplicity.

The DSP Bridge features the *Resource Manager (RM) Server* which handles the commands issued by the MPU. The main task of RM is to create, execute and delete nodes on the DSP. Thus, RM requires three C-callable functions, which are *create*, *execute*, and *delete* functions, to be implemented for a node for each of the corresponding phase. In the *create* phase of a node, depending on the features of the node, there exist the steps for allocating the necessary resources. The *delete* phase serves the purpose of deallocating all the resources, that have been allocated during the *create* phase. In the *execute* phase, the real processing steps take place. The *execute* function basically includes an infinite loop that runs until a special *break* command is sent from the MPU. Inside the loop, it checks for various predetermined commands issued by the MPU and executes the corresponding section of the code. The details of the *execute* functions of our applications are going to be discussed later.

The communication methods between the MPU and the DSP, supported by the DSP Bridge, are *messaging* and *streaming*. *Messaging* is used for exchanging short and fixed length packets, whereas *streaming* is for exchanging large amounts of data. Additionally, there exists another communication method, which is called "dynamic memory mapping" (DMM), as an alternative to *streaming*. In this method, a shared memory between the MPU and the DSP gets allocated, allowing the exchange of data between the two processors.

For our tests, the messaging method is used for exchanging handshaking commands; and the DMM is used to share the data to be processed between the MPU and the DSP. Since DMM is used instead of streaming, the *create* and *delete* functions of our nodes did not allocate or deallocate any resources for our nodes. For the *execute* stage, inside the infinite loop, the conditional statements checking for *break*, *DMM setup*, and *execute* commands from the MPU are inserted. When the *break* command is received, the *execute* stage ends and the node gets deleted by RM. When the DMM setup command is received, the already allocated shared memory address (allocation

is done at the MPU side) is passed to the node. Lastly, when the execute command is received the desired algorithm is executed on the data located in the shared memory (at the MPU side, the data has to be written on the shared memory location prior to sending this command). One note is that, for synchronization of MPU and DSP, at the end of the execution of the desired algorithm, this time DSP sends a message back to MPU informing that the data processing is completed. The MPU waits for this message from the DSP, before continuing its execution.

For compiling the *base image* and including a specific node in the *base image*, a configuration file has to be prepared for the node and then included inside the configuration file of the /textslbase image. Figure 5.3 shows the configuration file of the node prepared for "MbDecoder::xScale4x4Block" (testXSca4x4.tci).

```
// add instance NODE.testXSca4x4_canbal
testXSca4x4_canbal = bridge.NODE.create("testXSca4x4_canbal");

testXSca4x4_canbal.uuid = "28BA464F_9C3E_484E_990F_48305B183801";
testXSca4x4_canbal.name = "TESTXSCA4X4_CANBAL";
testXSca4x4_canbal.nodeType = "TASKNODE";
testXSca4x4_canbal.stackSize = 1024;
testXSca4x4_canbal.messageDepth = 3;
testXSca4x4_canbal.timeout = 1000;
testXSca4x4_canbal.createFxn = "TESTXSCA4X4_CANBAL_create";
testXSca4x4_canbal.executeFxn = "TESTXSCA4X4_CANBAL_execute";
testXSca4x4_canbal.deleteFxn = "TESTXSCA4X4_CANBAL_delete";

if (dynamicNode) {
    testXSca4x4_canbal.loadType = "Dynamic";
    testXSca4x4_canbal.splitPhases = splitPhases;
    if (chipType != "2430" && chipType != "3430") {
            testXSca4x4_canbal.createCodeSeg = bridge.DYN_EXTERNAL;
            testXSca4x4_canbal.createDataSeg = bridge.DYN_EXTERNAL;
            testXSca4x4_canbal.executeCodeSeg = bridge.DYN_EXTERNAL;
            testXSca4x4_canbal.executeDataSeg = bridge.DYN_EXTERNAL;
            testXSca4x4_canbal.deleteCodeSeg = bridge.DYN_EXTERNAL;
            testXSca4x4_canbal.deleteDataSeg = bridge.DYN_EXTERNAL;
    }
}
```

Figure 5.3: DSP node configuration file for "MbDecoder::xScale4x4Block".

To include the configured node in the *base image*, also a line has to be added to the *base image* configuration file (ddspbase.tci) as follows:
`utils.importFile("testXSca4x4.tci");` . When the configuration file given in Figure 5.3 is examined, in this file the identification number of the node is given as "uuid". Also the *create*, *execute* and *delete* function names specific to this node are listed in this file. In addition to this step, the filenames without the ".c" suffix that include those functions need to be passed to the compiler before compiling the *base image*.

This is done by including the source filenames in the build script (package.bld). Figure 5.4 shows which lines need to be added to which part of the build script.

```
{
name:    "ddspbase",
sources: [ "main", "pingCreate", "pingExecute", "pingDelete",
           "pinglib", "pingdata", "strmcopyCreate", "strmcopyExecute", "strmcopyDelete",
           "testXSca4x4Create", "testXSca4x4Execute", "testXSca4x4Delete", // this line is added
           "iscale", "scale_ti", "scale_ti_vtab", "ssktcrea", "ssktexec",
           "ssktdele", "zcmsgCreate", "zcmsgDelete", "zcmsgExecute",
           "dmmcopyCreate", "dmmcopyDelete", "dmmcopyExecute"]
}
```

Figure 5.4: Addition of necessary source files of "MbDecoder::xScale4x4Block" in build script.

Having included all the desired nodes in the *base image* as explained, once the *base image* is compiled and loaded, MPU can access to the node by using the DSP Bridge API functions.

MPU side of a program

The MPU side of the program is easier to compile but it needs to include some specific steps to have the DSP do the desired processing. As the API of the DSP Bridge is designed to support even more than one DSP chip, first of all the DSP needs to be attached and initialized in the MPU side of the program. After having the DSP ready, the next step is initializing a specific node of our choice. With this step, the *create* function of the DSP side of the application gets called and the RM creates an instance of the desired node. After this step, if streaming is going to be used for communication with the node, then the streams need to be initialized. However, as shared memory is used to transfer the data to the DSP for our tests, this step is simply omitted in our programs. The last step before starting the real processing is allocating the shared memory and sending its address to the DSP side of the program. In this step, for allocating the shared memory, the developer needs to use the specific DSP Bridge API functions, as this memory needs to be directly accessible by the DSP. Otherwise, when the DSP tries to access a memory location which is not allocated this way, the DSP Bridge raises an exception and the execution of the DSP node stops. After allocating the memory, it needs to be mapped to the DSP virtual space before it can be accessed

by the DSP. Once the shared memory is ready for access, its address on the DSP virtual space is sent to the DSP by messaging. After all these steps, the DSP is ready for processing and the MPU can put some data to the shared memory and send a message to the DSP to start processing. One note is that for synchronization with the DSP, MPU needs to wait for a message from the DSP indicating that it has completed processing the data, before MPU can continue its execution. These processing steps can be repeated for numerous times, without repeating the previous initialization steps. This is because the DSP side infinite loop continues to run throughout the entire runtime of the MPU side program until it receives a special *break* command from the MPU. After the MPU has all the data processed by the DSP, some additional steps are needed to clean up for the initialization steps. Shared memory needs to be freed; the streams need to be closed; the instance of the node should be deleted (by making RM call the *delete* function of the DSP side of the application) and finally DSP needs to be detached.

With these in mind, using the DSP Bridge API functions in the MPU side of the program brings an additional overhead to the program (when compared to the case that we do the data processing in the MPU). The initialization steps of the DSP requires an insignificant amount of additional time when compared to the average runtime of a program. However the memory access and messaging operations are unavoidable and required each time an algorithm is executed on the DSP. Hence, these determine the actual overhead to the overall runtime and it has to be less than the time gained by executing an algorithm on the DSP instead of the MPU. In the following section, this overhead is measured for each of the functions ported and its effect on the runtime performance is discussed.

5.4.2 Experimental Results

For testing the performance of DSP, each of the selected functions are isolated from the decoder and ported to run on the DSP. Note that, through the tests, the input data to these functions are extracted from the videos being decoded by the JMVM

decoder, so that each function is fed by the actual data it would process within the decoder.

For all of these functions, first the previously discussed communication overhead per call is calculated by running the DSP side application various times without any processing, and by averaging over the total runtime. The number of times the DSP side application is run is selected so that the communication overhead per call reaches to an asymptote. Then the processing parts of the DSP side applications are enabled and the functions are tested with 1000 different data sets coming directly from the decoded videos on both MPU and DSP. In order to minimize the effect of the communication overhead on the calculation of the processing time per call, for each of the data sets, the processing steps are repeated for 10000 times without any communication between MPU and DSP. This experimental procedure is repeated over the Bullinger, Car, Hands and Pantomime videos in order to eliminate the effect of the video content on the results. The results of these tests for each of the selected functions are listed in Tables 5.3, 5.4, 5.5, 5.6. Note that for these results the communication overheads are subtracted.

As it can be seen from the results listed in Tables 5.3, 5.4, 5.5, 5.6, when the communication overheads are neglected, porting some of these functions resulted in a significant gain. As expected, the functions with *for loops* benefit more from the DSP, whereas the functions with no *for loops* resulted in a negative gain. The reason for this is the *pipelining* capabilities of the DSP on the *for loops* that has been discussed earlier.

Table 5.3: MbDecoder::xScale4x4Block - DSP vs MPU performance comparison

Video name	DSP processing time per call (μs)	MPU processing time per call (μs)	Gain (%)
Bullinger	0.2582	0.3896	33.73
Car	0.2574	0.3876	33.59
Hands	0.2577	0.3887	33.70
Pantomime	0.2555	0.3808	32.90

Communication overhead per call = 160 μs

Table 5.4: LoopFilter::xFilter - DSP vs MPU performance comparison

Video name	DSP processing time per call (μs)	MPU processing time per call (μs)	Gain (%)
Bullinger	0.1596	0.1158	-37.82
Car	0.1627	0.1197	-35.92
Hands	0.1590	0.1130	-40.71
Pantomime	0.1605	0.1031	-55.67

Communication overhead per call = 160 μs

Table 5.5: QuarterPelFilter::xPredElse - DSP vs MPU performance comparison

Video name	DSP processing time per call (μs)	MPU processing time per call (μs)	Gain (%)
Bullinger	3.7736	9.1154	58.60
Car	4.1426	10.1162	59.05
Hands	3.7563	9.0540	58.51
Pantomime	3.8059	9.1889	58.58

Communication overhead per call = 550 μs

Table 5.6: Transform::xInvTransform4x4Blk - DSP vs MPU performance comparison

Video name	DSP processing time per call (μs)	MPU processing time per call (μs)	Gain (%)
Bullinger	0.6400	0.8339	23.25
Car	0.6395	0.8520	24.94
Hands	0.6397	0.8449	24.29
Pantomime	0.6395	0.8325	23.18

Communication overhead per call = 160 μs

To go into the details of ported functions, we need to examine the codes and the generated *pipelining* informations which are provided in the Appendix C. In the *pipelining* informations, the sections like "`ii = 4 Schedule found with 3 iterations in parallel`" indicate the execution throughput of one iteration of a *for loop*. In this specific example, it indicates that the loop throughput is four clock cycles per iteration, with three iterations executing in a *pipeline* structure. For more information on the generated pipeline information please refer to [39].

Continuing with the specifics of the ported functions, the MbDecoder::xScale4x4Block is a dequantization operation over a 4x4 macroblock and is multiplication intensive. Due to this nature of the function, it includes *for loops* that execute the same operation on different pixel locations of the macroblock. Because of this, the iterations of the *for loop* do not depend on each other, and this makes the loop a good candidate for *pipelining* and the function achieves about 30% gain on the DSP. The LoopFilter::xFilter on the other hand is a nonlinear filtering operation and has many logical instructions and no *for loop*. Therefore, no *pipeline* structure is generated for this function resulting in a slow down on the DSP. This is expected since the clock speed of the DSP is lower than the MPU. The QuarterPelFilter::xPredElse is the function which achieves the most gain, about 60%, from the DSP porting. The reason for this is that this function implements a conventional filtering operation, with a convolution like structure. Such kind of operations are the ones which benefit the most from running on the DSP. The Transform::xInvTransform4x4Blk function also achieves a significant amount of gain, about 25% when ported to the DSP. This function implements the inverse DCT operation which includes only shifts and additions within two *for loop* steps (actually the second *for loop* is separated into two *for loops* for a more efficient *pipeline* structure). Therefore this type of transform operations also make a good candidate for porting to DSP.

However, once the communication overheads are included in the picture, the overhead per call is at least 100 times more than the actual processing time per call, for all of the ported functions. This is unexpected but unavoidable. Thus, it is not possible to achieve any gain through porting any functions of the JMVM software to the DSP.

This is why the structure of the software is so that the most demanding algorithms are run on small data sets, for millions of times. Instead, if the software was structured so that a large amount of data could be transferred to be processed on the DSP, with only one messaging back and forth, then the communication overhead would be negligible when compared to the gain. As a conclusion, for taking advantage of the DSP on the OMAP34x^{TM} MDK, it is required to restructure the JMVM software or write a new MVC codec that would have the appropriate structure. However such a task is beyond the scope of this work.

Chapter 6

CONCLUSIONS

In this thesis, the problem of 3D video coding on a ZOOM^{TM} OMAP34x^{TM} MDK is considered for the 3DPhone project. The thesis work consists of two parts. In the first part, the visual performance of the simulcast, multiview video (MVC) and mixed-resolution stereoscopic (MRSC) video coding methods are compared with subjective tests. In the second part, the JMVM [33] reference multiview video decoder is implemented and tested on the OMAP34x^{TM} MDK.

For the first part of the thesis work, it is intended to find out the performance of the MRSC method, when compared to the simulcast coding and MVC methods. Simulcast coding and MVC methods are the least complex among the 3D video coding methods available, so they are thought to be the most promising coding methods for a mobile platform. On the other hand, MRSC brings some additional computational complexity over these methods, but is still promising since it further reduces the size of the generated bitstream simply by downsampling one of the views of the stereoscopic video data.

Since there are no analytical comparison methods available for 3D visual quality, we conducted subjective tests to compare the simulcast coding, MVC and MRSC without inter-view prediction methods, with basic coding schemes specified specifically for this thesis. Hence, it is crucial to mention that the results obtained from these subjective

tests are valid only for the specific coding schemes and the testbeds we used; however for the sake of simplicity we will use the general names of these methods when providing our findings. For the subjective tests, a few stereoscopic video sequences are encoded on a personal computer with the coding methods to be compared. For the MRSC method, it is possible to select the quantization levels of the two views separately. Hence, we fixed the bit rate ratio of the left-to-right view, and came up with three different MRSC coding methods. These are labeled as MRSC(2:1), MRSC(3:1), MRSC(4:1), where the information given inside the parenthesis refer to the left-to-right bit rate ratios (i.e. 2:1 means the total bit rate is allocated such that the left view consumes the two-thirds of the total bit rate and the right view consumes the remaining).

Then a set of subjective tests are conducted on 15" 3D parallax barrier based LCD screen. However, some of the results from these tests came out to be conflicting. Also, the test subjects informed us about the difficulty to assess the 3D visual quality with the display used. Therefore, we left these tests as inconclusive and conducted a new set of subjective tests on a polarization based display with the help of Fraunhofer HHI. From these tests, results came out to be more consistent. According to these results, MVC is selected to be the coding method providing the best 3D visual quality. For high bit rates, simulcast coding method is voted to be the second best; while for low bit rates MRSC(2:1) is selected to perform as the second best and very close to the MVC method. When our MRSC methods are compared among themselves, MRSC(2:1) is selected to be the most preferred. From these results it is possible to conclude that MRSC(2:1) is promising for mobile platforms, since it is expected to deal with low bit rate videos most of the time. Although MVC performed better than MRSC(2:1) even for the low bit rates, perhaps a lower bit rate ratio MRSC can perform better than MVC. Also since MRSC(2:1) performed better than simulcast coding for low bit rates, allowing inter-view prediction for MRSC might even make MRSC outperform the MVC method. Hence, deploying both MVC and MRSC decoding features on the mobile device can be a promising approach. Additionally, since the results are found to be highly dependent on the used display, it is concluded that MRSC needs further testing, both in terms of 3D perception quality and computational complexity, when the first prototype of the 3DPhone is ready with an embedded 3D display. Until

then, investigations on the video decoding performance of the already available mobile platform are continued with the MVC method.

For the second part of the thesis, it is intended to port to and run the JMVM MVC decoder on the OMAP34x^{TM} MDK. Initially the software is compiled for and tested on the desired platform without any modifications. However the decoding tests resulted in a low number of frames per second, and we decided to take advantage of the available digital signal processor (DSP) of the MDK while decoding 3D videos. To select which functions to implement on the DSP, we first did a profiling of the software and analyzed which functions are the most demanding ones in terms of computational power. As expected the scaling, filtering and transform operations came out to be the most computationally expensive as the JMVM decoder is based on the H.264/MPEG-4 AVC standard. Then, a scaling, a transform, a convolution-like filtering and a nonlinear filtering operation are selected to be ported to run on the DSP. The selected functions are implemented to run on the DSP and tested.

The tests showed us that we achieve a performance gain between 25% and 60% on the DSP, when only the processing times are considered. However, we needed to use some special API functions at the main processing unit to manage the DSP actions, which required an additional communication overhead. According to the test results, there exists a very large communication overhead between the MPU and the DSP. This communication overhead happens each time an algorithm is run on the DSP and is inevitable. When compared with the processing times, this overhead is about hundred times more than the processing times of any of the selected functions. Hence, it is not possible to achieve any gain from the DSP for the JMVM decoder. However, if the software structure is changed to allow longer processing times and less number of separate function calls to the DSP; DSP can be very beneficial for making the execution of the decoder faster. Yet, this is left apart of this thesis work, since the JMVM decoder consists of about 100000 lines of code and such a work is beyond the scope of this thesis work.

Appendix A

JMVM CONFIGURATION FILE

```
# JMVM Main Configuration File

# This is a sample configuration file for simulcast coding the Bullinger sequence with
# a 640x352 resolutionand a main QP of 30. For MVC follow the comments at the end of
# the file!

#====================== GENERAL ======================================================

InputFile               Bullinger_640x352          # input file
OutputFile              Bullinger_640x352_QP30     # bitstream file
ReconFile               Bullinger_640x352_QP30_rec # reconstructed file
MotionFile              Bullinger_640x352_QP30_mot # motion information file
SourceWidth             640                        # input  frame width
SourceHeight            352                        # input  frame height
FrameRate               25.0                       # frame rate [Hz]
FramesToBeEncoded       250                        # number of frames

#====================== CODING =======================================================

SymbolMode              1                          # 0=CAVLC, 1=CABAC
FRExt                   1                          # 8x8 transform (0:off, 1:on)
BasisQP                 30                         # Quantization parameters

#====================== STRUCTURE ====================================================

GOPSize                 2                          # GOP Size (at maximum frame rate)
IntraPeriod             8                          # Anchor Period
NumberReferenceFrames   2                          # Number of reference pictures
InterPredPicsFirst      1                          # 1 Inter Pics; 0 Inter-view Pics
DeltaLayer0Quant        0                          # differential QP for layer 0
DeltaLayer1Quant        1                          # differential QP for layer 1
DeltaLayer2Quant        2                          # differential QP for layer 2
DeltaLayer3Quant        3                          # differential QP for layer 3
DeltaLayer4Quant        4                          # differential QP for layer 4
DeltaLayer5Quant        5                          # differential QP for layer 5
```

Figure A.1: Sample JMVM configuration file

```
#====================== MOTION SEARCH ==============================================

SearchMode              4                       # Search mode (0:BlockSearch,
                                                #    4:FastSearch)
SearchFuncFullPel       3                       # Search function full pel
                                                #   (0:SAD, 1:SSE,
                                                #    2:HADAMARD, 3:SAD-YUV)
SearchFuncSubPel        2                       # Search function sub pel
                                                #   (0:SAD, 1:SSE, 2:HADAMARD)
SearchRange             32                      # Search range (Full Pel)
BiPredIter              4                       # Max iterations for bi-pred search
IterSearchRange         8                       # Search range for iterations
                                                #   (0: normal)

#====================== LOOP FILTER ================================================

LoopFilterDisable       0                       # Loop filter idc (0: on, 1: off,
                                                #    2: on except for
                                                #       slice boundaries)
LoopFilterAlphaC0Offset 0                       # AlphaOffset(-6..+6): valid range
LoopFilterBetaOffset    0                       # BetaOffset (-6..+6): valid range

#====================== WEIGHTED PREDICTION ========================================

WeightedPrediction      0                       # Weighting IP Slice (0:disable,
                                                #    1:enable)
WeightedBiprediction    0                       # Weighting B  Slice (0:disable,
                                                #    1:explicit, 2:implicit)

#====================== MULTIVIEW CODING PARAMETERS ================================

ICMode                  0                       #(0: IC off, 1: IC on)
MotionSkipMode          0                       #(0: Motion skip mode off,
                                                #    1: Motion skip mode on)
SingleLoopDecoding      0                       #(0: SLD mode off, 1: SLD mode on)

#====================== NESTING SEI MESSAGE ========================================

NestingSEI              0                       #(0: NestingSEI off,
                                                #    1: NestingSEI on)
SnapShot                0                       #(0: SnapShot off, 1: SnapShot on)

#====================== ACTIVE VIEW INFO SEI MESSAGE ================================

ActiveViewSEI           0                       #(0: ActiveViewSEI off,
                                                #    1: ActiveViewSEI on)

#====================== VIEW SCALABILITY INFOMATION SEI MESSAGE =====================

ViewScalInfoSEI         0                       #(0: ViewScalSEI off,
                                                #    1: ViewScalSEI on)

#====================== PARALLEL DECODING INFORMATION SEI Message ===================

PDISEIMessage           0                       # PDI SEI message enable
                                                #    (0: disable , 1:enable)
PDIInitialDelayAnc      2                       # PDI initial delay for anchor
                                                #    pictures
PDIInitialDelayNonAnc   2                       # PDI initial delay for non-anchor
                                                #    pictures

#====================== MULTI-VIEW REFERENCING INFORMATION  =========================

NumViewsMinusOne        1                       # (Number of view to be coded
                                                #    minus 1)
ViewOrder               0-1                     # (Order in which view_ids
                                                #    are coded)
```

Figure A.1: Sample JMVM Configuration File (continued)

```
View_ID                 0                       # (view_id of a view 0 - 1030)
Fwd_NumAnchorRefs       0                       # (number of list_0 references
                                                #    for anchor)
Bwd_NumAnchorRefs       0                       # (number of list 1 references
                                                #    for anchor)
Fwd_NumNonAnchorRefs    0                       # (number of list 1 references
                                                #    for non-anchor)
Bwd_NumNonAnchorRefs    0                       # (number of list 1 references
                                                #    for non-anchor)

View_ID                 1
Fwd_NumAnchorRefs       0                       # Change this value to 1 for MVC
Bwd_NumAnchorRefs       0
Fwd_NumNonAnchorRef     0
Bwd_NumNonAnchorRefs    0
#Fwd_AnchorRefs         0 0                     # Uncomment this line for MVC
#Bwd_AnchorRefs         0 0
#Fwd_NonAnchorRefs      0 0
#Bwd_NonAnchorRefs      0 0
```

Figure A.1: Sample JMVM Configuration File (continued)

Appendix B

PROFILING RESULTS OF JMVM

Table B.1: Profiling results for Bullinger

Time (%)	Number of calls	Function name
8.01	10251840	MbDecoder::xScale4x4Block
4.08	386918	IntYuvMbBuffer::add
3.94	165774615	QpParameter::rem
3.43	386918	IntYuvMbBuffer::loadBuffer
3.40	14096288	LoopFilter::xFilter
3.12	386918	YuvMbBuffer::loadBufferClip
1.83	1206187	MbTransformCoeffs::clear
1.63	165928695	QpParameter::per
1.56	6756844	LoopFilter::xCheckMvDataB
1.40	3499126	MotionCompensation::xPredChromaPel
1.37	65822	QuarterPelFilter::xPredElse
1.26	1749563	QuarterPelFilter::predBlk
1.21	752119	SampleWeighting::xMixB8x
1.00	1	H264AVCDecoderTest::go
0.86	37262022	MbMotionData::getRefPic
0.83	24409	QuarterPelFilter::xPredDy2Dx13
0.83	48214684	LumaIdx::x
0.81	6885920	LoopFilter::xGetVerFilterStrength
0.78	35279270	LumaIdx::operator+
0.78	440000	MbDecoder::xScaleTCoeffs
0.75	58688	QuarterPelFilter::xPredDx2
0.74	30271017	MbDataStruct::is4x4BlkCoded
0.73	440000	LoopFilter::xFilterMb
0.70	6885920	LoopFilter::xGetHorFilterStrength
0.70	440000	YuvPicBuffer::loadBuffer
0.67	1107460	QpParameter::setQp
0.67	867029	Transform::xInvTransform4x4Blk

Table B.2: Profiling results for Car

Time (%)	Number of calls	Function name
6.77	9298096	MbDecoder::xScale4x4Block
4.00	22466748	LoopFilter::xFilter
3.81	362753	IntYuvMbBuffer::add
3.80	214736	QuarterPelFilter::xPredElse
3.31	155956530	QpParameter::rem
3.17	113493	QuarterPelFilter::xPredDy2Dx13
2.57	362753	IntYuvMbBuffer::loadBuffer
2.41	2647944	MotionCompensation::xPredChromaPel
2.32	362753	YuvMbBuffer::loadBufferClip
1.76	153958	QuarterPelFilter::xPredDy0Dx13
1.62	182599	QuarterPelFilter::xPredDx2
1.35	119255	QuarterPelFilter::xPredDx0Dy13
1.11	101773	QuarterPelFilter::xPredDx2Dy13
0.99	156270682	QpParameter::per
0.97	1026348	MbTransformCoeffs::clear
0.94	511831	SampleWeighting::xMixB8x
0.93	886485	Transform::xInvTransform4x4Blk

Table B.3: Profiling results for Hands

Time (%)	Number of calls	Function names
5.31	33841224	LoopFilter::xFilter
5.16	9694192	MbDecoder::xScale4x4Block
3.52	210082	QuarterPelFilter::xPredElse
3.33	363098	IntYuvMbBuffer::add
2.69	363098	YuvMbBuffer::loadBufferClip
2.59	31198734	CabaDecoder::getSymbol
2.25	363098	IntYuvMbBuffer::loadBuffer
2.21	166800165	QpParameter::rem
1.79	64373	QuarterPelFilter::xPredDy2Dx13
1.48	31845009	CabaDecoder::xReadBit
1.48	1583319	Transform::xInvTransform4x4Blk
1.29	3017800	MotionCompensation::xPredChromaPel

Table B.4: Profiling results for Pantomime

Time (%)	Number of calls	Function name
7.47	23787472	MbDecoder::xScale4x4Block
4.29	951641	IntYuvMbBuffer::add
3.90	391141740	QpParameter::rem
3.16	951641	YuvMbBuffer::loadBufferClip
3.12	41345788	LoopFilter::xFilter
2.78	359943	QuarterPelFilter::xPredElse
2.69	951641	IntYuvMbBuffer::loadBuffer
1.63	2766203	MbTransformCoeffs::clear
1.62	8260408	MotionCompensation::xPredChromaPel
1.30	15359008	LoopFilter::xCheckMvDataB
1.22	391689604	QpParameter::per
1.20	4130204	QuarterPelFilter::predBlk
1.16	2521021	Transform::xInvTransform4x4Blk

Appendix C

CODES AND PIPELINE INFORMATION

The source codes provided in this chapter are taken from the JMVM reference software and slightly modified to be tested on the DSP core of the ZOOM^{TM} OMAP34x^{TM} MDK. The details on how to achieve the JMVM's source code is provided in [33].

MbDecoder::xScale4x4Block

```
1  void MbDecoder::xScale4x4Block(short* piCoeff, unsigned char
       isPucScale, int uiStart, int rcQPper, int rcQPrem, const
       unsigned char* pucScale)
2  {
3    unsigned int ui;
4
5    if( isPucScale == 'y')
6    {
7      int iAdd = ( rcQPper <= 3 ? ( 1 << ( 3 - rcQPper ) ) : 0 );
8
9      for(ui = uiStart; ui < 16; ui++ )
10     {
11       piCoeff[ui] = ( ( piCoeff[ui] *
             gaaiDequantCoef[rcQPrem][ui] * pucScale[ui] + iAdd ) <<
              rcQPper ) >> 4;
12     }
13   }
14   else if ( isPucScale == 'n')
15   {
```

```
16      for(ui = uiStart; ui < 16; ui++ )
17      {
18        piCoeff[ui] *= ( gaaiDequantCoef[rcQPrem][ui] << rcQPper );
19      }
20    }
21  }
```

```
-------------------------------------------------------------------------
SOFTWARE PIPELINE INFORMATION

   Loop source line                 : 9
   Loop opening brace source line   : 10
   Loop closing brace source line   : 12
   Known Minimum Trip Count         : 1
   Known Max Trip Count Factor      : 1
   Loop Carried Dependency Bound(^) : 0
   Unpartitioned Resource Bound     : 2
   Partitioned Resource Bound(*)    : 2
   Resource Partition:
                             A-side   B-side
   .L units                     0        0
   .S units                     1        0
   .D units                     1        2*
   .M units                     0        1
   .X cross paths               0        1
   .T address paths             1        2*
   Long read paths              0        0
   Long write paths             0        0
   Logical  ops (.LS)           0        0     (.L or .S unit)
   Addition ops (.LSD)          0        0     (.L or .S or .D unit)
   Bound(.L .S .LS)             1        0
   Bound(.L .S .D .LS .LSD)     1        1

   Searching for software pipeline schedule at ...
      ii = 2  Schedule found with 5 iterations in parallel

   Register Usage Table:
       +-------------------------------------------------------------------+
       |AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA|BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB|
       |00000000001111111111222222222233|00000000001111111111222222222233|
       |01234567890123456789012345678901|01234567890123456789012345678901|
       |--------------------------------+--------------------------------|
    0: |   * **                         |      **                        |
    1: |   ****                         |    ****                        |
       +-------------------------------------------------------------------+

   Done
-------------------------------------------------------------------------
```

Figure C.1: Pipeline information for MbDecoder::xScale4x4Block (first loop)

```
--------------------------------------------------------------------------
SOFTWARE PIPELINE INFORMATION

   Loop source line                 : 16
   Loop opening brace source line   : 17
   Loop closing brace source line   : 19
   Known Minimum Trip Count         : 1
   Known Max Trip Count Factor      : 1
   Loop Carried Dependency Bound(^) : 2
   Unpartitioned Resource Bound     : 2
   Partitioned Resource Bound(*)    : 2
   Resource Partition:
                              A-side   B-side
   .L units                      0        0
   .S units                      1        1
   .D units                      2*       2*
   .M units                      0        2*
   .X cross paths                1        1
   .T address paths              2*       2*
   Long read paths               0        0
   Long write paths              0        0
   Logical  ops (.LS)            0        0     (.L or .S unit)
   Addition ops (.LSD)           0        1     (.L or .S or .D unit)
   Bound(.L .S .LS)              1        1
   Bound(.L .S .D .LS .LSD)      1        2*

   Searching for software pipeline schedule at ...
      ii = 2  Schedule found with 10 iterations in parallel

   Register Usage Table:
       +-----------------------------------------------------------------+
       |AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA|BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB|
       |00000000001111111111222222222233|00000000001111111111222222222233|
       |01234567890123456789012345678901|01234567890123456789012345678901|
       |--------------------------------+--------------------------------|
    0: |   ***                          |    ******      * *             |
    1: |   ****                         |    ******      **              |
       +-----------------------------------------------------------------+

   Done
--------------------------------------------------------------------------
```

Figure C.2: Pipeline information for MbDecoder::xScale4x4Block (second loop)

LoopFilter::xFilter

```
1  void LoopFilter::xFilter(unsigned char pFlt[], unsigned char
       bLumCode, int iAlpha, int iBeta, int iDelta, int C0, unsigned
        char ucBs)
2  {
3    int P0 = pFlt[-1 + 4];
4    int Q0 = pFlt[ 0 + 4];
5    int P1 = pFlt[-2 + 4];
6    int Q1 = pFlt[ 1 + 4];
7    int iAbsDelta = abs(iDelta);
8    int P2,Q2,ap,aq,iDiff,bEnable,PQ0;
9
10   if( ucBs < 4 )
11   {
12
13     if( bLumCode == 'y' )
14     {
15       P2 = pFlt[-3 + 4] ;
16       Q2 = pFlt[ 2 + 4] ;
17       aq = (( abs( Q2 - Q0 ) < iBeta ) ? 1 : 0 );
18       ap = (( abs( P2 - P0 ) < iBeta ) ? 1 : 0 );
19
20       if( ap )
21       {
22         pFlt[-2 + 4] = P1 + gClipMinMax((P2 + ((P0 + Q0 + 1)>>1)
               - (P1<<1)) >> 1, -C0, C0 );
23       }
24
25       if( aq  )
26       {
27         pFlt[   1 + 4] = Q1 + gClipMinMax((Q2 + ((P0 + Q0 +
               1)>>1) - (Q1<<1)) >> 1, -C0, C0 );
28       }
29
30       C0 += ap + aq -1;
31     }
32
33     C0++;
34     iDiff      = gClipMinMax(((iDelta << 2) + (P1 - Q1) + 4) >>
          3, -C0, C0 ) ;
35     pFlt[-1 + 4] = gClip( P0 + iDiff );
36     pFlt[ 0 + 4] = gClip( Q0 - iDiff );
37     return;
38   }
39
40   if( bLumCode == 'n' )
41   {
42     pFlt[ 0 + 4] = ((Q1 << 1) + Q0 + P1 + 2) >> 2;
43     pFlt[-1 + 4] = ((P1 << 1) + P0 + Q1 + 2) >> 2;
44   }
45   else
46   {
47     P2 = pFlt[-3 + 4] ;
48     Q2 = pFlt[ 2 + 4] ;
```

```
49      bEnable  = (iAbsDelta < ((iAlpha >> 2) + 2));
50      aq       = bEnable & ( abs( Q2 - Q0 ) < iBeta );
51      ap       = bEnable & ( abs( P2 - P0 ) < iBeta );
52      PQ0 = P0 + Q0;
53
54      if( aq )
55      {
56        pFlt[0 + 4] = (P1 + ((Q1 + PQ0) << 1) +  Q2 + 4) >> 3;
57        pFlt[1 + 4] = (PQ0 +Q1 + Q2 + 2) >> 2;
58        pFlt[2 + 4] = (((pFlt[ 3 + 4] + Q2) <<1) + Q2 + Q1 + PQ0 +
              4) >> 3;
59      }
60      else
61      {
62        pFlt[0 + 4] = ((Q1 << 1) + Q0 + P1 + 2) >> 2;
63      }
64
65      if( ap )
66      {
67        pFlt[-1 + 4] = (Q1 + ((P1 + PQ0) << 1) +  P2 + 4) >> 3;
68        pFlt[-2 + 4] = (PQ0 +P1 + P2 + 2) >> 2;
69        pFlt[-3 + 4] = (((pFlt[-4 + 4] + P2) <<1) + pFlt[-3 + 4] +
              P1 + PQ0 + 4) >> 3;
70      }
71      else
72      {
73        pFlt[-1 + 4] = ((P1 << 1) + P0 + Q1 + 2) >> 2;
74      }
75    }
76  }
```

For LoopFilter::xFilter, the compiler cannot generate a pipeline since there is no *for loop* inside the function.

QuarterPelFilter::xPredElse

```
1  void QuarterPelFilter::xPredElse(int iSrcStride, int iDestStride,
       int iDx, int iDy, int uiSizeX, int uiSizeY, unsigned char*
     restrict pucSrc, unsigned char* restrict pucDest)
2  {
3    unsigned char* pucSrcX = pucSrc;
4    unsigned char* pucSrcY = pucSrc;
5    unsigned int x, y;
6    int iTempX, iTempY;
7
8    pucSrcY += (iDx == 1) ? 0 : 1;
9    pucSrcX += (iDy == 1) ? 0 : iSrcStride;
10
11   for( y = 0; y < uiSizeY; y++)
12   {
13     #pragma MUST_ITERATE (4,16,4)
14     for( x = 0; x < uiSizeX; x++)
15     {
16       iTempX  = pucSrcX[x - 0];
17       iTempX += pucSrcX[x + 1];
18       iTempX  = iTempX << 2;
19       iTempX -= pucSrcX[x - 1];
20       iTempX -= pucSrcX[x + 2];
21       iTempX += iTempX << 2;
22       iTempX += pucSrcX[x - 2];
23       iTempX += pucSrcX[x + 3];
24       iTempX = gClip( (iTempX + 16) >> 5 );
25
26       iTempY  = pucSrcY[x - 0*iSrcStride];
27       iTempY += pucSrcY[x + 1*iSrcStride];
28       iTempY  = iTempY << 2;
29       iTempY -= pucSrcY[x - 1*iSrcStride];
30       iTempY -= pucSrcY[x + 2*iSrcStride];
31       iTempY += iTempY << 2;
32       iTempY += pucSrcY[x - 2*iSrcStride];
33       iTempY += pucSrcY[x + 3*iSrcStride];
34       iTempY = gClip( (iTempY + 16) >> 5 );
35
36       pucDest[x] = (iTempX + iTempY + 1) >> 1;
37     }
38     pucDest += iDestStride;
39     pucSrcX += iSrcStride;
40     pucSrcY += iSrcStride;
41   }
42 }
```

```
----------------------------------------------------------------------------
SOFTWARE PIPELINE INFORMATION

   Loop source line                 : 14
   Loop opening brace source line   : 15
   Loop closing brace source line   : 37
   Known Minimum Trip Count         : 4
   Known Maximum Trip Count         : 16
   Known Max Trip Count Factor      : 4
   Loop Carried Dependency Bound(^) : 3
   Unpartitioned Resource Bound     : 10
   Partitioned Resource Bound(*)    : 10
   Resource Partition:
                             A-side   B-side
   .L units                     2        4
   .S units                     7        6
   .D units                     6        6
   .M units                     2        1
   .X cross paths               3        7
   .T address paths             6        7
   Long read paths              0        0
   Long write paths             0        0
   Logical  ops (.LS)           0        0     (.L or .S unit)
   Addition ops (.LSD)         14       14     (.L or .S or .D unit)
   Bound(.L .S .LS)             5        5
   Bound(.L .S .D .LS .LSD)    10*      10*

   Searching for software pipeline schedule at ...
      ii = 10 Register is live too long
      ii = 10 Schedule found with 4 iterations in parallel

   Register Usage Table:
       +----------------------------------------------------------------+
       |AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA|BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB|
       |00000000001111111111222222222233|00000000001111111111222222222233|
       |01234567890123456789012345678901|01234567890123456789012345678901|
       |--------------------------------+--------------------------------|
    0: | * ** **        ************    |*   ***  *      ** ********     |
    1: |** *****        ************    |    ******      ***********     |
    2: |*  *****        ************    |    ******      ***********     |
    3: |*  *******      ************    |    ******      *  ********     |
    4: |*  *******      ************    |    ******         ********     |
    5: |*  *******      ************    |*   * ****         ********     |
    6: |** *******      ************    |*     * **         ********     |
    7: |** *****        ************    |*    ** **         ********     |
    8: |** **           ************    |    ******         ********     |
    9: | *    *         ************    |*   ****          * ********     |
       +----------------------------------------------------------------+

   Done
----------------------------------------------------------------------------
```

Figure C.3: Pipeline information for QuarterPelFilter::xPredElse (inner loop)

Transform::xInvTransform4x4Blk

```
1  void Transform::xInvTransform4x4Blk(short * restrict puc, short *
       restrict piCoeff, int iStride)
2  {
3    int aai[4][4];
4    int tmp1, tmp2, tmp3, tmp4, x, y, iStride2, iStride3;
5    short tmpS1, tmpS2;
6    short *pucInd1, *pucInd2;
7
8    pucInd1 = puc;
9    pucInd2 = puc;
10
11   _nassert(iStride == 24);
12
13   iStride2 = 2*iStride;
14   iStride3 = 3*iStride;
15
16   for( x = 0; x < 4; x++, piCoeff+=4 )
17   {
18     tmp1 = piCoeff[0] + piCoeff[2];
19     tmp2 = (piCoeff[3]>>1) + piCoeff[1];
20
21     aai[0][x] = tmp1 + tmp2;
22     aai[3][x] = tmp1 - tmp2;
23
24     tmp3 = piCoeff[0] - piCoeff[2];
25     tmp4 = (piCoeff[1]>>1) - piCoeff[3];
26
27     aai[1][x] = tmp3 + tmp4;
28     aai[2][x] = tmp3 - tmp4;
29   }
30
31   for( y = 0; y < 4; y++, pucInd1 ++ )
32   {
33     tmp1 =  aai[y][0] + aai[y][2];
34     tmp2 = (aai[y][3]>>1) + aai[y][1];
35
36     tmpS1 = xClip( xRound( tmp1 + tmp2) + pucInd1[0] );
37     tmpS2 = xClip( xRound( tmp1 - tmp2) + pucInd1[iStride3] );
38
39     pucInd1[0]         = tmpS1;
40     pucInd1[iStride3] = tmpS2;
41   }
42
43
44   for( y = 0; y < 4; y++, pucInd2 ++ )
45   {
46     tmp1 =  aai[y][0] - aai[y][2];
47     tmp2 = (aai[y][1]>>1) - aai[y][3];
48
49     tmpS1 = xClip( xRound( tmp1 + tmp2) + pucInd2[iStride] );
50     tmpS2 = xClip( xRound( tmp1 - tmp2) + pucInd2[iStride2] );
51
52     pucInd2[iStride]  = tmpS1;
```

```
53      pucInd2[iStride2] = tmpS2;
54    }
55  }
```

```
----------------------------------------------------------------------------
SOFTWARE PIPELINE INFORMATION

   Loop source line                 : 16
   Loop opening brace source line   : 17
   Loop closing brace source line   : 29
   Known Minimum Trip Count         : 4
   Known Maximum Trip Count         : 4
   Known Max Trip Count Factor      : 4
   Loop Carried Dependency Bound(^) : 3
   Unpartitioned Resource Bound     : 4
   Partitioned Resource Bound(*)    : 4
   Resource Partition:
                             A-side   B-side
   .L units                     0        0
   .S units                     2        0
   .D units                     4*       4*
   .M units                     0        0
   .X cross paths               2        2
   .T address paths             4*       4*
   Long read paths              0        0
   Long write paths             0        0
   Logical  ops (.LS)           2        0     (.L or .S unit)
   Addition ops (.LSD)          3        4     (.L or .S or .D unit)
   Bound(.L .S .LS)             2        0
   Bound(.L .S .D .LS .LSD)     4*       3

   Searching for software pipeline schedule at ...
      ii = 4  Schedule found with 3 iterations in parallel

   Register Usage Table:
       +--------------------------------------------------------------------+
       |AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA|BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB|
       |00000000001111111111222222222233|00000000001111111111222222222233|
       |01234567890123456789012345678901|01234567890123456789012345678901|
       |--------------------------------+--------------------------------|
    0: |   *******      **              |    *** **      *               |
    1: |   *******      **              |    *** **      **              |
    2: |   *******      **              |    ******      **              |
    3: |   *******      *               |    ******      **              |
       +--------------------------------------------------------------------+

   Done
----------------------------------------------------------------------------
```

Figure C.4: Pipeline information for Transform::xInvTransform4x4Blk (first loop)

```
--------------------------------------------------------------------------
SOFTWARE PIPELINE INFORMATION

   Loop source line                 : 31
   Loop opening brace source line   : 32
   Loop closing brace source line   : 41
   Known Minimum Trip Count         : 4
   Known Maximum Trip Count         : 4
   Known Max Trip Count Factor      : 4
   Loop Carried Dependency Bound(^) : 3
   Unpartitioned Resource Bound     : 6
   Partitioned Resource Bound(*)    : 6
   Resource Partition:
                             A-side   B-side
   .L units                     3        3
   .S units                     5        4
   .D units                     4        2
   .M units                     0        0
   .X cross paths               4        3
   .T address paths             3        3
   Long read paths              0        0
   Long write paths             0        0
   Logical  ops (.LS)           0        0     (.L or .S unit)
   Addition ops (.LSD)          5        9     (.L or .S or .D unit)
   Bound(.L .S .LS)             4        4
   Bound(.L .S .D .LS .LSD)     6*       6*

   Searching for software pipeline schedule at ...
      ii = 6  Schedule found with 7 iterations in parallel

   Register Usage Table:
       +--------------------------------------------------------------------+
       |AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA|BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB|
       |00000000001111111111222222222233|00000000001111111111222222222233|
       |01234567890123456789012345678901|01234567890123456789012345678901|
       |--------------------------------+--------------------------------|
    0: |***** ****      ****            | *  ******      ***   ***       |
    1: |**********      *****           |**  ******      *********       |
    2: | *********      ***             |**  ******      *********       |
    3: |***** ****      ***             |*** ******      *********       |
    4: |* ********      ***             |*** * ****      *** *** *       |
    5: | *********      ***             |*** ******      *** *  **       |
       +--------------------------------------------------------------------+

   Done
--------------------------------------------------------------------------
```

Figure C.5: Pipeline information for Transform::xInvTransform4x4Blk (second loop)

```
--------------------------------------------------------------------------
SOFTWARE PIPELINE INFORMATION

   Loop source line                 : 44
   Loop opening brace source line   : 45
   Loop closing brace source line   : 54
   Known Minimum Trip Count         : 4
   Known Maximum Trip Count         : 4
   Known Max Trip Count Factor      : 4
   Loop Carried Dependency Bound(^) : 3
   Unpartitioned Resource Bound     : 6
   Partitioned Resource Bound(*)    : 6
   Resource Partition:
                             A-side   B-side
   .L units                     3        3
   .S units                     5        4
   .D units                     4        2
   .M units                     0        0
   .X cross paths               4        3
   .T address paths             3        3
   Long read paths              0        0
   Long write paths             0        0
   Logical  ops (.LS)           0        2     (.L or .S unit)
   Addition ops (.LSD)          5        7     (.L or .S or .D unit)
   Bound(.L .S .LS)             4        5
   Bound(.L .S .D .LS .LSD)     6*       6*

   Searching for software pipeline schedule at ...
      ii = 6  Schedule found with 7 iterations in parallel

   Register Usage Table:
       +--------------------------------------------------------------------+
       |AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA|BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB|
       |00000000001111111111222222222233|00000000001111111111222222222233|
       |01234567890123456789012345678901|01234567890123456789012345678901|
       |--------------------------------+--------------------------------|
    0: |***** ****      ****            | *  ******      * **  ***       |
    1: |**********      *****           |**  ******      *********       |
    2: | *********      ***             |**  ******      *********       |
    3: |***** ****      ***             |*** ******      *********       |
    4: |* ********      ***             |***  *****      * ***** *       |
    5: | *********      ***             |*** ******      * ***  **       |
       +--------------------------------------------------------------------+

   Done
--------------------------------------------------------------------------
```

Figure C.6: Pipeline information for Transform::xInvTransform4x4Blk (third loop)

Appendix D

USER'S MANUAL

D.1 Repeating the subjective tests

For this section the Sharp Actius AL3DU laptop is used. However it can be repeated also on on a regular PC, with a Windows XP (or later) operating system. Of course in that case, a stereoscopic 3D monitor is also required.

D.1.1 Preliminary parts

Install AutoHotKey: `www.autohotkey.com`

Install Stereoscopic Player: `www.3dtv.at`

Install IrfanView and associate `.png` files with it: `www.irfanview.com`

Copy the following files and folders located in the `Subjective Tests` folder inside the DVD:

- inscreens
- Subjective Test Videos
- All `.ahk` files

into `D:\` partition, keeping the file hierarchy. Otherwise the AutoHotKey scripts need to be edited, and the required editing will be explained later.

D.1.2 Running the tests

On Sharp Actius AL3DU laptop

Before starting the tests, make sure you open Stereoscopic Player and leave it in the desired state (change the "Viewing Method" to "Sharp 3D Display" and the "Preferred Viewing Method" to "Stereoscopic").

Close all instances of Stereoscopic Player before running the script.

Open IrfanView and leave it in maximized state for better visual appearance of the tests. Then make sure you close all instances of IrfanView too.

Double-click on the script "Subjective Test.ahk" to start tests.

On generic PC

For using the script on a generic PC, it needs to be modified according to the setup. To do those, we need to record some information related with the setup:

Run Stereoscopic Player without changing its original position and press Ctrl+F.

In installation folder of AutoHotKey, find AU3_Spy.exe and run it.
(i.e. `C:\Program Files\AutoHotkey\AU3_Spy.exe`)

This utility will help us to find the pixel positions of the "Left file:" and "Right file:" boxes of Stereoscopic Player.

Position your pointer (and click) on "Left file:" box and observe the "In Active Window" pixel locations from "Active Window Info". Record these pixel locations for later use.

The steps for running the subjective tests on a generic PC will be continued after listing the required changes on the script. Some of these changes may be necessary for the Sharp laptop case as well. (if the file hierarchy is modified, or `D:\` partition is not available).

Changes required on the script

If the video files are not put directly onto the `D:\` drive, you have to change the "currentDir" values in the script appropriately. Search through the script for "currentDir" and make sure that you include also the sequence names (with an underscore at the end).
(i.e. `currentDir = D:\Subjective Test Videos\Bullinger\HIGH\Bullinger_`)

If the "inscreens" folder is not put directly onto the `D:\` drive, you have to change the "inscreenDir" values in the script appropriately. Search through the script for "inscreensDir" and make sure that you include also "inscreen_" at the end.
(i.e. `inscreenDir = D:\inscreens\inscreen_`)

The script assumes that you have installed Stereoscopic Player in the default installation location.
(i.e. `C:\Program Files\Stereoscopic Player\`).
If your installation folder differs, search for "StereoPlayer.exe" in the script and modify these lines appropriately.

If a registered version of Stereoscopic Player is used, search for " (unregistered)" in the script and remove them.

Search for “Click” in the script and change the value of its 1st occurrence (meaning that there will always be two consecutive occurrences, but they appear more than once through the script!) with pixels recorded for “Left file:” box and 2nd with the pixels recorded for the “Right file:”.

“videoLength” variable appear throughout the script and it controls the amount of time to pause while watching the upcoming videos until its value is changed. Its unit is milliseconds. For optimized play time on your PC, you have to search for “videoLength” and play with its value for optimized play time.

Through the script there are some “sleep 3000” commands in order to make sure that videos are loaded in Stereoscopic Player, before hitting for “fullscreen”. In the scripts, unscaled fullscreen mode is used. If it is required to change that replace “Send ^!Enter”s with “Send !Enter”.

On generic PC (continued)

Now it is possible to run the subjective tests with steps similar to the ones given for the Sharp laptop case.

Before starting the tests, make sure you open Stereoscopic Player and leave it in the desired state (change the “Viewing Method” according the monitor to be used and the “Preferred Viewing Method” to “Stereoscopic”).

Close all instances of Stereoscopic Player before running the script.

Open IrfanView and leave it in maximized state for better visual appearance of the tests. Then make sure you close all instances of IrfanView too.

Double-click on the script “Subjective Test.ahk” to start tests.

D.1.3 Additional comments - in case an error occurs

If some error occurs, first close all instances of Stereoscopic Player and IrfanView. Then close the running script.

To start again at a certain stage (i.e. Car, 7th test), comment out (;) all "CompareVideos" calls before that, save and restart the script by doubleclicking.

D.2 Repeating the implementation steps

For this section a Linux operating system is required on the host PC. For our implementation, Ubuntu 8.04 is used but any other Linux distribution should be usable. As a note, the data that is used in this chapter is provided in the supplemented DVDs and it is assumed that the DVD is mounted as `/media/dvd/` in the OS.

D.2.1 Setting up hardware platform

Setting up the terminal

For communicating with the OMAP34x™ MDK via the host PC, we need to set up a terminal emulator. This section explains how to set up this terminal emulator.

Install `minicom` as the terminal emulator. `minicom` is available from the repositories. (i.e. `sudo apt-get install minicom`)

After installation run minicom.

```
$ minicom
```

For configuring minicom, press "Ctrl+A O".

Go to "Serial port setup" and press A. Here insert the serial port name that you intend to use (i.e. `/dev/ttyS0`). Then press Enter.

Press E and set the baud rate to 115200 8N1 using A and B keys. Then press Enter.

Press F in the serial port setup, and set Yes for "HW Flow Control". Then press G to set No for "SW Flow Control". Then press Enter.

Go to "Modem and dialog" configuration and clear all the parameters between A and L. Then press Enter.

`minicom` is ready to connect to the OMAP34x^{TM} MDK through the serial port.

Setting up OMAP34x^{TM} MDK

We use a MicroSD card to boot the OMAP34x^{TM} MDK from. Therefore the kernel, boot loader and the file system needs to be put onto the MicroSD card.

To do that we need to format the MicroSD card into two partitions. (A 1GB card is enough, but the MDK supports up to 2GB; and we used a 2GB MicroSD card for our implementation.) The first partition needs to be a 64MB partition to hold the boot loader and kernel; and the rest of the card hosts the file system. For partitioning the MicroSD card appropriately, please follow the "Formatting the SD Card" section in the walkthrough given in OMAP34x^{TM} MDK's support website (`https://omapzoom.org/gf/project/omapzoom/wiki/?pagename=BootingAndFlashing`).

For simple referencing let us assume that the partitions of the MicroSD card are mounted as follows on the host PC:

1st partition: `/media/boot/`

2nd partition: `/media/disk/`

In the first partition copy the files inside `/media/dvd/omap/boot`, which are:

- kernel-23.11.02.uImg
- MLO
- u-boot.bin

```
$ sudo cp /media/dvd/omap/boot/* /media/boot/
```

Into the second partition extract the archive file `fs-23.11.02.tar.bz2`, which is located inside `/media/dvd/filesystem/`, as follows:

```
$ cd /media/disk/
$ sudo tar xf /media/dvd/filesystem/fs-23.11.02.tar.bz2
```

Note that operations regarding the MicroSD card need to be conducted as `root`, since the permission of some files needs `root` permissions to be extracted.

Once these steps are executed, MicroSD card is ready to be used for booting the OMAP34x™ MDK. For getting OMAP34x™ MDK ready for booting, plug the MicroSD card into the card slot of the MDK and connect the MDK to the host PC via serial port. After these, once OMAP34x™ MDK's AC adapter is plugged into an AC socket, MDK boots automatically from the MicroSD card. Once booted, it can be controlled with common Linux shell commands from the `minicom`.

D.2.2 How to compile DSP programs

Before compiling the DSP programs we need to install the developer tools onto the host PC. These tools are, "Code Generation Tools", "DSP/BIOS" and the "arm compilers". These are provided inside `/media/dvd/developer_tools/`.

To install “Code Generation Tools” simply execute its installer as follows:

```
$ sudo /media/dvd/TI-C6000-CGT-v6.0.22.bin
```

Follow the installation instructions and install it under the directory `/usr/local/cgt6x-6.0.22/`.

To install “DSP/BIOS” simply execute its installer as follows:

```
$ sudo /media/dvd/bios_setuplinux_5_33_04.bin
```

Follow the installation instructions and install it under the directory `/usr/local/bios_5_33_04/`.

To install “arm compilers”, first create a folder named `omap` under your home folder and then create `arm-2006q1-3` under the `omap` folder:

```
$ mkdir ~/omap && cd ~/omap && mkdir arm-2006q1-3
```

Then inside the `arm-2006q1-3` folder, extract the archive file named `arm-2006q1-3.tar.bz2` inside as follows:

```
$ cd arm-2006q1-3
$ tar xf /media/dvd/developer_tools/arm-2006q1-3.tar.bz2
```

Once these steps are complete, all the tools that we need to compile the DSP programs are available on the host PC. In order to make these tools globally accessible, we have to add folders containing these tools to the path. To to that, append the lines in the upcoming grey box to the `/.bashrc`:

```
export BIOS_INSTALL_DIR=/usr/local/bios_5_33_04
export C6000_C_DIR=/usr/local/cgt6x-6.0.22/include
export C6000_C_DIR=$C6000_C_DIR:/usr/local/cgt6x-6.0.22/lib
export PATH=$PATH:$HOME/omap/arm-2006q1-3/bin
export PREFIX=$HOME/omap/dspbridge
export TGTROOT=$HOME/omap/dspbridge
export KRNLSRC=$HOME/omap/kernel-23.11.02
export DEPOT=/usr/local
export PATH=$PATH:$DEPOT/BIOS-5.33.04/bios_5_33_04/xdctools
export PATH=$PATH:$DEPOT/BIOS-5.33.04/bios_5_33_04/xdctools/bin
```

To make this work, it is also required to log out of your account and log back in. After having these tools available for global use, we can continue with the compilation procedure of the DSP programs.

The source files of the DSP programs are archived into a `tar` file. To access the source files, inside the ∼`/omap/` folder extract the `dspbridge_07272009.tar` file, which is located inside `/media/dvd/omap/`:

```
$ cd ∼/omap && tar xf /media/dvd/omap/dspbridge_07272009.tar
```

Once the file is extracted, go into the `dspbridge` folder, which has just been created by the extraction operation.

```
$ cd dspbridge
```

Inside this directory, there is a shell script called `copyToMDK.sh`. This script automates the compilation of both the DSP-side and MPU-side of the applications and copies the appropriate files to the MicroSD card. One note is that this script assumes that the partitions of the MicroSD card are mounted in the way explained before (`/media/boot/` and `/media/disk/`). In order to specify any other mount point, it is required to go into the script and change the mount points with the desired ones. To run the script, simply execute the following command:

```
$ sudo ./copyToMDK.sh
```

To manually compile the programs, DSP-side and MPU-side of the programs get compiled in two separate steps.

DSP-side

Inside ∼`/omap/dspbridge` there is a makefile for the DSP-side of the programs.

To start the compilation process, execute the following command:

```
$ gmake -f samplemakefile .samples
```

Once the DSP-side compilation is complete, next step is to compile the MPU-side of the programs.

MPU-side

To compile the MPU-side of the programs, first change the directory to ∼/omap/dspbridge/samples/mpu/src:

```
$ cd /omap/dspbridge/samples/mpu/src
```

Then simply run the makefile as follows:

```
$ make all && make install
```

The previous `make install` command make the MPU-side of the programs to be copied into ∼/omap/dspbridge/target/. Once the compilation finishes, we are ready to proceed with copying the generated binary files to the MicroSD card.

D.2.3 Copying the binaries to OMAP34x™ MDK

In order to copy the binary files to the MicroSD card, execute the following commands:

```
$ sudo cp ∼/omap/dspbridge/target/dspbridge/* /media/disk/dspbridge/
$ sudo cp ∼/omap/samples/dsp/*.dll64P /media/disk/dspbridge/
$ sudo cp ∼/omap/samples/dsp/*.dof64P /media/disk/dspbridge/
```

D.2.4 Running the DSP programs on OMAP34x™ MDK

Before running the DSP programs we also need the data files to be copied to the MicroSD card. These data files are located inside

`/media/dvd/omap/dsp_test_data/`. Simply copy all of these data files into the `/media/disk/dspbridge/` folder:

```
$ sudo cp /media/dvd/omap/dsp_test_data/*.dat /media/disk/dspbridge/
```

Once this step is completed boot up the OMAP34x™ MDK and open `minicom`. Once MDK asks for `login:`, write `root` and press Enter. Then it is possible to run the programs as if they are being executed on any Linux shell.

The programs are located inside `/dspbridge/` folder. Therefore, to run the programs, first change the directory as follows:

```
$ cd /dspbridge/
```

For all the programs written to test the DSP performance of the JMVM functions, same usage syntax is used and is as follows:

```
Usage testFuncName.out [number of execution cycles per data] [number of
    data to be executed] [sequence identifier]

sequence identifier = {0,1,2,3}
0:Bullinger
1:Car
2:Hands
3:Pantomime
```

For the programs written to test the communication overhead, same usage syntax is used and is as follows:

```
Usage testFuncName.out [number fo execution cycles per call] [number of
    calls] [to take average number of times to run the test]
```

For example to test MbDecoder::xScale4x4Block function, and run the algorithm 10000 times on the DSP for each data points, and run it for 999 data points coming from the Hands sequence, one has to execute the DSP program as follows:

```
$ ./testXSca4x4.out 10000 999 2
```

Then one has to calculate and subtract the communication overhead from the results. This is done as follows:

```
$ ./overh0x1000.out 10000 999 1
```

Note that the `overh0x1000` needs to be executed with the exact input values as the `testXSca4x4.out` with the exception that the last input is just a multiplier to the first input this time. This generates the data provided in Table 5.3 for the Hands sequence. A screenshot for this test is provided in Figure D.1.

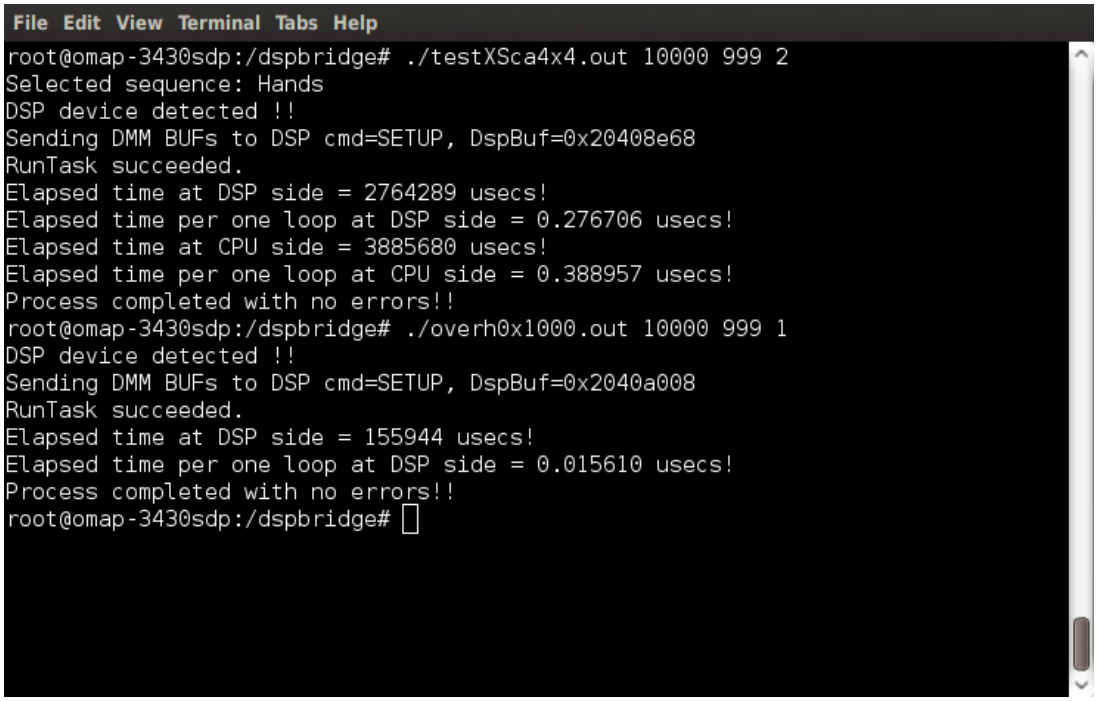


Figure D.1: Screenshot for regenerating the data provided in Table 5.3 for the Hands sequence

One remark is that, the number of data points that the DSP program will run on cannot exceed 999, since the data files we generated includes that many data points. A higher value is allowed by the program syntax; however would lead unexpected behavior for these data files.

In order to match the names of the programs to the functions of JMVM, Table D.1 provides the required information.

Table D.1: Correspondences of the DSP Programs to JMVM functions

Program Executable Name	Corresponding Function
testXSca4x4.out	MbDecoder::xScale4x4Block
testXFilter.out	LoopFilter::xFilter
testXPredElse.out	QuarterPelFilter::xPredElse
testXInv4x4.out	Transform::xInvTransform4x4Blk
over0x10000.out	Calculates the communication overhead for QuarterPelFilter::xPredElse
over0x1000.out	Calculates the communication overhead for MbDecoder::xScale4x4Block, LoopFilter::xFilter, Transform::xInvTransform4x4Blk

D.2.5 Compiling JMVM for OMAP34x™ MDK

The jmvm software with version 8.1 is provided in the DVD, in the folder `/media/dvd/jmvm/`. To compile and run jmvm for OMAP34x™ MDK, follow the instructions given in this section:

Copy the `jmvm` folder to ∼`/omap/dspbridge/`.

```
$ cp -R /media/dvd/jmvm/ ∼/omap/dspbridge/
```

Then go into the `jmvm` directory and edit the `makefile.base` file, located in the `JSVM/H264Extension/build/linux/common` directory, with your favorite text editor. Replace the following lines:

```
CPP   = g++
AR    = ar
```

with

```
CROSS = arm-none-gnueabi-
CPP   = g++
AR    = ar
```

Then simply by following the build instructions given in the Software Manual of JMVM [33], it is possible to compile JMVM to run on OMAP34x™ MDK.

The generated binary files of JMVM software get located inside the `bin` folder, inside the `jmvm` main folder. Copy this folder onto anywhere in the second partition of the MicroSD card.

After these steps running the JMVM software on OMAP34x^{TM} MDK is very straightforward, and is just like running it on a Linux based PC.

Bibliography

[1] A. Hatabu, T. Miyazaki, and I. Kuroda, "QVGA/CIF resolution MPEG-4 video codec based on a low-power and general-purpose DSP," *The Journal of VLSI Signal Processing*, vol. 39, pp. 7–14, January 2005.

[2] M. Zhou and R. Talluri, "Dsp-based real-time video decoding," in *Consumer Electronics, 1999. ICCE. International Conference on*, pp. 296–297, 1999.

[3] W. Lin, K. Goh, B. Tye, G. Powell, T. Ohya, and S. Adachi, "Real time H.263 video codec using parallel DSP," in *Image Processing, 1997. Proceedings., International Conference on*, vol. 2, pp. 586–589 vol.2, October 1997.

[4] K. Ramkishor and V. Gunashree, "Real time implementation of MPEG-4 video decoder on ARM7TDMI," in *Intelligent Multimedia, Video and Speech Processing, 2001. Proceedings of 2001 International Symposium on*, pp. 522–526, 2001.

[5] K. Denolf, A. Chirila-Rus, and D. Verkest, "Low-power MPEG-4 video encoder design," in *Signal Processing Systems Design and Implementation, 2005. IEEE Workshop on*, pp. 284–289, November 2005.

[6] H. Nakayama, T. Yoshitake, H. Komazaki, Y. Watanabe, H. Araki, K. Morioka, J. Li, L. Peilin, S. Lee, H. Kubosawa, and Y. Otobe, "An MPEG-4 video LSI with an error-resilient codec core based on a fast motion estimation algorithm," in *Solid-State Circuits Conference, 2002. Digest*

of Technical Papers. ISSCC. 2002 IEEE International, vol. 1, pp. 368–474 vol.1, 2002.

[7] S.-H. Wang, W.-H. Peng, Y. He, G.-Y. Lin, C.-Y. Lin, S.-C. Chang, C.-N. Wang, and T. Chiang, "A software-hardware co-implementation of MPEG-4 advanced video coding (AVC) decoder with block level pipelining," *The Journal of VLSI Signal Processing*, vol. 41, pp. 93–110, August 2005.

[8] S.-M. Kim, J.-H. Park, S.-M. Park, B.-T. Koo, K.-S. Shin, K.-B. Suh, I.-K. Kim, N.-W. Eum, and K.-S. Kim, "Hardware-software implementation of MPEG-4 video codec," *ETRI Journal*, vol. 25, pp. 489–502, December 2003.

[9] Y.-C. Chang, W.-M. Chao, and L.-G. Chen, "Platform-based MPEG-4 video encoder SOC design," in *Signal Processing Systems, 2004. SIPS 2004. IEEE Workshop on*, pp. 251–256, October 2004.

[10] S. Park, H. Cho, H. Jung, and D. Lee, "An implemented of H.264 video decoder using hardware and software," in *Custom Integrated Circuits Conference, 2005. Proceedings of the IEEE 2005*, pp. 271–275, September 2005.

[11] G. Tech, A. Smolic, H. Brust, P. Merkle, K. Dix, Y. Wang, K. Müller, and T. Wiegand, "Optimization and comparision of coding algorithms for mobile 3DTV," in *3DTV-CON*, May 2009.

[12] H. Brust, A. Smolic, K. Mueller, G. Tech, and T. Wiegand, "Mixed resolution coding of stereoscopic video for mobile devices," in *3DTV-CON*, May 2009.

[13] T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, pp. 560–576, July 2003.

[14] I. E. Richardson, *H.264 and MPEG-4 Video Compression: Video Coding for Next Generation Multimedia*. Wiley, August 2003.

[15] H. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky, "Low-complexity transform and quantization in H.264/AVC," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, pp. 598–603, July 2003.

[16] D. Marpe, H. Schwarz, and T. Wiegand, "Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, pp. 620–636, July 2003.

[17] P. List, A. Joch, J. Lainema, G. Bjontegaard, and M. Karczewicz, "Adaptive deblocking filter," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, pp. 614–619, July 2003.

[18] C. L. Zitnick, S. B. Kang, M. Uyttendaele, S. Winder, and R. Szeliski, "High-quality video view interpolation using a layered representation," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 600–608, 2004.

[19] P. Merkle, K. Müller, A. Smolic, and T. Wiegand, "Efficient compression of multi-view video exploiting inter-view dependencies based on H.264/MPEG4-AVC," in *Multimedia and Expo, 2006 IEEE International Conference on,*, pp. 1717–1720, July 2006.

[20] M. Flierl, A. Mavlankar, and B. Girod, "Motion and disparity compensated coding for multiview video," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 17, pp. 1474–1484, November 2007.

[21] K. Yamamoto, M. Kitahara, H. Kimata, T. Yendo, T. Fujii, M. Tanimoto, S. Shimizu, K. Kamikura, and Y. Yashima, "Multiview video coding using view interpolation and color correction," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 17, pp. 1436–1449, November 2007.

[22] L. Zhang and W. Tam, "Stereoscopic image generation based on depth images for 3D TV," *Broadcasting, IEEE Transactions on*, vol. 51, pp. 191–199, June 2005.

[23] P. Merkle, A. Smolic, K. Müller, and T. Wiegand, "Multi-view video plus depth representation and coding," in *Image Processing, 2007. ICIP 2007. IEEE International Conference on*, vol. 1, pp. I –201–I –204, October 2007.

[24] S.-U. Yoon and Y.-S. Ho, "Multiple color and depth video coding using a hierarchical representation," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 17, pp. 1450–1460, November 2007.

[25] K. Müller, A. Smolic, K. Dix, P. Kauff, and T. Wiegand, "Reliability-based generation and view synthesis in layered depth video," in *Multimedia Signal Processing, 2008 IEEE 10th Workshop on*, pp. 34–39, October 2008.

[26] A. Smolic, K. Müller, P. Merkle, P. Kauff, and T. Wiegand, "An overview of available and emerging 3D video formats and depth enhanced stereo as efficient generic solution," in *Proc. PCS 2009, Picture Coding Symposium*, May 2009.

[27] A. Aksay, Ç. Bilen, E. Kurutepe, T. Özçelebi, G. Bozdağı Akar, M. R. Civanlar, and A. M. Tekalp, "Temporal and spatial scaling for stereoscopic video compression," in *14th European Signal Processing Conference*, September 2006.

[28] B. Julesz, *Foundations of Cyclopeon Perception*. The University of Chicago Press, 1971.

[29] I. Dinstein, M. G. Kim, A. Henik, and J. Tselgov, "Compression of stereo images using subsampling and transform coding," *Optical Engineering*, vol. 30, no. 9, pp. 1359–1364, 1991.

[30] W. Woo and A. Ortega, "Optimal blockwise dependent quantization for stereo image coding," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 9, pp. 861–867, September 1999.

[31] C. Fehn, P. Kauff, S. Cho, H. Kwon, N. Hur, and J. Kim, "Asymmetric coding of stereoscopic video for transmission over T-DMB," in *3DTV Conference, 2007*, pp. 1–4, May 2007.

[32] Y. Chen, S. Liu, Y.-K. Wang, M. Hannuksela, H. Li, and M. Gabbouj, "Low-complexity asymmetric multiview video coding," in *Multimedia and Expo, 2008 IEEE International Conference on*, pp. 773–776, April 2008.

[33] *JMVM Software Manual*, June 2008.

[34] *JSVM Software Manual*, June 2009.

[35] S. Sun and J. Reichel, "AHG report on spatial scalability resampling," Doc. JVT-R006, Joint Video Team, January 2008.

[36] S. Cho, B. Lee, N. Hur, J. Kim, and S.-I. Lee, "Preliminary subjective test results for mixed resolution stereo video coding," Doc. JVT-Z034, Joint Video Team, January 2006.

[37] Logic Product Development, *Zoom OMAP34x Mobile Development Kit Product Brief*, Rev.B. April 2008.

[38] Logic Product Development, *OMAP3430 SOM-LV Hardware Specification*, Rev.B. May 2008.

[39] Texas Instruments, *TMS320C6000 Programmer's Guide - SPRU198i*, March 2006.

[40] Texas Instruments, *DSP Bridge Application Integration Guide 3430*, 2006.

[41] Texas Instruments, *Programming Guide for DSP/BIOS™Bridge*, 2008.