UNIVERSITY OF CALIFORNIA
RIVERSIDE

Parallel and Scalable Architectures for Video Encoding

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering

by

Zhuo Zhao

December 2010

Dissertation Committee:

Dr. Ping Liang , Chairperson
Dr. Ilya Dumer
Dr. Zhengyuan Xu

The Dissertation of Zhuo Zhao is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

The printed pages of this dissertation hold far more than the culmination of years of study. These pages also reflect the relationships with many generous and inspiring people I have met since beginning my graduate work. The list is long, but I cherish each contribution to the completion of my dissertation.

The work described in this thesis could not have been accomplished without the help and support of others. Foremost, I would like to thank my research advisor Professor Ping Liang for his guidance and support. I am especially grateful to you.

To my committee members Dr. Ilya Dumer, Dr. Zhengyuan Xu for their encouraging words, thoughtful criticism, and time and attention during busy semesters.

To my colleagues for sharing their enthusiasm for and comments on my work: Mohammad Ahmad, Jun Dai, Sheng Wang, Mehran Ramezani and Vahid Ordoubadian.

To the Department staffs for assisting me with the administrative tasks necessary for completing my doctoral program: Adrienne L. Thomas, Steven D. Haughton and William Bingham.

To my invaluable network of supportive, generous and loving friends: Yufan Li, Lili Huang, Yi Huang, Zhen Zhang, Lucy Zhou, Zhenyu Qi and Ning Liu.

To my parents and family for their love, support and understanding during the long years of my education.

And finally, to Ying Gao, my girlfriend. Thank you for always there giving me support and cheering me up and stood by me through the good times and bad.

To my parents for all the support.

ABSTRACT OF THE DISSERTATION

Parallel and Scalable Architectures for Video Encoding

by

Zhuo Zhao

Doctor of Philosophy, Graduate Program in Electrical Engineering
University of California, Riverside, December 2010
Dr. Ping Liang , Chairperson

As the latest video compression standard, H.264/AVC exhibits great compression performance than its previous ancestors. Many new features are used to achieve much better rate-distortion efficiency and subjective quality, but the high computational complexity and intensive memory access are the penalties. Such high requirement of memory and computational resources leads to long processing cycles and high power consumption. This made real-time encoding of H.264/AVC hard to implement.

To address these difficulties, this thesis is focused on fast algorithm, data reuse and parallel architectures of H.264/AVC encoder. For data reuse, we proposed a partially forward processing algorithm (PFPA) which reuses the reference information to avoid duplicated reference data loading. For fast algorithms, we studied the statistical features of fractional motion estimation (FME) and proposed a FME mode reduction scheme. For parallel architectures, we proposed two solutions for block level and MB level parallelization respectively. At the block level, we proposed a FME parallel architecture which achieved both memory and processing cycle efficiency (reduced about 67% memory accesses and about 50% processing cycles compared with most of state of the art architectures). At the MB level, we proposed wavefront architecture. Theoretically,

this architecture can extend a multi-core encoder to a system with any desired number of cores without sacrificing encoding quality.

Both JM model and Tensilica XTMP are used to verify the proposed architectures. Architecture implementation detail are discussed and cycle-accurate test results show good performance improvements with very small overhead. From dual-core to three-core and quad-core, the overhead of the P-Core performance are 0.8% and 1.3% for I-frames; 1.7% and 2.4% for P-frames. The speed-ups from dual-core to three-core and quad-core are 1.49 and 1.97 for I-frames; 1.47 and 1.95 for P-frames. System up-scaling methodologies are also covered at the end of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 H.264/AVC Overview

In early 1998, the Video Coding Experts Group (VCEG) ITU-T SG16 Q.6 issued a call for proposals on a project called H.26L, with the target to double the coding efficiency (which means halving the bit rate necessary for a given level of fidelity) in comparison to any other existing video coding standards for a broad variety of applications. The first draft design for that new standard was adopted in October of 1999. In December of 2001, VCEG and the Moving Picture Experts Group (MPEG) ISO/IEC JTC 1/SC 29/WG 11 formed a Joint Video Team (JVT), with the charter to finalize the draft new video coding standard for formal approval submission as H.264/AVC in March 2003. As of today, H.264/AVC is still the newest international video coding standard [14].

Before discussing encoding algorithms and parallel architectures, we will present an overview of the H.264 protocol in this chapter, whose primary features include:

- **Variable block-size motion compensation with small block sizes:** This standard supports more flexibility in the selection of motion compensation block

sizes and shapes than any previous standard, with a minimum luma motion compensation block size as small as 4×4.

- **Quarter-sample-accurate motion compensation:** Most prior standards enable half-sample motion vector accuracy at most. The new design improves up on this by adding quarter-sample motion vector accuracy, as first found in an advanced profile of the MPEG-4 Visual (part-2) standard, but further reduces the complexity of the interpolation processing compared to the prior design.

- **Motion vectors over picture boundaries:** While motion vectors in MPEG-2 and its predecessors were required to point only to areas within the previously-decoded reference picture, the picture boundary extrapolation technique first found as an optional feature in H.263 is included in H.264/AVC.

- **Multiple reference picture motion compensation:**Previous video coding standards use only one previous picture to predict the values in an incoming picture. H.264/AVC allows an encoder to select the reference frames among a large number of pictures that have been decoded an stored in the decoder.

- **Decoupling of referencing order from display order:** In prior standards, the display order and reference order have a strict dependency. In H.264/AVC, these restrictions are largely removed.

- **Decoupling of picture representation methods from picture referencing capability:** In previous standards, B-frame cannot be used as references for prediction of other pictures. H.264/AVC removed this restriction to gain more flexibility and closer approximation in the prediction.

- **Weighted prediction:** This is a new feature in H.264/AVC, which can dramat-

ically improve the coding efficiency for scenes containing fades, and can also be used flexibly for other purposes as well.

- **Improved "skipped" and "direct" motion inference:** In previous standards, a "skipped" area of a predictively-coded picture could not motion in the scene content. H.264/AVC introduced motion for "skipped" areas to remove the detrimental effects when coding video containing global motion.

- **Directional spatial prediction for intra coding:** A new technique of extrapolating the edges of the previously-decoded parts of the current picture is applied in regions of pictures that are coded as intra (i.e., coded without reference to the content of some other picture). This improves the quality of the prediction signal, and also allows prediction from neighboring areas that were not coded using intra coding.

- **In-the-loop deblocking filtering:** Block-based video coding produces artifacts known as blocking artifacts. These can originate from both the prediction and residual difference coding stages of the decoding process. Application of an adaptive deblocking filter is a well-known method of improving the resulting video quality, and when designed well, this can improve both objective and subjective video quality. Building further on a concept from an optional feature of H.263+, the deblocking filter in the H.264/AVC design is brought within the motion-compensated prediction loop, so that this improvement in quality can be used in inter-picture prediction to improve the ability to predict other pictures as well.

- **Small block-size transform:** All major prior video coding standards used a transform block size of 8×8, while the new H.264/AVC design is based primarily

on a 4×4 transform. This allows the encoder to represent signals in a more locally-adaptive fashion, which reduces artifacts known colloquially as "ringing".

- **Hierarchical block transform:** While in most cases, using the small 4×4 transform block size is perceptually beneficial, there are some signals that contain sufficient correlation to call for some method of using a representation with longer basis functions. The H.264/AVC standard enables this in two ways: 1) by using a hierarchical transform to extend the effective block size use for low-frequency chroma information to an 8×8 array and 2) by allowing the encoder to select a special coding type for intra coding, enabling extension of the length of the luma transform for low-frequency information to a 16×16 block size in a manner very similar to that applied to the chroma.

- **Short word-length transform:** All prior standard designs have effectively required encoders and decoders to use more complex processing for transform computation. While previous designs have generally required 32-bit processing, the H.264/AVC design requires only 16-bit arithmetic.

- **Exact-match inverse transform:** In previous video coding standards, the transform used for representing the video was generally specified only within an error tolerance bound, due to the impracticality of obtaining an exact match to the ideal specified inverse transform. As a result, each decoder design would produce slightly different decoded video, causing a "drift" between encoder and decoder representation of the video and reducing effective video quality. Building on a path laid out as an optional feature in the H.263++ effort, H.264/AVC is the first standard to achieve exact equality of decoded video content from all decoders.

- **Arithmetic entropy coding:** An advanced entropy coding method known as arithmetic coding is included in H.264/AVC. While arithmetic coding was previously found as an optional feature of H.263, a more effective use of this technique is found in H.264/AVC to create a very powerful entropy coding method known as CABAC (context-adaptive binary arithmetic coding).

- **Context-adaptive entropy coding:** The two entropy coding methods applied in H.264/AVC, termed CAVLC (context-adaptive variable-length coding) and CABAC, both use context-based adaptivity to improve performance relative to prior standard designs.

- **Parameter set structure:** The parameter set design provides for robust and efficient conveyance header information. As the loss of a few key bits of information (such as sequence header or picture header information) could have a severe negative impact on the decoding process when using prior standards, this key information was separated for handling in a more flexible and specialized manner in the H.264/AVC design.

- **NAL unit syntax structure:** Each syntax structure in H.264/AVC is placed into a logical data packet called a NAL unit. Rather than forcing a specific bitstream interface to the system as in prior video coding standards, the NAL unit syntax structure allows greater customization of the method of carrying the video content in a manner appropriate for each specific network.

- **Flexible slice size:** Unlike the rigid slice structure found in MPEG-2 (which reduces coding efficiency by increasing the quantity of header data and decreasing the effectiveness of prediction), slice sizes in H.264/AVC are highly flexible, as was

the case earlier in MPEG-1.

- **Flexible macroblock ordering (FMO):** A new ability to partition the picture into regions called slice groups has been developed, with each slice becoming an independently- decodable subset of a slice group. When used effectively, flexible macroblock ordering can significantly enhance robustness to data losses by managing the spatial relationship between the regions that are coded in each slice. (FMO can also be used for a variety of other purposes as well.)

- **Arbitrary slice ordering (ASO):** Since each slice of a coded picture can be (approximately) decoded independently of the other slices of the picture, the H.264/AVC design enables sending and receiving the slices of the picture in any order relative to each other. This capability, first found in an optional part of H.263+, can improve end-to-end delay in real-time applications, particularly when used on networks having out-of-order delivery behavior (e.g., internet protocol networks).

- **Redundant pictures:** In order to enhance robustness to data loss, the H.264/AVC design contains a new ability to allow an encoder to send redundant representations of regions of pictures, enabling a (typically somewhat degraded) representation of regions of pictures for which the primary representation has been lost during data transmission.

- **Data Partitioning:** Since some coded information for representation of each region (e.g., motion vectors and other prediction information) is more important or more valuable than other information for purposes of representing the video content, H.264/AVC allows the syntax of each slice to be separated into up to

three different partitions for transmission, depending on a categorization of syntax elements. This part of the design builds further on a path taken in MPEG-4 Visual and in an optional part of H.263++. Here, the design is simplified by having a single syntax with partitioning of that same syntax controlled by a specified categorization of syntax elements.

- **SP/SI synchronization/switching pictures:** The H.264/AVC design includes a new feature consisting of picture types that allow exact synchronization of the decoding process of some decoders with an ongoing video stream produced by other decoders without penalizing all decoders with the loss of efficiency resulting from sending an I picture. This can enable switching a decoder between representations of the video content that used different data rates, recovery from data losses or errors, as well as enabling trick modes such as fast-forward, fast-reverse, etc.

In the following sections, we will only introduce primary new features of H.264/AVC. For more detail or other features, please refer to [14].

1.2 Intra-Prediction

If a block or macroblock is encoded in intra mode, a prediction block is formed based on previously encoded and reconstructed (but un-filtered) blocks. This prediction block P is subtracted from the current block prior to encoding. For the luminance (luma) samples, P may be formed for each 4×4 subblock or for a 16×16 macroblock. There are a total of 9 optional prediction modes for each 4×4 luma block; 4 optional modes for a 16×16 luma block; and one mode that is always applied to each 4×4 chroma block.

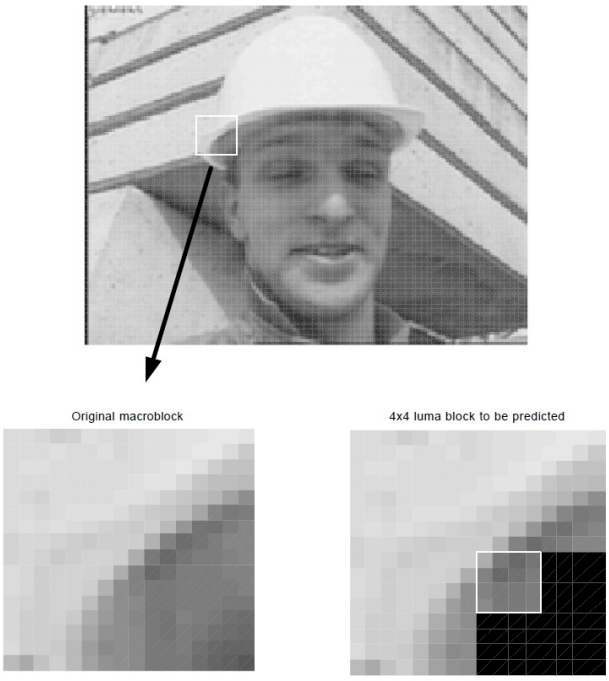

Figure 1.1: Original macroblock and 4×4 luma block to be predicted

M	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**
I	a	b	c	d				
J	e	f	g	h				
K	i	j	k	l				
L	m	n	o	p				

Figure 1.2: Labelling of prediction samples (4×4)

1.2.1 4×4 Luma Prediction Modes

Figure.1.1 shows a luminance macroblock in a QCIF frame and a 4×4 luma block that is required to be predicted. The samples above and to the left have previously been encoded and reconstructed and are therefore available in the encoder and decoder to form a prediction reference. The prediction block P is calculated based on the samples labelled A-M in Figure.1.2, as follows. Note that in some cases, not all of the samples

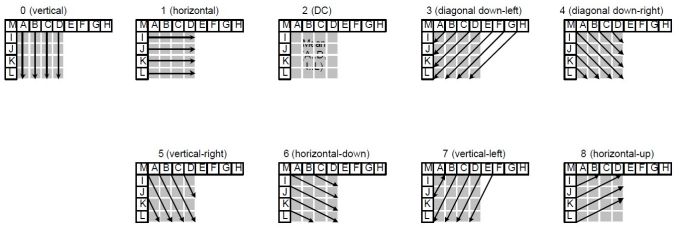

Figure 1.3: 4×4 luma prediction modes

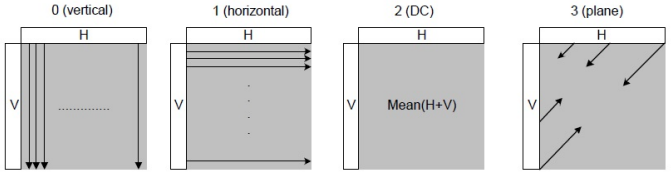

Figure 1.4: H.264/AVC Intra 16×16 prediction modes (all predicted from pixels H and V)

A-M are available within the current slice: in order to preserve independent decoding of slices, only samples within the current slice are available for prediction. DC prediction (mode 0) is modified depending on which samples A-M are available; the other modes (1-8) may only be used if all of the required prediction samples are available (except that, if E, F, G and H are not available, their value is copied from sample D).

The arrows in Figure.1.3 indicate the direction of prediction in each mode. For modes 3-8, the predicted samples are formed from a weighted average of the prediction samples A-Q. The encoder may select the prediction mode for each block that minimizes the residual between P and the block to be encoded.

1.2.2 16×16 Luma Prediction Modes

As an alternative to the 4×4 luma modes described above, the entire 16×16 luma component of a macroblock may be predicted. Four modes are available, shown in diagram form in Figure.1.4:

- Mode 0 (vertical): extrapolation from upper samples (H).
- Mode 1 (horizontal): extrapolation from left samples (V).
- Mode 2 (DC): mean of upper and left-hand samples (H+V).
- Mode 4 (Plane): a linear "plane" function is fitted to the upper and left-hand samples H and V. This works well in areas of smoothly-varying luminance.

1.2.3 8×8 Chroma Prediction Modes

Each 8×8 chroma component of a macroblock is predicted from chroma samples above and/or to the left that have previously been encoded and reconstructed. The 4 prediction modes are very similar to the 16×16 luma prediction modes described in section 3 and illustrated in Figure.1.4, except that the order of mode numbers is different: DC (mode 0), horizontal (mode 1), vertical (mode 2) and plane (mode 3). The same prediction mode is always applied to both chroma blocks.

1.3 Inter-Prediction

Inter prediction creates a prediction model from one or more previously encoded video frames. The model is formed by shifting samples in the reference frame(s) (motion compensated prediction). The H.264/AVC CODEC uses block-based motion compensation, the same principle adopted by every major coding standard since H.261. Important differences from earlier standards include the support for a range of block sizes (down to 4×4) and fine sub-pixel motion vectors (1/4 pixel in the luma component).

1.3.1 Tree structured motion compensation

H.264/AVC supports motion compensation block sizes ranging from 16×16 to 4×4 luminance samples with many options between the two. The luminance component of each macroblock (16×16 samples) may be split up in 4 ways as shown in Figure.1.5:16×16, 16×8, 8×16 or 8×8. Each of the sub-divided regions is a macroblock partition. If the 8×8 mode is chosen, each of the four 8×8 macroblock partitions within the macroblock may be split in a further 4 ways as shown in Figure.1.6: 8×8, 8×4, 4×8 or 4×4 (known as macroblock sub-partitions). These partitions and sub-partitions give rise to a large number of possible combinations within each macroblock. This method of partitioning macroblocks into motion compensated sub-blocks of varying size is known as tree structured motion compensation.

A separate motion vector is required for each partition or sub-partition. Each motion vector must be coded and transmitted; in addition, the choice of partition(s) must be encoded in the compressed bitstream. Choosing a large partition size (e.g. 16×16, 16×8, 8×16) means that a small number of bits are required to signal the choice of motion vector(s) and the type of partition; however, the motion compensated residual may contain a significant amount of energy in frame areas with high detail. Choosing a small partition size (e.g. 8×4, 4×4, etc.) may give a lower-energy residual after motion compensation but requires a larger number of bits to signal the motion vectors and choice of partition(s). The choice of partition size therefore has a significant impact on compression performance. In general, a large partition size is appropriate for homogeneous areas of the frame and a small partition size may be beneficial for detailed areas.

The resolution of each chroma component in a macroblock (Cr and Cb) is

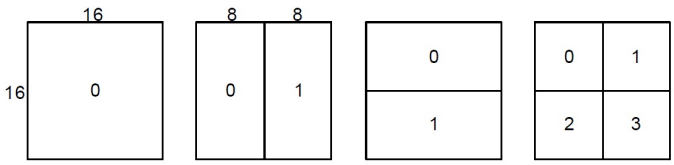

Figure 1.5: Macroblock partitions: 16×16, 8×16, 16×8, 8×8

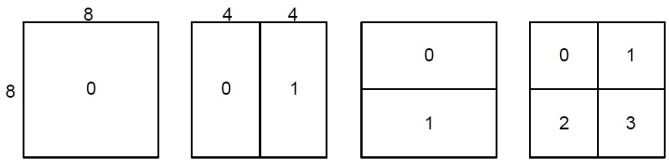

Figure 1.6: Macroblock sub-partitions: 8×8, 4×8, 8×4, 4×4

half that of the luminance (luma) component. Each chroma block is partitioned in the same way as the luma component, except that the partition sizes have exactly half the horizontal and vertical resolution (an 8×16 partition in luma corresponds to a 4×8 partition in chroma; an 8×4 partition in luma corresponds to 4×2 in chroma; and so on). The horizontal and vertical components of each motion vector (one per partition) are halved when applied to the chroma blocks.

Figure.1.7 shows a residual frame (without motion compensation). The H.264/AVC reference encoder selects the best partition size for each part of the frame, i.e. the partition size that minimizes the coded residual and motion vectors. The macroblock partitions chosen for each area are shown superimposed on the residual frame. In areas where there is little change between the frames (residual appears grey), a 16×16 partition is chosen; in areas of detailed motion (residual appears black or white), smaller partitions are more efficient.

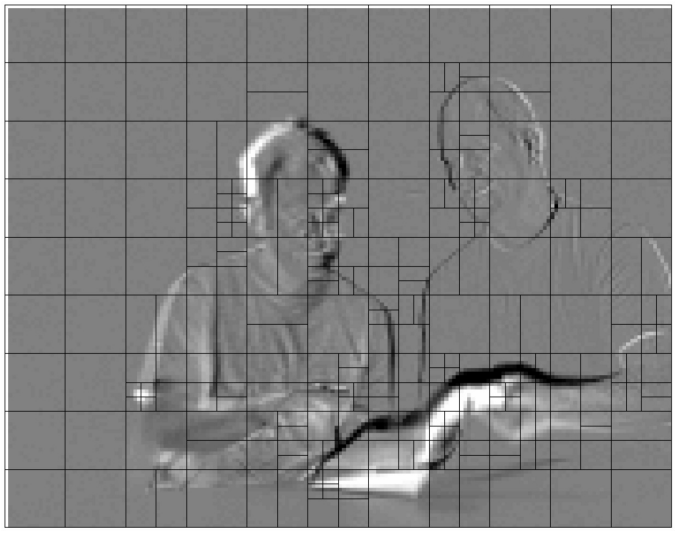

Figure 1.7: Residual (without MC) showing optimum choice of partitions

1.3.2 Sub-pixel motion vectors

Each partition in an inter-coded macroblock is predicted from an area of the same size in a reference picture. The offset between the two areas (the motion vector) has 1/4 pixel resolution (for the luma component). The luma and chroma samples at sub-pixel positions do not exist in the reference picture and so it is necessary to create them using interpolation from nearby image samples. Figure.1.8 gives an example. A 4×4 sub-partition in the current frame (a) is to be predicted from a neighboring region of the reference picture. If the horizontal and vertical components of the motion vector are integers (b), the relevant samples in the reference block actually exist (grey dots). If one or both vector components are fractional values (c), the prediction samples (grey dots) are generated by interpolation between adjacent samples in the reference frame (white dots).

Sub-pixel motion compensation can provide significantly better compression performance than integer-pixel compensation, at the expense of increased complexity. Quarter-pixel accuracy outperforms half-pixel accuracy.

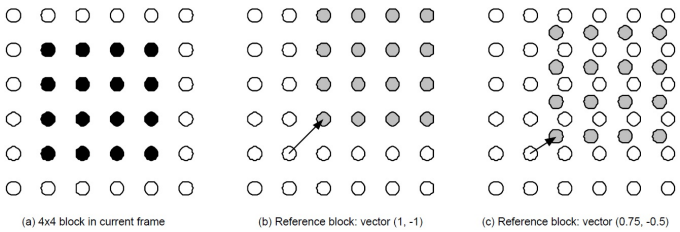

Figure 1.8: Example of integer and sub-pixel prediction

In the luma component, the sub-pixel samples at half-pixel positions are generated first and are interpolated from neighboring integer-pixel samples using a 6-tap Finite Impulse Response filter. This means that each half-pixel sample is a weighted sum of 6 neighboring integer samples. Once all the half-pixel samples are available, each quarter-pixel sample is produced using bilinear interpolation between neighboring half- or integer-pixel samples.

If the video source sampling is 4:2:0, 1/8 pixel samples are required in the chroma components (corresponding to 1/4 pixel samples in the luma). These samples are interpolated (linear interpolation) between integer-pixel chroma samples.

1.3.3 Motion vector prediction

Encoding a motion vector for each partition can take a significant number of bits, especially if small partition sizes are chosen. Motion vectors for neighboring partitions are often highly correlated and so each motion vector is predicted from vectors of nearby, previously coded partitions. A predicted vector, MV_p, is formed based on previously calculated motion vectors. MV_d, the difference between the current vector and the predicted vector, is encoded and transmitted. The method of forming the prediction MV_p depends on the motion compensation partition size and on the availability of nearby vectors. The "basic" predictor is the median of the motion vectors of the

macroblock partitions or subpartitions immediately above, diagonally above and to the right, and immediately left of the current partition or sub-partition. The predictor is modified if (a) 16×8 or 8×16 partitions are chosen and/or (b) if some of the neighboring partitions are not available as predictors. If the current macroblock is skipped (not transmitted), a predicted vector is generated as if the MB was coded in 16×16 partition mode.

At the decoder, the predicted vector MV_p is formed in the same way and added to the decoded vector difference MV_d. In the case of a skipped macroblock, there is no decoded vector and so a motion compensated macroblock is produced according to the magnitude of MV_p.

1.4 Transform and Quantization

Each residual macroblock is transformed, quantized and coded. Previous standards such as MPEG-1, MPEG-2, MPEG-4 and H.263 made use of the 8×8 Discrete Cosine Transform (DCT) as the basic transform. The "baseline" profile of H.264/AVC uses three transforms depending on the type of residual data that is to be coded: a transform for the 4×4 array of luma DC coefficients in intra macroblocks (predicted in 16×16 mode), a transform for the 2×2 array of chroma DC coefficients (in any macroblock) and a transform for all other 4×4 blocks in the residual data. If the optional "adaptive block size transform" mode is used, further transforms are chosen depending on the motion compensation block size (4×8, 8×4, 8×8, 16×8, etc).

Data within a macroblock are transmitted in the order shown in Figure.1.9. If the macroblock is coded in 16×16 Intra mode, then the block labelled "-1" is transmitted first, containing the DC coefficient of each 4×4 luma block. Next, the luma residual

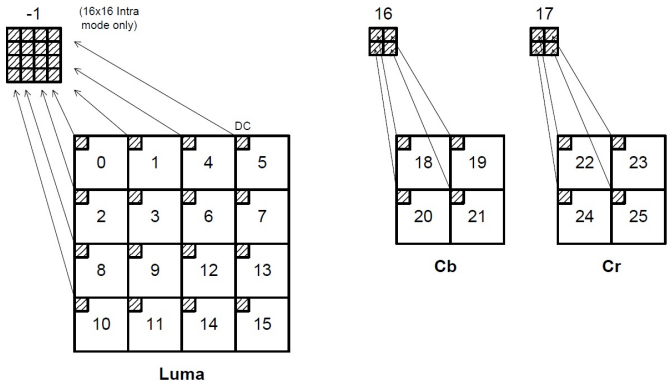

Figure 1.9: Scanning order of residual blocks within a macroblock

blocks 0-15 are transmitted in the order shown (with the DC coefficient set to zero in a 16×16 Intra macroblock). Blocks 16 and 17 contain a 2×2 array of DC coefficients from the Cb and Cr chroma components respectively. Finally, chroma residual blocks 18- 25 (with zero DC coefficients) are sent.

1.4.1 4×4 Residual Transform and Quantization (Blocks 0-15, 18-25)

This transform operates on 4×4 blocks of residual data (labelled 0-15 and 18-25 in Figure 1-1) after motion-compensated prediction or Intra prediction. The transform is based on the DCT but with some fundamental differences:

- It is an integer transform (all operations can be carried out with integer arithmetic, without loss of accuracy).
- The inverse transform is fully specified in the H.264/AVC standard and if this specification is followed correctly, mismatch between encoders and decoders should not occur.
- The core part of the transform is multiply-free, i.e. it only requires additions and shifts.

- A scaling multiplication (part of the complete transform) is integrated into the quantizer (reducing the total number of multiplications).

1.4.1.1 Development from the 4×4 DCT

The 4×4 DCT of an input array X is given by:

$$Y = AXA^T = \begin{bmatrix} a & a & a & a \\ b & c & -c & -b \\ a & -a & -a & a \\ c & -b & b & -c \end{bmatrix} \begin{bmatrix} X \end{bmatrix} \begin{bmatrix} a & b & a & c \\ a & c & -a & -b \\ a & -c & -a & b \\ a & -b & a & -c \end{bmatrix} \tag{1.1}$$

where:

$a = \frac{1}{2}$, $b = \sqrt{\frac{1}{2}}cos(\frac{\pi}{8})$, $c = \sqrt{\frac{1}{2}}cos(\frac{3\pi}{8})$

This matrix multiplication can be factorized to the following equivalent form:

$$Y = (CXC^T) \otimes E = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & d & -d & -1 \\ 1 & -1 & -1 & 1 \\ d & -1 & 1 & -d \end{bmatrix} [X] \begin{bmatrix} 1 & 1 & 1 & d \\ 1 & d & -1 & -1 \\ 1 & -d & -1 & 1 \\ 1 & -1 & 1 & -d \end{bmatrix} \right) \otimes \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix} \tag{1.2}$$

CXC^T is a "core" 2-D transform. E is a matrix of scaling factors and the symbol $\otimes$ indicates that each element of (CXC^T) is multiplied by the scaling factor in the same position in matrix E (scalar multiplication rather than matrix multiplication). The constants a and b are as before; d is c/b (approximately 0.414).

To simplify the implementation of the transform, d is approximated by 0.5. To ensure that the transform remains orthogonal, b also needs to be modified so that:

$a = \frac{1}{2}$, $b = \sqrt{\frac{2}{5}}$, $d = \frac{1}{2}$

The 2nd and 4th rows of matrix C and the 2nd and 4th columns of matrix C^T are scaled by a factor of 2 and the post-scaling matrix E is scaled down to compensate. (This avoids multiplications by 1/2 in the "core" transform CXC^T which would result in loss of accuracy using integer arithmetic). The final forward transform becomes:

$$Y = C_f X C_f^T \otimes E_f \tag{1.3}$$

$$= \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} [X] \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix} \right) \otimes \begin{bmatrix} a^2 & ab/2 & a^2 & ab/2 \\ ab/2 & b^2/4 & ab/2 & b^2/4 \\ a^2 & ab/2 & a^2 & ab/2 \\ ab/2 & b^2/4 & ab/2 & b^2/4 \end{bmatrix} \tag{1.4}$$

This transform is an approximation to the 4×4 DCT. Because of the change to factors d and b, the output of the new transform will not be identical to the 4×4 DCT.

The inverse transform (defined in [14]) is given by:

$$X' = C_i^T (Y \otimes E_i) C_i \tag{1.5}$$

$$= \begin{bmatrix} 1 & 1 & 1 & 1/2 \\ 1 & 1/2 & -1 & -1 \\ 1 & -1/2 & -1 & 1 \\ 1 & -1 & 1 & -1/2 \end{bmatrix} \left([Y] \otimes \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix} \right) \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1/2 & -1/2 & -1 \\ 1 & -1 & -1 & 1 \\ 1/2 & -1 & 1 & -1/2 \end{bmatrix} \tag{1.6}$$

This time, Y is pre-scaled by multiplying each coefficient by the appropriate

weighting factor from matrix E_i . Note the factors $\pm 1/2$ in the matrices C and C^T ; these can be implemented by a right-shift without a significant loss of accuracy because the coefficients Y are pre-scaled.

The forward and inverse transforms are orthogonal, i.e. $T^{-1}(T(X)) = X$.

1.4.1.2 Quantization

H.264/AVC uses a scalar quantizer . The definition and implementation are complicated by the requirements to (a) avoid division and/or floating point arithmetic and (b) incorporate the post- and pre-scaling matrices E_f and E_i described above.

The basic forward quantizer operation is as follows:

$Z_{ij} = round(Y_{ij}/Q_{step})$

where Y_{ij} is a coefficient of the transform described above, Q_{step} is a quantizer step size and Z_{ij} is a quantized coefficient.

A total of 52 values of Q_{step} are supported by the standard and these are indexed by a Quantization Parameter, QP. The values of Q_{step} corresponding to each QP are shown in Table 2-1. Note that Q_{step} doubles in size for every increment of 6 in QP; Q_{step} increases by 12.5% for each increment of 1 in QP. The wide range of quantizer step sizes makes it possible for an encoder to accurately and flexibly control the trade-off between bit rate and quality. The values of QP may be different for luma and chroma; both parameters are in the range 0-51 but QP_{Chroma} is derived from QP_Y so that it QP_C is less that QP_Y for values of QP_Y above 30. A user-defined offset between QP_Y and QP_C may be signalled in a Picture Parameter Set.

The post-scaling factor a^2, ab^2 or $b^2/4$ is incorporated into the forward quantizer. First, the input block X is transformed to give a block of unscaled coefficients

Table 1.1: Quantization step sizes in H.264/AVC CODEC

QP	0	1	2	3	4	5	6	7	8	9
QP_{step}	0.625	0.6875	0.8125	0.875	1	1.125	1.25	1.375	1.625	1.75
QP	10	11	12	...	18	...	24	...	30	...
QP_{step}	2	2.25	2.5	...	5		10		20	
QP	36	...	42	...	48	...	51			
QP_{step}	40		80		160		224			

Table 1.2: PF Value for different Positions

Position	PF
(0,0),(2,0),(0,2)or(2,2)	a^2
(1,1),(1,3),(3,1)or(3,3)	$b^2/4$
Other	$ab/2$

$W = CXC^T$. Then, each coefficient W_{ij} is quantized and scaled in a single operation:

$$Z_{ij} = round\left(W_{ij} \cdot \frac{PF}{Q_{step}}\right) \tag{1.7}$$

PF is a^2, $ab/2$ or $b^2/2$ depending on the position (i, j) as Table.1.2.

The factor PF/Q_{step} is implemented in the H.264 reference model software [1] as a multiplication by MF (a multiplication factor) and a right-shift, thus avoiding any division operations:

$$Z_{ij} = round\left(W_{ij} \cdot \frac{MF}{2_{qbits}}\right) \tag{1.8}$$

where $\frac{MF}{2^{qbits}} = \frac{PF}{Q_{step}}$ and $qbits = 15 + floor\,(QP/6)$.

In integer arithmetic, the equation above can be implemented as follows:

$$|Z_{ij}| = (|W_{ij}| \cdot MF + f) >> qbits \tag{1.9}$$

$$sign\left(Z_{ij}\right) = sign\left(W_{ij}\right) \tag{1.10}$$

where $>>$ indicates a binary shift right. In the reference model software, f is $2^{qbits}/3$ for Intra blocks or $2^{qbits}/6$ for Inter blocks.

1.4.1.3 Rescaling

The basic rescale (or "inverse quantizer") operation is:

$$Y'_{ij} = Z_{ij} \cdot Q_{step} \tag{1.11}$$

The pre-scaling factor for the inverse transform (matrix E_i , containing values a^2, ab and b^2 depending on the coefficient position) is incorporated in this operation, together with a further constant scaling factor of 64 to avoid rounding errors:

$$W'_{ij} = Z_{ij} \cdot Q_{step} \cdot PF \cdot 64 \tag{1.12}$$

W'_{ij} is a scaled coefficient which is then transformed by the "core" inverse transform ($C_i^T W C_i$). The values at the output of the inverse transform are divided by 64 to remove the scaling factor (this can be implemented using only an addition and a right-shift).

The H.264 standard does not specify Q_{step} or PF directly. Instead, the parameter $V = (Q_{step} \cdot PF \cdot 64)$ are defined for $0 \leq QP \leq 51$ and each coefficient position and the rescaling operation is:

$$W'_{ij} = Z_{ij}. \cdot V_{ij} \cdot 2^{floor(QP/6)} \tag{1.13}$$

1.4.1.4 4×4 Luma DC coefficient transform and quantization (16×16 Intra-mode only)

If the macroblock is encoded in 16×16 Intra prediction mode (where the entire 16×16 luminance component is predicted from neighboring pixels), each 4×4 residual block is first transformed using the "core" transform described above ($C_f X C_f^T$). The DC coefficient of each 4×4 block is then transformed again using a 4×4 Hadamard transform:

$$Y_D = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} [W_D] \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) /2 \tag{1.14}$$

W_D is the block of 4×4 DC coefficients and Y_D is the block after transformation. The output coefficients $Y_{D(i,j)}$ are divided by 2 (with rounding).

The output coefficients $Y_{D(i,j)}$ are then quantized to produce a block of quantized DC coefficients:

$$|Z_{D(i,j)}| = (|Y_{D(i,j)}| \cdot MF_{(0,0)} + 2f) >> (qbits + 1) \tag{1.15}$$

$$sign(Z_{D(i,j)}) = sign(Y_{D(i,j)}) \tag{1.16}$$

where MF, f and $qbits$ are defined as before and MF depends on the position (i, j) within the 4×4 DC coefficient block as before.

At the decoder, an inverse Hadamard transform is applied followed by rescaling (note that the order is not reversed as might be expected):

$$W_{QD} = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} [Z_D] \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) \tag{1.17}$$

If QP is greater than or equal to 12, rescaling is performed by:

$$W'_{D(i,j)} = W_{QD(i,j)} \cdot V_{(0,0)} \cdot 2^{floor(QP/6)-2} \tag{1.18}$$

If QP is less than 12, rescaling is performed by:

$$W'_{D(i,j)} = [W_{QD(i,j)} \cdot V_{(0,0)} + 2^{1-floor(QP/6)}] >> (2 - floor(QP/6)) \tag{1.19}$$

V is defined as before. The rescaled DC coefficients W'_D are then inserted into their respective 4×4 blocks and each 4×4 block of coefficients is inverse transformed using the core DCT-based inverse transform ($C_i^T W' C_i$).

In an intra-coded macroblock, much of the energy is concentrated in the DC coefficients and this extra transform helps to de-correlate the 4×4 luma DC coefficients (i.e. to take advantage of the correlation between the coefficients).

1.4.1.5 2×2 chroma DC coefficient transform and quantization

Each chroma component in a macroblock is made up of four 4×4 blocks of samples. Each 4×4 block is transformed as described in the sections above. The DC coefficients of each 4×4 block of coefficients are grouped in a 2×2 block (W_D) and are further transformed prior to quantization:

$$Y_D = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} [W_D] \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{1.20}$$

Quantization of the 2×2 output block Y_D is performed by:

$$|Z_{D(i,j)}| = (|Y_{D(i,j)}| \cdot MF_{(0,0)} + 2f) >> (qbits + 1) \tag{1.21}$$

$$sign(Z_{D(i,j)}) = sign(Y_{D(i,j)}) \tag{1.22}$$

where MF, f and $qbits$ are defined as before. During decoding, the inverse transform is applied before rescaling:

$$W_{QD} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} [Z_D] \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{1.23}$$

If QP is greater than or equal to 6, rescaling is performed by:

$$W'_{D(i,j)} = W_{QD(i,j)} \cdot V_{(0,0)} \cdot 2^{floor(QP/6)-1} \tag{1.24}$$

If QP is less than 6, rescaling is performed by:

$$W'_{D(i,j)} = [W_{QD(i,j)} \cdot V_{(0,0)}] >> 1 \tag{1.25}$$

The rescaled coefficients are replaced in their respective 4×4 blocks of chroma coefficients which are then transformed as above ($C_i^T W' C_i$). As with the Intra luma DC coefficients, the extra transform helps to de-correlate the 2×2 chroma DC coefficients and hence improves compression performance.

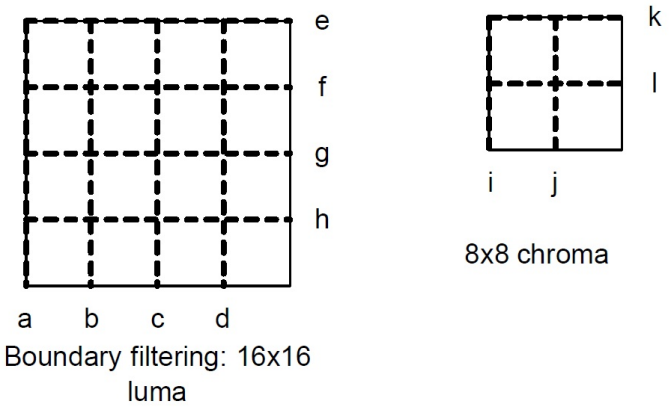

Figure 1.10: Edge filtering order in a macroblock

1.5 Deblock-Filter

A filter is applied to every decoded macroblock in order to reduce blocking distortion. The deblocking filter is applied after the inverse transform in the encoder (before reconstructing and storing the macroblock for future predictions) and in the decoder (before reconstructing and displaying the macroblock). The filter has two benefits: (1) block edges are smoothed, improving the appearance of decoded images (particularly at higher compression ratios) and (2) the filtered macroblock is used for motion-compensated prediction of further frames in the encoder, resulting in a smaller residual after prediction. (Note: intra-coded macroblocks are filtered, but intra prediction is carried out using unfiltered reconstructed macroblocks to form the prediction). Picture edges are not filtered.

1.5.1 Reconstruction filter

Filtering is applied to vertical or horizontal edges of 4×4 blocks in a macroblock, in the following order:

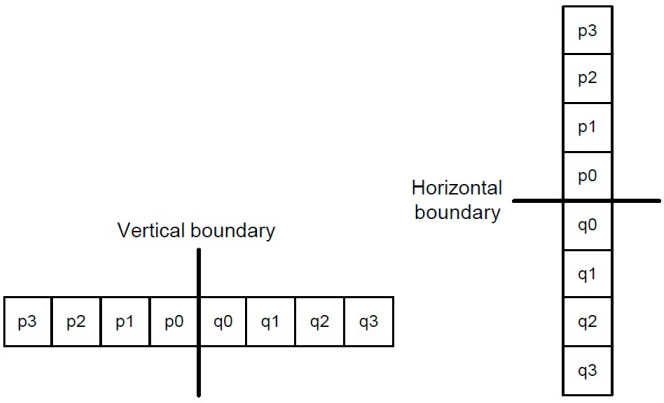

Figure 1.11: Pixels adjacent to vertical and horizontal boundaries

- Filter 4 vertical boundaries of the luma component (in order a,b,c,d in Figure.1.10)
- Filter 4 horizontal boundaries of the luma component (in order e,f,g,h, Figure.1.10)
- Filter 2 vertical boundaries of each chroma component (i,j)
- Filter 2 horizontal boundaries of each chroma component (k,l)

Each filtering operation affects up to three pixels on either side of the boundary. Figure.1.11 shows 4 pixels on either side of a vertical or horizontal boundary in adjacent blocks p and q (p0,p1,p2,p3 and q0,q1,q2,q3). Depending on the current quantizer, the coding modes of neighboring blocks and the gradient of image samples across the boundary, several outcomes are possible, ranging from (a) no pixels are filtered to (b) p0, p1, p2, q0, q1, q2 are filtered to produce output pixels P0, P1, P2, Q0, Q1 and Q2.

1.5.2 Boundary Strength

The choice of filtering outcome depends on the boundary strength and on the gradient of image samples across the boundary. The boundary strength parameter Bs is chosen according to the rules in Figure.1.12.

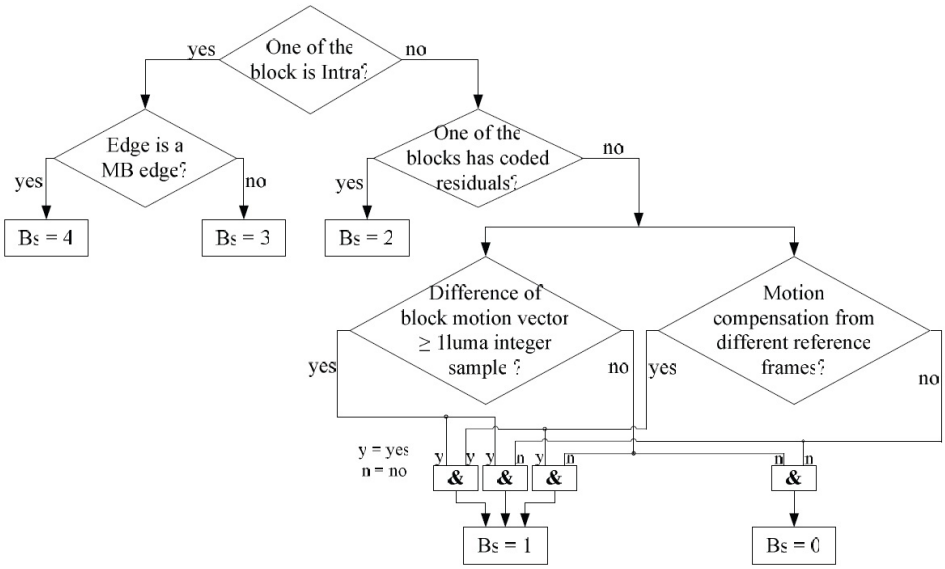

Figure 1.12: Boundary strength assignment flow

The filter is "stronger" at places where there is likely to be significant blocking distortion, such as the boundary of an intra coded macroblock or a boundary between blocks that contain coded coefficients.

1.5.3 Filter algorithm

Edges with $Bs = 0$ will not be filtered. For edges with non-zero Bs, a set of quantization dependent parameters, defined as α and β, are evaluated that each set of samples ($p_0 \sim p_2$, $q_0 \sim q_2$) is filtered only if all the following conditions all hold true:

$$Bs \neq 0 \tag{1.26}$$

$$|p_0 - q_0| < \alpha(Index_A) \tag{1.27}$$

$$|p_1 - p_0| < \beta(Index_B) \tag{1.28}$$

$$|q_1 - q_0| < \beta(Index_B) \tag{1.29}$$

$$|p_0 - q_0| < \alpha(Index_A) >> 2 + 2 \tag{1.30}$$

$$|p_2 - p_0| < \beta(Index_B) \tag{1.31}$$

$$|q_2 - q_0| < \beta(Index_B) \tag{1.32}$$

A summary of all the filtering cases of Bs = 1 4 in Table.1.3. For more details about basic algorithm of deblocking filter in H.264/AVC, please refer to [14] and [21].

1.6 Entropy Coding

The standard [14] specifies two types of entropy coding: Context-based Adaptive Binary Arithmetic Coding (CABAC) and Variable-Length Coding (VLC). In this thesis, we only use the Variable-Length Coding scheme, which is a part of the baseline Profile of H.264/AVC.

1.6.1 Coded elements

Parameters that require to be encoded and transmitted include the following.

Above the slice layer, syntax elements are encoded as fixed- or variable-length binary codes. At the slice layer and below, elements are coded using either variable-length codes (VLCs) [6] or contextadaptive arithmetic coding (CABAC) [23] depending on the entropy encoding mode.

Table 1.3: Deblocking filter summary

$Bs = 4$	Luma	p_0 $\sim$ p_2	(1.30)&(1.31) both true	$p'_0 = (p_2 + 2p_1 + 2p_0 + 2q_0 + q_1 + 4) >> 3$ $p'_1 = (p_2 + p_1 + p_0 + q_0 + 2) >> 2$ $p'_2 = (2p_3 + 3p_2 + p_1 + p_0 + q_0 + 4) >> 3$
			(1.30)&(1.31) not both true	$p'_0 = (2p_1 + p_0 + q_1 + 2) >> 2$ p_1 and p_2 unchanged
		q_0 $\sim$ q_2	(1.30)&(1.32) both true	$q'_0 = (q_2 + 2q_1 + 2q_0 + 2p_0 + p_1 + 4) >> 3$ $q'_1 = (q_2 + q_1 + q_0 + p_0 + 2) >> 2$ $q'_2 = (2q_3 + 3q_2 + q_1 + q_0 + p_0 + 4) >> 3$
			(1.30)&(1.32) not both true	$q'_0 = (2q_1 + q_0 + p_1 + 2) >> 2$ q_1 and q_2 unchanged
	Chroma	p_0 $\sim$ p_2	-	$p'_0 = (2p_1 + p_0 + q_1 + 2) >> 2$ p_1 and p_2 unchanged
		q_0 $\sim$ q_2	-	$q'_0 = (2q_1 + q_0 + p_1 + 2) >> 2$ q_1 and q_2 unchanged
$Bs = 1$ /2/3	Luma	p_0 & q_0	-	$p'_0 = p_0 + \Delta_0$ $q'_0 = q_0 - \Delta_0$ $\Delta_0 = min(max(-c_0, \Delta_{0i}), c_0)$ $\Delta_{0i} = [4(q_0 - p_0) + (p_1 - q_1) + 4] >> 3$ $c_0 = c_1 + [(1.30)?1 : 0] + [(1.31)?1 : 0]$ c_1: LUT operation
		p_1	(1.31)true	$p'_1 = p_1 + \Delta_{p1}$ $\Delta_{p1} = min[max(-c_1, \Delta_{p1i}), c_1]$ $\Delta_{p1i} = [p_2 + (p_0 + q_0 + 1) >> 1 - 2p_1] >> 1$ c_1: LUT operation
			(1.31)true	p_1 unchanged
		q_1	(1.32)true	$q'_1 = q_1 + \Delta_{q1}$ $\Delta_{q1} = min[max(-c_1, \Delta_{q1i}), c_1]$ $\Delta_{q1i} = [q_2 + (q_0 + p_0 + 1) >> 1 - 2q_1] >> 1$ c_1: LUT operation
			(1.32)false	q_1 unchanged
		p_2	-	unchanged
		q_2	-	unchanged
	Chroma	p_0 & q_0	-	$p'_0 = p_0 + \Delta_0$ $q'_0 = q_0 - \Delta_0$ $\Delta_0 = min[max(-c_0, \Delta_{0i}), c_0]$ $\Delta_{0i} = [4(q_0 - p_0) + (p_1 - q_1) + 4] >> 3$ c_1: LUT operation
		p_1 q_1 p_2 q_2	- - - -	unchanged unchanged unchanged unchanged

Table 1.4: Parameters to be encoded

Parameters	Description
Sequence-, picture- and slice-layer syntax elements	
Macroblock type *mb_type*	Prediction method for each coded macroblock
Coded block pattern	Indicates which blocks within a macroblock contain coded coefficients
Quantizer parameter	Transmitted as a delta value from the previous value of QP
Reference frame index	Identify reference frame(s) for inter prediction
Motion vector	Transmitted as a difference (mvd) from predicted motion vector
Residual data	Coefficient data for each 4×4 or 2×2 block

1.6.2 Variable length coding (VLC)

When *entropy_coding_mode* in the spec [14] is set to 0, residual block data is coded using a context-adaptive variable length coding (CAVLC) scheme and other variable-length coded units are coded using Exp-Golomb codes.

1.6.3 Exp-Golomb entropy coding

Exp-Golomb codes (Exponential Golomb codes) are variable length codes with a regular construction. Table.1.5 lists the first 9 codewords; it is clear from this table that the codewords progress in a logical order. Each codeword is constructed as follows:

$[Mzeros][1][INFO]$

where $INFO$ is an M-bit field carrying information. The first codeword has no leading zero or trailing $INFO$; codewords 1 and 2 have a single-bit $INFO$ field; codewords 3-6 have a 2-bit $INFO$ field; and so on. The length of each codeword is (2M+1) bits.

Each Exp-Golomb codeword can be constructed by the encoder based on its

Table 1.5: Exp-Golomb codewords

code_num	0	1	2	3	4	5	6	7	8	...
codeword	1	010	011	00100	00101	00110	00111	0001000	0001001	...

index *code_num*:

$$M = log_2(code\_num + 1)$$

$$INFO = code\_num + 1 - 2^M$$

A parameter **v** to be encoded is mapped to *code_num* in one of 3 ways as follows. Each of these mappings (ue, se and me) is designed to produce short codewords for frequently occurring values and longer codewords for less common parameter values.

- **ue(v)**: Unsigned direct mapping, $code\_num = v$. Used for macroblock type, reference frame index and others.

- **se(v)**: Signed mapping, used for motion vector difference, delta QP and others. v is mapped to *code_num* as follows:

  $code\_num = 2|v|\ (v < 0)$

  $code\_num = 2|v| - 1\ (v \geq 0)$

- **me(v)**: Mapped symbols; parameter v is mapped to *code_num* according to a table specified in the standard. This mapping is used for the *coded_block_pattern* parameter. Table.1.6 lists a small part of the table for Inter predicted macroblocks: *coded_block_pattern* indicates which 8×8 blocks in a macroblock contain non-zero coefficients.

Table 1.6: Part of *coded_block_pattern* table

coded_block_pattern (Inter prediction)	*code_num*
0 (no none-zero blocks)	0
16 (chroma DC block none-zero)	1
1 (top-left 8×8 luma block none-zero)	2
2 (top-right 8×8 luma block none-zero)	3
4 (lower-left 8×8 luma block non-zero)	4
8 (lower-right 8×8 luma block non-zero)	5
32 (chroma DC and AC blocks non-zero)	6
3 (top-left and top-right 8×8 luma blocks non-zero)	7
...	...

1.6.4 Context-based adaptive variable length coding (CAVLC)

This is the method used to encode residual, zig-zag ordered 4×4 (and 2×2) blocks of transform coefficients. CAVLC is designed to take advantage of several characteristics of quantized 4×4 blocks:

- After prediction, transformation and quantization, blocks are typically sparse (containing mostly zeros). CAVLC uses run-level coding to compactly represent strings of zeros.

- The highest non-zero coefficients after the zig-zag scan are often sequences of ± 1. CAVLC signals the number of high-frequency ± 1 coefficients ("Trailing 1s" or "T1s") in a compact way.

- The number of non-zero coefficients in neighbouring blocks is correlated. The number of coefficients is encoded using a look-up table; the choice of look-up table depends on the number of non-zero coefficients in neighbouring blocks.

- The level (magnitude) of non-zero coefficients tends to be higher at the start of the reordered array (near the DC coefficient) and lower towards the higher frequencies. CAVLC takes advantage of this by adapting the choice of VLC look-up table for

the "level" parameter depending on recently-coded level magnitudes.

1.6.4.1 Encode the number of coefficients and trailing ones

The first VLC, $coeff\_token$, encodes both the total number of non-zero coefficients (TotalCoeffs) and the number of trailing ± 1 values (T1). TotalCoeffs can be anything from 0 (no coefficients in the 4×4 block) 1 to 16 (16 non-zero coefficients). T1 can be anything from 0 to 3; if there are more than 3 trailing $\pm 1s$, only the last 3 are treated as "special cases" and any others are coded as normal coefficients.

There are 4 choices of look-up table to use for encoding $coeff\_token$, described as Num-VLC0, Num- VLC1, Num-VLC2 and Num-FLC (3 variable-length code tables and a fixed-length code). The choice of table depends on the number of non-zero coefficients in upper and left-hand previously coded blocks N_U and N_L. A parameter N is calculated as follows:

- If blocks U and L are available (i.e. in the same coded slice), $N = (N_U + N_L)/2$
- If only block U is available, $N = N_U$; if only block L is available, $N = N_L$; if neither is available, $N = 0$.

N selects the look-up table (Table.1.7) and in this way the choice of VLC adapts depending on the number of coded coefficients in neighbouring blocks (context adaptive). Num-VLC0 is "biased" towards small numbers of coefficients; low values of TotalCoeffs (0 and 1) are assigned particularly short codes and high values of TotalCoeff particularly long codes. Num-VLC1 is biased towards medium numbers of coefficients (TotalCoeff values around 2-4 are assigned relatively short codes), Num-VLC2 is biased towards higher numbers of coefficients and FLC assigns a fixed 6-bit code to every value of TotalCoeff.

Table 1.7: Choice of look-up table for $coeff\_token$

N	0,1	2,3	4,5,6,7	8 or above
Table for $coeff\_token$	Num-VLC0	Num-VLC1	Num-VLC2	FLC

1.6.4.2 Encode the sign of each T1

For each T1 (trailing ±1) signalled by $coeff\_token$, a single bit encodes the sign (0=+, 1=-). These are encoded in reverse order, starting with the highest-frequency T1.

1.6.4.3 Encode the levels of the remaining non-zero coefficients

The level (sign and magnitude) of each remaining non-zero coefficient in the block is encoded in reverse order, starting with the highest frequency and working back towards the DC coefficient. The choice of VLC table to encode each level adapts depending on the magnitude of each successive coded level (context adaptive). There are 7 VLC tables to choose from, Level_VLC0 to Level_VLC6. Level_VLC0 is biased towards lower magnitudes; Level_VLC1 is biased towards slightly higher magnitudes and so on. The choice of table is adapted in the following way:

- Initialize the table to Level_VLC0 (unless there are more than 10 non-zero coefficients and less than 3 trailing ones, in which case start with Level_VLC1).
- Encode the highest-frequency non zero coefficient.
- If the magnitude of this coefficient is larger than a pre-defined threshold, move up to the next VLC table.

In this way, the choice of level is matched to the magnitude of the recently-encoded coefficients. The thresholds are listed in Table.1.8; the first threshold is zero

Table 1.8: Thresholds for determining whether to increment Level table number

Current VLC table	VLC0	VLC1	VLC2	VLC3	VLC4	VLC5	VLC6
Threshold to increment table	0	3	6	12	24	48	N/A (highest table)

which means that the table is always incremented after the first coefficient level has been encoded.

1.6.4.4 Encode the total number of zeros before the last coefficient

TotalZeros is the sum of all zeros preceding the highest non-zero coefficient in the reordered array. This is coded with a VLC. The reason for sending a separate VLC to indicate TotalZeros is that many blocks contain a number of non-zero coefficients at the start of the array and (as will be seen later) this approach means that zero-runs at the start of the array need not be encoded.

1.6.4.5 Encode each run of zeros

The number of zeros preceding each non-zero coefficient (run_before) is encoded in reverse order. A run_before parameter is encoded for each non-zero coefficient, starting with the highest frequency, with two exceptions:

- If there are no more zeros left to encode (i.e. run_before = TotalZeros), it is not necessary to encode any more run_before values.
- It is not necessary to encode run_before for the final (lowest frequency) non-zero coefficient.

The VLC for each run of zeros is chosen depending on (a) the number of zeros that have not yet been encoded (ZerosLeft) and (b) run_before. For example, if there

are only 2 zeros left to encode, run_before can only take 3 values (0,1 or 2) and so the VLC need not be more than 2 bits long; if there are 6 zeros still to encode then run_before can take 7 values (0 to 6) and the VLC table needs to be correspondingly larger.

Chapter 2

Fast Algorithms and Data Reuse

2.1 Frame-level Data Re-use & Mode Decision Strategy

The newest video compression standard H.264/AVC exhibits excellent compression ratio due to many brand new features compared with its previous counterparts. Among these features, multiple reference inter-prediction and variable block-size play very important roles. Besides the high compression ratio, high memory bandwidth is required for its real-time implementation. In order to reduce the memory bandwidth, this thesis presents a partially forward processing algorithm (PFPA) for frame-level data re-use and a mode decision strategy. Simulation results show that up to 2/3 of the original memory access bandwidth can be saved if using 5 reference frames and a search range of $[-64, 64]$. Also,the new mode decision method avoid sharply increased memory size used to store intermediate processing results and at the same time, the PSNR for decoded sequence is very close to the optimal result produced by the reference software JM9.0 provided by Joint Video Team (JVT) [14].

Established by Joint Video Team (JVT) and Moving Pictures Expert Group (MPEG), H.264/AVC has great advantage of coding efficiency compared with the suc-

cessful prior coding standards. It can save 64.46%, 48.80% and 38.62% bit-rate compared with that of MPEG-2, H.263++, and MPEG-4 [13]. However, high computational complexity and required bandwidth become two main obstacles for its real-time implementation. By using well-designed parallel processing architecture, the heavy computational load can be shared by several low-end processors. Furthermore, if the data-dependencies are carefully considered, these parallel processing architecture will not bring any visual quality degradation in the final encoded result [57]. But even in the parallel encoders, frequent data transmission between processors and external memories is still inevitable. Here, We will put our concentration on the system bus bandwidth reduction during the encoding. Section 2.1.1 will give an overview of the inter-prediction in H.264/AVC which is the most memory consuming part during the encoding, Section 2.1.2 will introduce our frame-level data re-use algorithm PFPA and Section 2.1.3 will introduce the mode decision strategy for PFPA, Section 2.1.4 is the supporting simulation results.

2.1.1 Inter-Prediction in H.264/AVC

Inter prediction creates a prediction model from one or more previously encoded video frames. The model is formed by shifting samples in the reference frame(s). Important differences from earlier standards primarily include the support for multiple reference frames and a range of different block sizes (down to 4×4).

H.264/AVC supports motion compensation block sizes ranging from 16×16 to 4×4 luminance samples with many options beteen the two. The luminance component of each macroblock (MB) (16×16 samples) may be split up in 4 ways as shown in Fig. 2.1(a): 16×16, 16×8, 8×16 or 8×8. Each of the sub-divided regions is a MB partition. If the 8×8 mode is chosen, each of the four 8×8 MB partitions within the MB may be

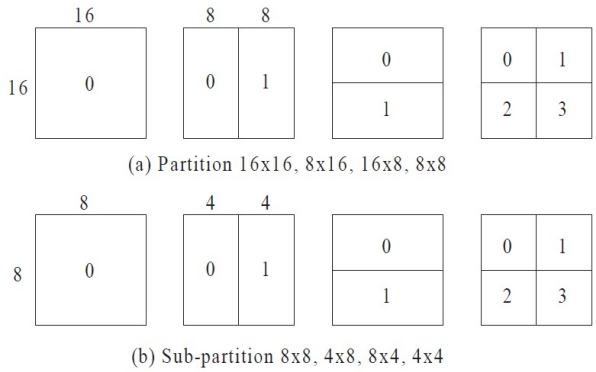

Figure 2.1: Variable block sizes in H.264/AVC

split in a further 4 ways as show in Fig.2.1(b).These partitions and sub-partitions give rise to a large number of possible combinations within each MB.

In H.264/AVC, the encoder can search in several different reference frames for the best match of every sub-partition in a MB. This makes it possible for every sub-partition with the size greater than 8×8 can has its own reference frame. It is very effective for uncovered backgrounds, repetitive motions, highly textured areas, etc [33]. But the local memory size are almost linearly increased with the number of reference frame because the search window (SW) and current MB are usually buffered in local memories to reduce data access from external memories.

2.1.2 Partially forward processing algorithm (PFPA)

Before the processor begins to encode a MB, all the required reference information needs to be loaded from the external memory to local buffer. This brings heavy burden for the data bus connecting local buffer in the processor and the external memory which stores corresponding reference frames. According to the JVT draft for H.264 [14], up to 5 reference frames can be used for a H.264/AVC encoder. The motion estimation

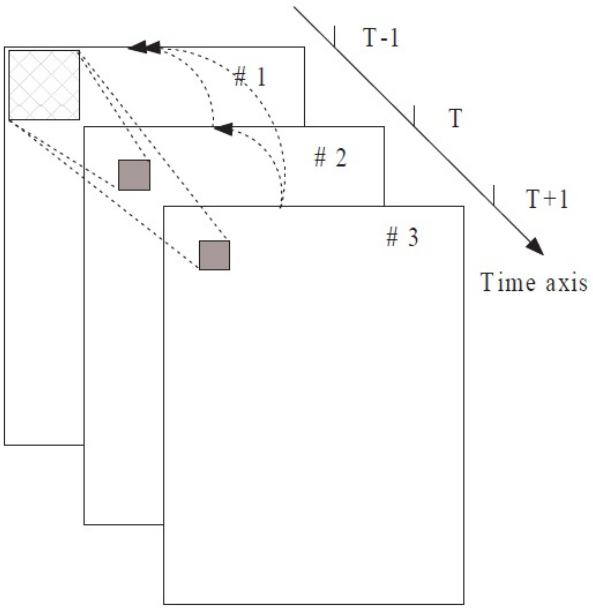

Figure 2.2: Data dependency for multiple reference prediction

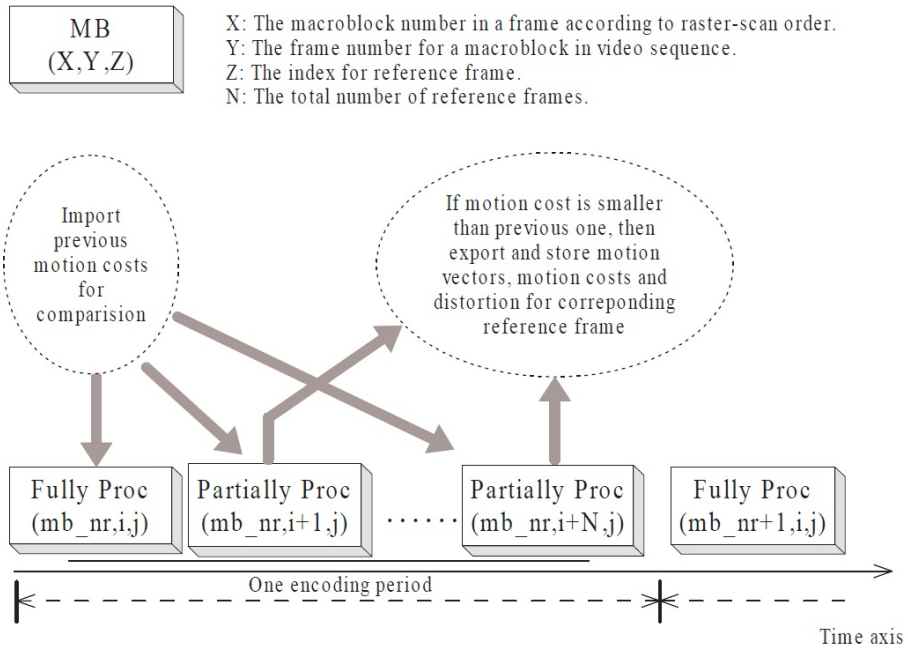

Figure 2.3: Timing diagram for an encoding period

core searches all the possible positions for all inter-prediction modes in the specified range in all these reference frames. Plus the quarter-pixel motion search, it becomes the most power and memory consuming part during encoding process. This explains why the reference software, JM7.3, requires computing power of 3.6 tera-operations/s and memory access of 5.6 tera-bytes/s on a general processor to encode HDTV720P videos [18].

Fig.2.2 shows the data dependency for inter-prediction using multiple reference frames. #1, #2 and #3 mark the processing order of three frame. Frame #2 is the current frame, frame #1 serves as the reference frame for current frame and frame #3 is the next frame in time axis. Obviously, frame #1 is also the reference frame for frame #3. Intuitively, this can be a clue for us to reuse this reference data.

In multiple reference inter-prediction, the data in one SW is useful for several co-located MBs in different frames. Here, our proposed PFPA load these SW only once to reduce unnecessary system bandwidth utilization. Fig.2.3 shows the timing diagram of encoding a MB. Actually, in such a MB period (encoding time of a MB) of our algorithm, encoder also processed other co-located MBs in following frames partially. This means, the life-time of a SW is just one MB encoding period. In this period, all MB which need this SW will be processed or partially processed. Among all these co-located MBs in Fig.2.3, only the MB with the smallest frame index are fully encoded, others just passed the ME+Q+DCT stages. Fig.2.4 shows the system bus bandwidth saving ratio with the increase of search range. According to this figure, the bandwidth using PFPA becomes only about one third of the original one in most cases.

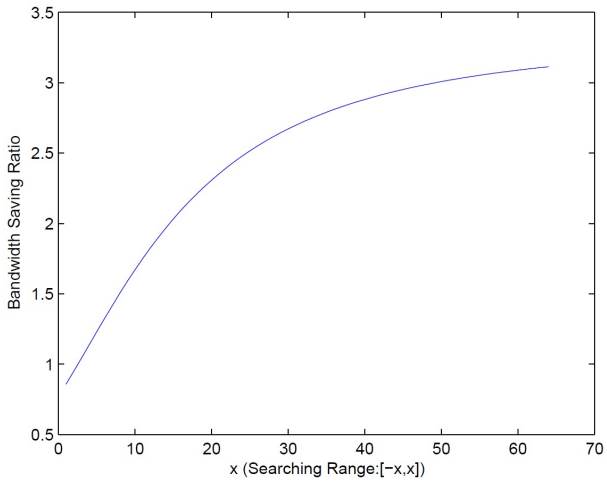

Figure 2.4: Saving ratio

2.1.3 Mode decision for PFPA

Different from traditional processing order, we start the motion search from the reference frame with the largest index in the list. However, in PFPA, not all the reference frames are at the reference buffer at the same time. Thus, we cannot decide which reference frame is the best during one encoding period. So, we can only compare the cost with its previous one. If smaller, the intermediate encoding result, the motion cost & MVs will overwrite its previous counterparts. Once we finished searching all these reference frames, we may decided which mode & refernce are the optimal.

Changing the search order of these reference frames also post two challenges. First, the accurate predicted motion vector (PMV) defined in [14] is unavailable if the MB is doing partial processing since its neighboring MVs have not been finally determined. Second, every time a MB is partially processed using one reference frame, too much intermediate information needs to be stored for future use, because we never know what kind of mode&ref_index combination will be the best. A simple example is shown in Fig.2.5, using different ref_index may make 8×8 become the best prediction mode in

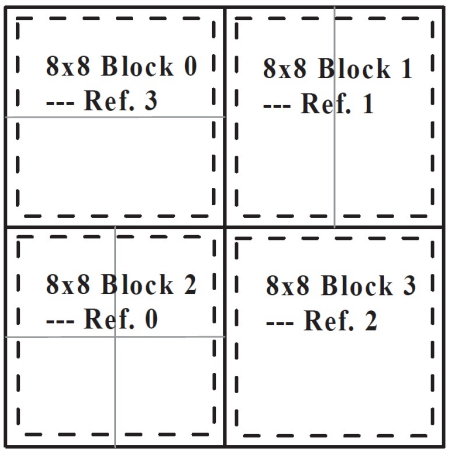

Figure 2.5: Multiple reference for 8x8 mode

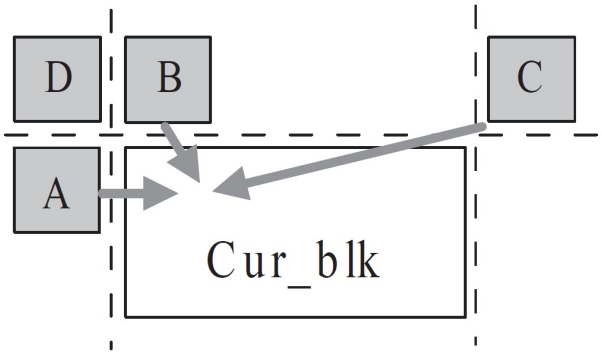

Figure 2.6: Predicted Motion Vector determination

the final. Intuitively, we have to keep these related information for all the modes. But increased memory size makes it impractical. Thus, a trade-off must be made between the efficiency and the final encoding quality.

In order to solve the problems above, we provided two schemes and made a comparison between them. Fig.2.6 and Fig.2.7 shows how PMV is produced according to the standard. In Fig.2.7, the meaning of "available" is three folded: 1. physically, this block exists; 2. this block has been fully encoded before current block; 3. the optimal ref_index of this block is the same as current reference frame. Here,"cur_blk"

```
If (Block_A, Block_B and Block_C are all available)
    PMV = median (MV_A, MV_B, MV_C);
else if (only Block_X is available)
    PMV = MV_X, // Here, X = A, B or C
else if (only Block_C is available)
    Block_C = Block_D;
else  // Block_A is unavailable
    MV_A = 0;
    PMV = median (MV_A, MV_B, MV_C);
```

Figure 2.7: Pseudo-code for PMV computation

can be a sub-block inside a MB with any size. Obviously, for PFPA, condition no.2 and no.3 are usually unsatisfied. So, in PFPA, we used our own definition of "available": 1.the same as above; 2.this block has partially encoded before current block using the same reference frame. By this definition, "MV_A" to "MV_D" are all motion vectors based on current reference frame. This is why these motion vectors should be kept as intermediate motion information. So, the PMVs in partial encoding stage are just sub-optimal. However, in the full encoding stage of a MB (see Fig.2.3), the motion vector difference (MVD) should be computed by its neighboring and current optimal MVs.

Our two proposed schemes both keep only one best mode and related MVs, partial encoding results and motion costs at any time. These motion costs are compared in partial encoding or full encoding. If the current motion cost is smaller than previous reference frame, then the corresponding motion information and distortion will overwrite previous one.

Table.2.1 shows some statistical data for different video test sequences using JM9.0. These sequences contains many kinds of scenes, from relatively still scene (like "Akiyo") to highly motive scene (like "Mobile calendar"). The second column is the percentage of MBs which only use the previous neighbored reference (ref_index=1). The third column shows the percentage of MBs which are benefit from multiple references (ref_index>1). The fourth to seventh column are the percentages for 4 main block sizes

Sequences	% of MBs, ref_idx=1	% of MBs, ref_idx>1	16×16	16×8	8×16	8×8
Akiyo	90.3%	9.7%	19.0%	20.2%	25.7%	35.1%
News	89.4%	10.6%	4.8%	15.0%	18.6%	53.6%
Mother and Daughter	86.9%	13.1%	32.3%	22.9%	24.7%	20.1%
Silent	84.9%	15.1%	12.1%	12.4%	15.9%	59.6%
Hall Monitor	83.2%	16.8%	24.5%	30.0%	16.3%	29.2%
Foreman	69.3%	30.7%	28.0%	17.0%	21.3%	33.7%
Coastguard	48.1%	51.9%	11.5%	5.7%	4.6%	78.2%
Stefan	45.2%	54.8%	12.1%	14.1%	7.8%	66.0%
Mobile Calendar	20.9%	79.1%	8.4%	10.7%	6.9%	74.0%

Table 2.1: Statistical data for H.264/AVC inter-prediction modes

if their used a reference frame with ref_index>1.

From this table, we found that about 80% of the macroblocks only use their neighbored reference frames (ref_index = 1). Also, we found that 8×8 occupies almost 50% among these 4 primary modes if ref_index>1. Especially, for large-motion video sequences, such like "Coastguard","Stefan" and "Mobile Calendar", 8×8 actually is the dominant mode when ref_index>1. This fact inspired us with such an idea: if we only use 8×8 mode when ref_index>1, it should not bring a big degradation for the encoding quality. This is verified to be right when we compared our mode decision scheme 1 and scheme 2 in the following sections.

Scheme 1 assume that the encoder can use any mode for any ref_index, but in each mode, all sub-blocks should share the same reference frame. So, after each encoding period, the previous stored motion information and distortion are all replaced (if current motion cost if smaller), or kept unchanged.

In scheme 2, if ref_index is greater than 1, only 8×8 mode is allowed. However, in this scheme the reference frames for every 8×8 blocks are unnecessary to be the same.

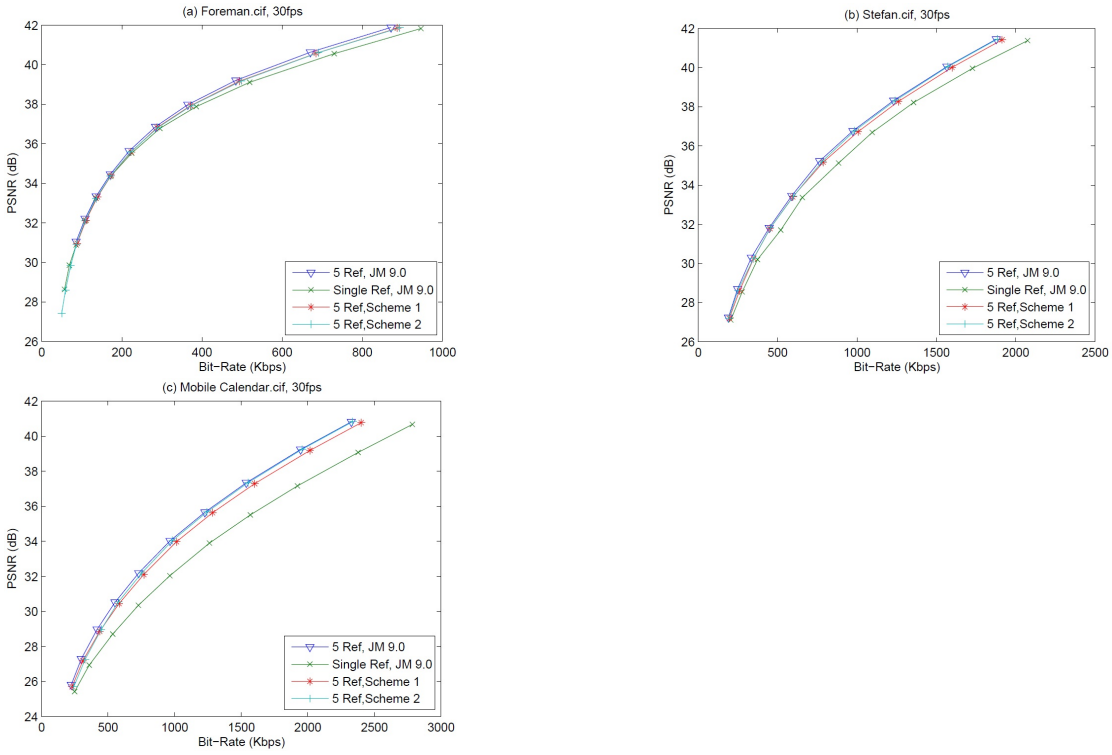

Figure 2.8: R-D Curves for different test sequences

This means, the MVs, motion costs and distortions of these 4 8×8 blocks are stored and compared respectively. If ref_index $= 1$, all prediction modes are allowed, because most of the large size prediction mode can find the best match in the first reference frame.

2.1.4 Simulation Results

In this section, several Rate-distortion (R-D) curves are provided, including "Foreman.cif", "Stefan.cif" and "Mobile Calendar.cif" with 4 kinds of encoding schemes: JM9.0 using 5 reference frames, JM9.0 using single reference frame, Scheme 1 and Scheme 2. The simulation is based on H.264/AVC baseline encoder, search range [-64,64], using context adaptive variable length coding (CAVLC). Fig.2.8 compares the rate-distortion curves among these 4 encoding methods.

It is shown that for the slight-motion sequence, the coding qualities for scheme 1

and scheme 2 are very close, such like Fig.2.8(a). But for the large-motion sequence, such like Fig.2.8(b)&(c),the encoding quality of scheme 2 will be much better than scheme 1 for about 0.2-0.3 dB. Furthermore, for all kinds of video sequences, the encoding quality of scheme 2 actually is very closed to JM9.0 optimal coding.

2.2 FME Mode Reduction

Motion estimation eliminates temporal redundancy between adjacent frames for better coding efficiency. In order to support variable block size and quarter-pixel precision, H.264/AVC motion estimation needs large amount of computation and accounts for $60\% \sim 90\%$ of encoding time.

Motion estimation greedily refines the best candidate hierarchically in two phases: Integer Motion Estimation (IME) and Fractional Motion Estimation (FME). IME finds the integer motion vectors (IMV) for each of 41 variable-size blocks according to both minimum sum of absolute difference (SAD) and MV-bit-rate estimation. FME refines those 41 IMVs based on both sum of absolute transformed difference (SATD) in quarter-pixel precision and MV-bit-rate estimation. This cascaded flow limits the FME search area to around the IMV instead of the whole search window in H.264/AVC [10].

Currently, many VLSI architectures are based on this two-stage data flow, e.g., [10], [45], [31], [30], [54] and [37]. These architectures obtain coarse IMVs in the IME stage and make mode decisions in FME stage after all the 7 modes have been iterated with quarter-pixel precision. Some of them use hardware specific algorithms to expedite the processing speed and save hardware resources, e.g., data reuse, hardware time division multiplexer (TDM) and pipelining. However, even with these optimizations, exercising all possible modes in FME is still a bottleneck for both memory throughput

and processing capability.

In this thesis, we present a statistical analysis of the key motion estimation factors, including relationship between SAD and SATD in MB level and changes of SATD from full-pixel precision to quarter-pixel precision in motion estimation flow, including IME and FME. We show that in some circumstances, the final mode decision can be made in the IME stage and fractional motion search only needs to be done for a single mode in FME.

Reduction of motion estimation computations can be achieved using fast motion estimation and early termination algorithms, e.g., [49] and [50]. These algorithms aim at reducing motion searching points in both IME and FME stage, while our proposed scheme can be complementary to these algorithms and further speed up motion estimation process by avoiding unnecessary modes in FME.

2.2.1 Motin Estimation in H.264/AVC

2.2.1.1 Variable Block-size Motion Search

In H.264/AVC [14], four INTER modes are supported: 16×16, 8×16, 16×8 and $P8 \times 8$. If $P8 \times 8$ mode is selected, each 8×8 sub-block can select one of the following sub-modes: 8×8, 4×8, 8×4 and 4×4. Also, depending on the encoding algorithm (e.g., JM9.0 in [1]), if the 16×16 mode is selected and the corresponding coding results satisfy certain conditions, the MB may be skipped for encoding, which is known as SKIP mode.

Each block/subblock has its own motion vectors. The differences between these motion vectors and their corresponding predicted motion vectors (PMVs) will be encoded.

2.2.1.2 Integer Motion Search

Before the motion search of every block/subblock starts, PMVs will be provided. The standard [14] specified how PMVs are generated from neighboring blocks/subblocks around current block/subblock. According to the [14], PMVs are pointing to the search center of current block/subblock.

The criterion to calculate the best motion vectors in IME is based on motion costs which consists of SAD cost, the costs of MV-bit-rate estimation and reference information. MVs with the minimal motion cost will be selected and stored for further refinement in FME stage.

Usually, the number of reference frames is fixed for a video sequence. So, in most cases, it is not included in the cost equation.

Let's define X as the pixels from current MB and X' as the predicted pixels from the reference picture. $S = X - X'$ denotes the difference between these two. Then, MB level SAD is defined as (2.1) .

$$SAD = \sum_{i,j} |S_{i,j}|, \;\; i = 0 \sim 15 \text{ and } j = 0 \sim 15 \tag{2.1}$$

The overall cost of a MB mode is based on (2.2)

$$J = SAD + \lambda_{motion} \times MV\_BITS \tag{2.2}$$

Here, MV_BITS stands for the number of bits which are needed to encode the difference of current MVs and PMVs. λ_{motion} is a quantization parameter (QP) dependent constant which is determined by (2.3) [19].

$$\lambda_{motion} = \sqrt{0.85 \times 2^{(QP-12)/3}} \tag{2.3}$$

2.2.1.3 Fractional Motion Search

FME is conducted right after IME and its search center is pointed by the optimal MVs found by IME. Half-pixels around optimal integer positions will be searched and then quarter-pixels around optimal half-pixel positions.

The standard [14] specifies that half-pixels are interpolated using a 6-Tap FIR from neighboring integer pixels and quarter-pixels are interpolated using a bilinear filter from two neighboring half-pixels.

Different from IME, the FME uses SATD instead of SAD to represent prediction errors. So, instead of (2.2), (2.4) is used to calculate overall prediction cost in FME. The prediction mode and MVs with minimal cost will be selected as the final mode and final MVs.

$$J = SATD + \lambda_{motion} \times MV\_BITS \tag{2.4}$$

$$SATD_{4\times4} = (\sum_{i,j} |DiffT(i,j)|)/2, \;\; 0 \leq i,j \leq 3 \tag{2.5}$$

Here, $SATD$ stands for MB level $SATD$ which is the sum of all the $SATD_{4\times4}$ defined in (2.5) in that MB. In (2.5), $DiffT$ denotes the Hadamard transformed differences of original pixels and motion compensated pixels.

2.2.2 Relationship between SAD and SATD

To simplify the analysis, the INTER prediction modes we are discussing here include mode 16×16, 8×16, 16×8 and submode 8×8 in $P8 \times 8$ and the simulation is based on H.264/AVC baseline. But the analysis conclusion can be extended to any submodes under $P8 \times 8$ (hereafter SAD or SATD refers to MB level SAD or SATD) and

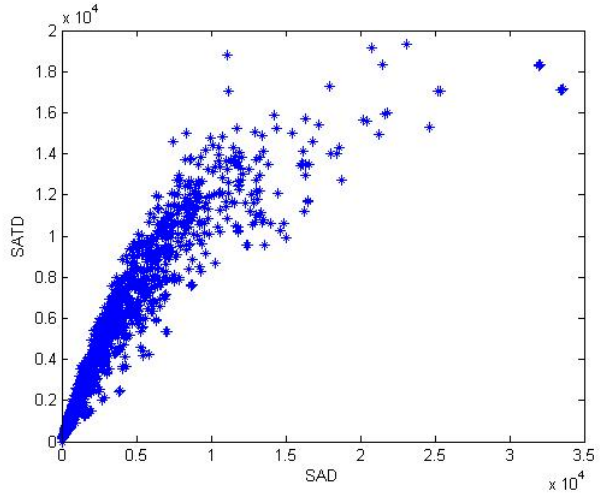

Figure 2.9: MB Level SAD vs. SATD

other H.264/AVC profiles. Since in JM9.0 (without RDOPT), MB skip mode has the same block size as the 16×16 mode and cannot be set as the best mode until residual coding has been finished, we consider it as a special case of the 16×16 mode in this paper.

Conventional belief is that SATD is not dependent on SAD and they do not have fixed relationships.

Fig.2.9 shows the relationships between MB Level SAD and SATD when we encode "Ice.CIF". This figure seems to indicate that the ratio between SATD and SAD is random. From the definition of SATD and SAD, SATD is a frequency-domain based variable and SAD is spatial-domain based variable, it is true that there is no fixed relationships between them, especially for different MBs with different video contents. Thus, it becomes very hard to predict SATD with SAD.

However, if we think about different INTER prediction modes for the same MB, intuitively, there should be some kind of similarities because the motion compensated MBs in the same location but with different modes should have similar textures and

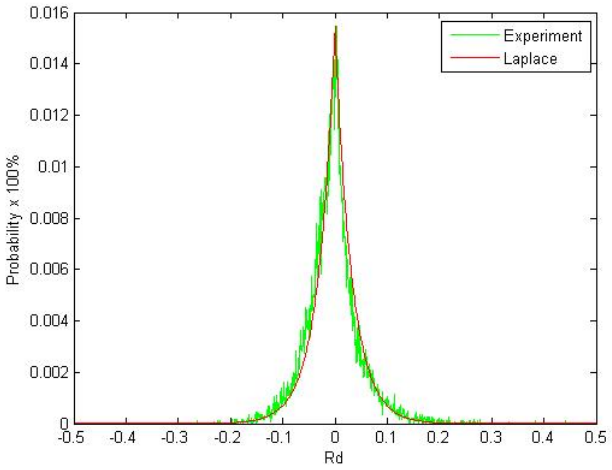

Figure 2.10: Statistics of Rd ("Mobile.SIF"/40 Frms/QP=24)

most likely, the trend for pixel changes will be very close.

Let's define $SATD_b$ and SAD_b as the MB level SATD and MB level SAD for the best full pixel mode in a MB. Also, we define $SATD_i$ and SAD_i as the MB level SATD and MB level SAD for full-pixel modes other than the best in a MB. Then R_b and R_i are defined as (2.6) and (2.7). Here, $i = 1, 2, 3, P8 \times 8$ and $i \neq b$.

$$R_b = SATD_b/SAD_b \tag{2.6}$$

$$R_i = SATD_i/SAD_i \tag{2.7}$$

Also, we define R_d as the difference between these two ratios over R_b. Note that in this paper, R_b, R_i, R_d are all for the same MB.

$$R_d = (R_i - R_b)/R_b \tag{2.8}$$

Fig.2.10 shows a typical statistical curve for R_d. More experimental data can

be found in Fig.2.11. The names of the video files used in this paper are from [2]. These two figures indicate that the ratios for SATD vs. SAD in the same MB have very high correlations. We also found their characteristics can be closely fitted by a Laplace distribution. Its probability density function (pdf) is

$$f(R_d) = \frac{\alpha}{2 \cdot \lambda} \cdot \exp\left(-\frac{\alpha \cdot |R_d - \mu|}{\lambda}\right), -\infty < R_d < +\infty \tag{2.9}$$

Here, $\alpha > 0, \lambda > 0$ (λ is a factor in the Laplace distribution which is not the same as λ_{motion} above). α is a scale factor and μ is the mean of R_d which is approximately zero mean in most cases. Eq.(2.9) is a continuous pdf function and experimental data in Fig.2.10 are probabilities for discrete samples. We define the interval between every two samples as $1/\alpha$ and multiplied the pdf in eq.(2.9) with this interval to get the Laplace curve in Fig.2.10, so that the two curves have the same meaning (which is probability, instead of probability density) and are comparable to each other.

Based on this fact, we have the cumulative distribution function (cdf) $F_{R_d}(R_d)$ as:

$$F_{R_d}(R_d) = \begin{cases} 1 - \frac{1}{2} \cdot \exp\left(-\frac{\alpha \cdot R_d}{\lambda}\right), & R_d \geq 0 \\ \frac{1}{2} \cdot \exp\left(\frac{\alpha \cdot R_d}{\lambda}\right), & R_d < 0 \end{cases} \tag{2.10}$$

In order to find the condition under which a minimum SAD also maps to a minimum SATD for the same MB, we define R_{SAD} as the ratio between the full-pixel SAD of non-best mode and the best full-pixel mode, and R_{SATDf} as the ratio between the full-pixel SATD of non-best mode and the full-pixel SATD of the best mode:

$$R_{SAD} = SAD_i / SAD_b \tag{2.11}$$

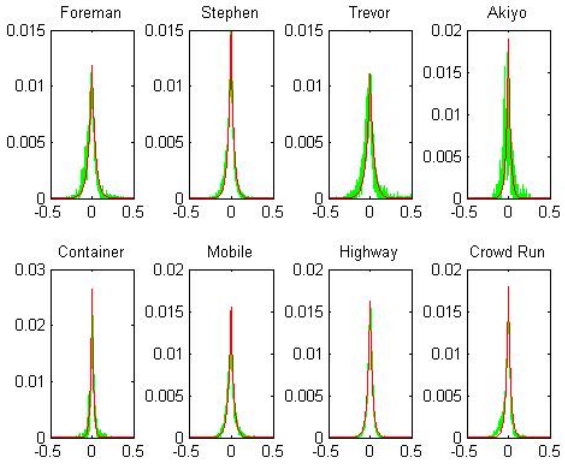

Figure 2.11: Statistics of Rd and the corresponding Laplace Distribution

$$R_{SATDf} = SATD_i / SATD_b \tag{2.12}$$

Here, $i = 1, 2, 3, P8 \times 8$ and $i \neq b$. According to (2.6) and (2.7):

$$\begin{aligned} SATD_i \quad - \quad & SATD_b = R_i \cdot SAD_i - R_b \cdot SAD_b && (2.13) \\ = \quad & R_i \cdot SAD_b \cdot R_{SAD} - R_b \cdot SAD_b && (2.14) \\ = \quad & [(R_d + 1) \cdot R_b \cdot R_{SAD} - R_b] \cdot SAD_b && (2.15) \\ = \quad & [(R_d + 1) \cdot R_{SAD} - 1] \cdot R_b \cdot SAD_b && (2.16) \end{aligned}$$

Since $R_b \cdot SAD_b = SATD_b$, we have:

$$R_{SATDf} = (R_d + 1) \cdot R_{SAD} \tag{2.17}$$

If R_{SATDf} is given, R_{SAD} and R_d have a one-to-one mapping relationship, and the probability that R_{SAD} is less than a threshold is equal to the probability that R_d is greater than another threshold. Therefore, for a given R_{SATDf}, $F_{R_{SAD}}(R_{SAD})$, the cdf

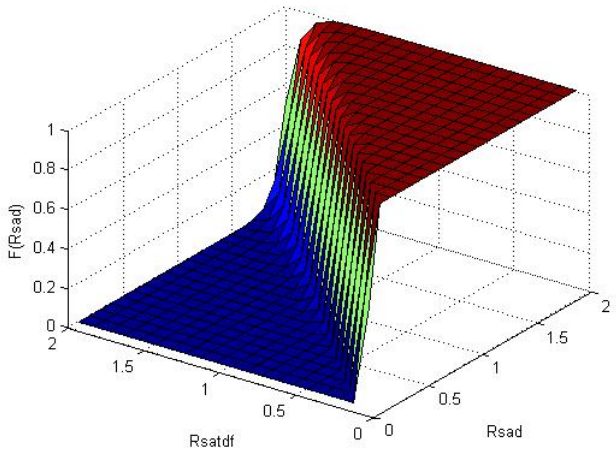

Figure 2.12: CDF of R_{SAD} ($\alpha = 1000, \lambda = 41.67$)

of R_{SAD} is:

$$F_{R_{SAD}}(R_{SAD}) = 1 - F_{R_d}\left(\frac{R_{SATDf}}{R_{SAD}} - 1\right) \tag{2.18}$$

$$= \begin{cases} \frac{1}{2} \cdot \exp\left(-\frac{\alpha \cdot (R_{SATDf}/R_{SAD} - 1)}{\lambda}\right), \\ \text{if } R_{SATDf} \geq R_{SAD} \\ 1 - \frac{1}{2} \cdot \exp\left(\frac{\alpha \cdot (R_{SATDf}/R_{SAD} - 1)}{\lambda}\right), \\ \text{if } R_{SATDf} < R_{SAD} \end{cases} \tag{2.19}$$

Both R_{SATDf} and R_{SAD} are greater than zero. Since α is large ($\alpha = 1000$ in our model), when $R_{SAD} \gg R_{SATDf}$, $F_{R_{SAD}} \approx \left(1 - \frac{1}{2} \cdot \exp\left(\frac{-\alpha}{\lambda}\right)\right)$ approaches 1. Fig.2.12 shows the cdf of R_{SAD}. Therefore, given an expected probability $F_{R_{SAD}}$, once we have R_{SATDf} determined, we can quickly decide what will be the most probable minimum value for R_{SAD}. This conclusion will be used in the next section.

2.2.3 Relationship between full-pixel SATD and quarter-pixel SATD

At the second stage of motion estimation in H.264/AVC, FME needs to do both the half-pixel and quarter-pixel interpolation, and also the sub-pixel motion costs refinement. To study how much motion estimation cost (mcost) the refinement process can save, lets define $SATD_{q,b}$ as quarter-pixel SATD for the best mode from IME, $SATD_{f,b}$ as the full-pixel SATD for the best mode from IME, $SATD_{q,i}$ and $SATD_{f,i}$ as the quarter-pixel and full-pixel SATDs respectively for any modes other than the best mode. Now, two ratios about the SATDs are defined as following:

$$\delta_b = \frac{SATD_{q,b} - SATD_{f,b}}{SATD_{f,b}} \tag{2.20}$$

$$\delta_i = \frac{SATD_{q,i} - SATD_{f,i}}{SATD_{f,i}} \tag{2.21}$$

δ_d defines the differences between these two modes.

$$\delta_d = \delta_i - \delta_b \tag{2.22}$$

Fig.2.13 shows typical examples of the probabilities of δ_d we obtained from experiments. It indicates that during FME, all the modes are prone to have the same or close refinement ratio, and the best mode from IME has a better chance to produce more improvements in FME.

The probabilities of δ_d are not very regular, but their cdfs are very close in $(-\infty, 0)$, as shown in Fig.2.14.

Fig.2.14 shows $F_{\delta_d}(\delta_d)$, the cdf of δ_d. These curves are created using the data shown in Fig.2.13. In Fig.2.14, there are some big variations when $\delta_d > 0$. However,

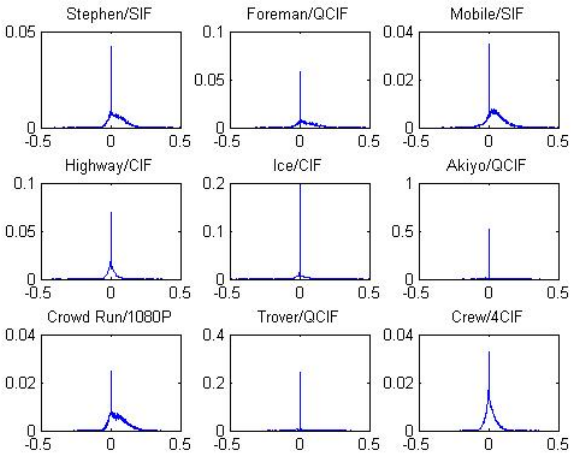

Figure 2.13: Probabilities of δ_d

what we really care is the interval when $\delta_d \leq 0$. This is further discussed below.

If the best mode in IME stage is still the best mode after FME, it must satisfy:

$$\frac{SATD_{q,b}}{SATD_{q,i}} = \frac{(\delta_b + 1) \cdot SATD_{f,b}}{(\delta_i + 1) \cdot SATD_{f,i}} \tag{2.23}$$

$$= \frac{(\delta_b + 1)}{(\delta_i + 1) \cdot R_{SATDf}} \tag{2.24}$$

$$= \frac{(\delta_b + 1)}{(\delta_b + \delta_d + 1) \cdot R_{SATDf}} \leq 1 \tag{2.25}$$

$$R_{SATDf} \geq \frac{\delta_b + 1}{\delta_d + \delta_b + 1} = \frac{C}{\delta_d + C} \tag{2.26}$$

Here, $C = \delta_b + 1$. So, for a certain value of R_{SATDf}, the probability to satisfy (2.25) is:

$$F_{R_{SATDf}}(R_{SATDf}) = 1 - F_{\delta_d}\left(\frac{C}{R_{SATDf}} - C\right) \tag{2.27}$$

Since $\delta_b = (SATD_{q,b} - SATD_{f,b})/SATD_{f,b}$ and $SATD_{q,b}$ is the quarter-pixel SATD for the best mode from IME while $SATD_{f,b}$ is the full-pixel SATD for the best

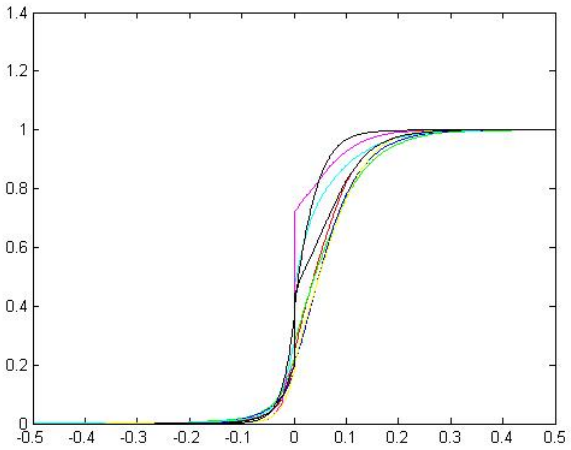

Figure 2.14: CDF of δ_d

mode from IME, $SATD_{q,b}$ must be smaller than or equal to $SATD_{f,b}$, thus $\delta_b \leq 0$. As a result, $C \leq 1$ and very close to 1, it has very minor impact on $F_{R_{SATDf}}(R_{SATDf})$. Therefore, we may approximate $C = 1$ in the analysis. For $F_{R_{SATDf}}(R_{SATDf})$, the interval $(1, +\infty)$ is mapped to $(-1, 0)$ in $F_{\delta_d}(\delta_d)$. As we mentioned earlier, in this interval, $F_{\delta_d}(\delta_d)$ is almost the same for different video contents.

Based on (2.27), we plot $F_{R_{SATDf}}(R_{SATDf})$ in Fig.2.15. From Fig.2.15, it is easy to see that only if $R_{SATDf} > T_{R_{SATDf}}$, we will have a probability of $F_{R_{SATDf}}(T_{R_{SATDf}})$ to keep full-pel best mode after FME. In our experiment, we take $T_{R_{SATDf}} = 1.05$ and $F_{R_{SATDf}}(T_{R_{SATDf}}) = 95\%$. Then we substitute $R_{SATDf} = 1.05$ to (2.19), we will have a similar curve for the cdf of R_{SAD} in Fig.2.16

From this 1-D cdf of R_{SAD}, , it is easy to know whenever we have $R_{SAD} > 1.1$, we have a more than 90% probability to keep the best mode of full-pixel as the best mode after FME. This provides a criterion for removing any less probable modes in FME. A different R_{SAD} may be chosen if a higher or lower probability is desired.

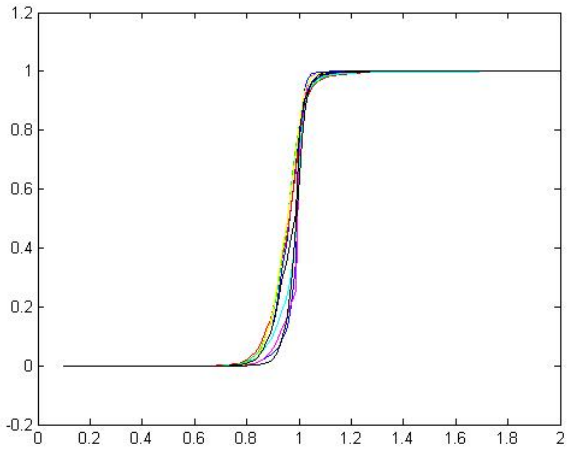

Figure 2.15: CDF of R_{SATDf}

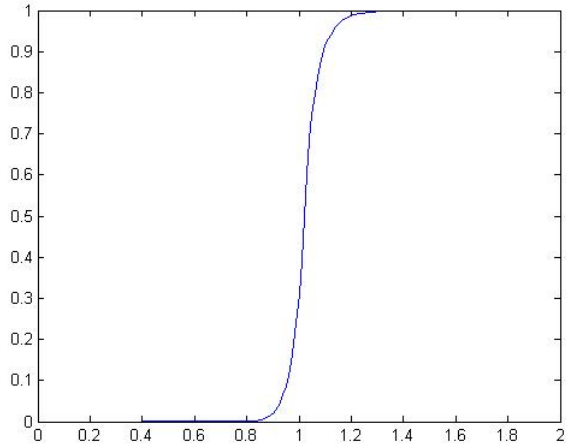

Figure 2.16: CDF of R_{SAD} ($R_{SATD} = 1.05$)

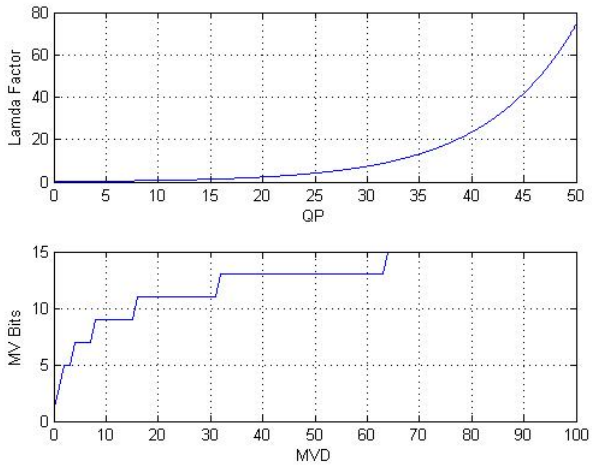

Figure 2.17: MV Impacts on Motion Cost

2.2.4 Impacts of Motion Vectors

Motion vector computations account for part of the cost for both IME and FME. As shown in (2.2) and (2.4), the second part of the motion cost is dependent on factor λ_{motion} and MV bits.

Fig.2.17 shows how λ_{motion} changes with QP and the relationship between motion vector differences (MVD) and the number of bits to encode them [1]. Eq.(2.28) is the formula for calculating MVD.

$$MVD_{x/y} = MV_{x/y} - PMV_{x/y} \tag{2.28}$$

In Fig.2.17, it is obvious that λ_{motion} increases quickly with large QP values. When QP is large (e.g., $QP > 40$), it puts a big weight on motion bits and sometimes MV costs can even dominate the cost J in (2.2) and (2.4). This paper is primarily targeted at high-end applications with a small QP (≤ 24), thus λ_{motion} will be a value smaller than 4.

The second part of Fig.2.17 shows the positive half of MVD, the negative part is symmetric to it. According to Fig.2.17, if MVD changes are within ±2 units, it only changes 2 bits at the most on MV bits. When MVD is located in certain intervals, its small change will not impact MV bits at all.

It is shown in [39] that since the sub-pels are generated from the interpolation of integer pixels, the correlation inside a fractional pixel search window is much higher than that inside an integer-pel search window. Hence, the matching error decreases monotonically as the search point moves closer to the global minimum and in most cases, the best match in FME will be within ±2 units distance from the best match in IME. Even in the worst case, during FME, the MVD changes will be within ±3 units.

Therefore, if λ_{motion} is small, variations of MV bits have very low probability to alternate the best modes.

2.2.5 Scheme for FME mode reduction and simulation results

For FME mode reduction, a very straight forward method is to use IME best mode as the final best mode and only do sub-pixel INTER prediction for a single mode in FME. Experimental data shows that the local image quality downgrade (Local PSNR) for these MBs with wrong best modes may be up to 2dBs as compared with the regular full FME flow. This local downgrade is visually noticeable in these MBs.

Fig.2.18 presents some of the experimental results of "IME Best Mode Miss-Rate" with different QPs and different video contents. Here, a "Miss" means the best IME mode was replaced by other mode in FME.

This figure also shows, even though for the same video content, the smaller the QP, the smaller the miss-rate is, we still have a high miss-rate when $QP \leq 24$. In some cases, even if the QP is very small, the miss-rate is still above 30% (e.g.,

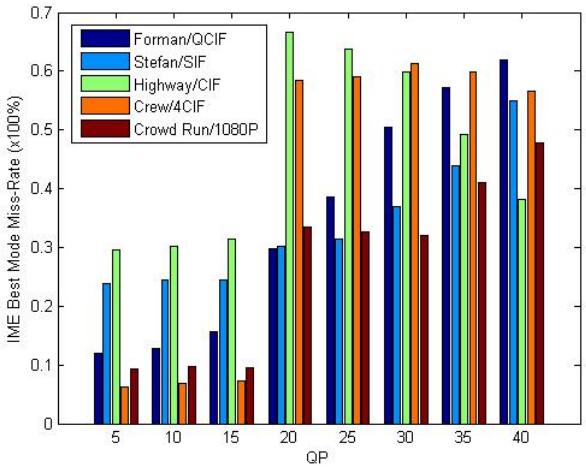

Figure 2.18: "IME Best Mode Miss-Rate" vs QP

"Highway/CIF").

The results in this paper are useful in two ways. First, it shows a scheme to achieve video quality equivalent to full FME best mode flow while significantly reducing motion estimation computation cost. Second, it shows which MBs will have video quality downgrades if the FME mode estimation is skipped and the IME best mode is always used as the final best mode.

Based on the observations in above sections, we know that if QP is not very large (e.g., $QP \leq 24$), and if the R_{SAD} satisfies a certain criterion, the probability is high that the best mode decided in IME will continue to be the best mode in FME. In this case, sub-pixel motion search is needed for only a single mode, i.e., the best IME mode.

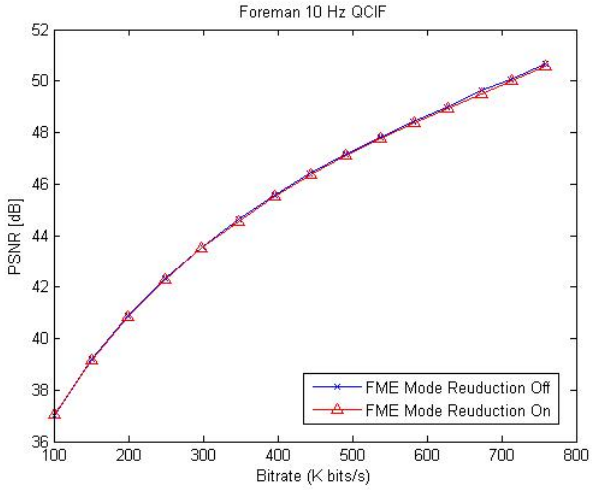

Figure 2.19: Luminance PSNR curve for "Foreman"

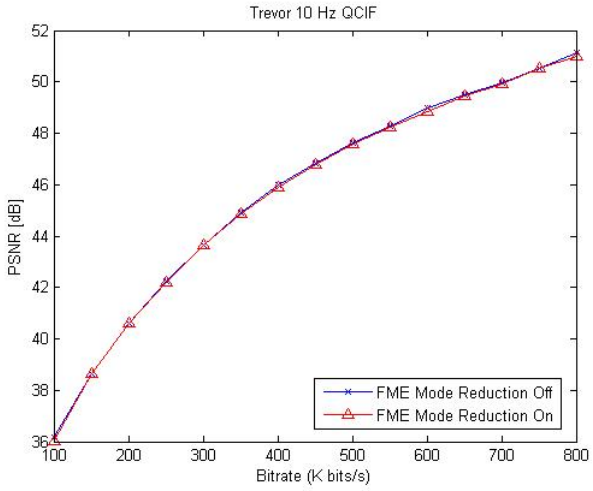

Figure 2.20: Luminance PSNR curve for "Trevor"

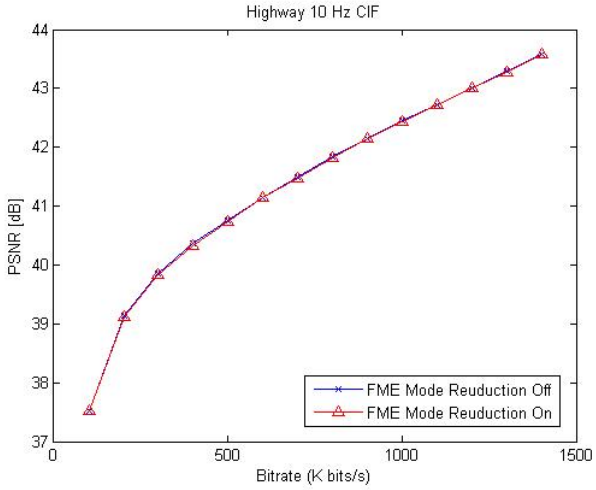

Figure 2.21: Luminance PSNR curve for "Highway"

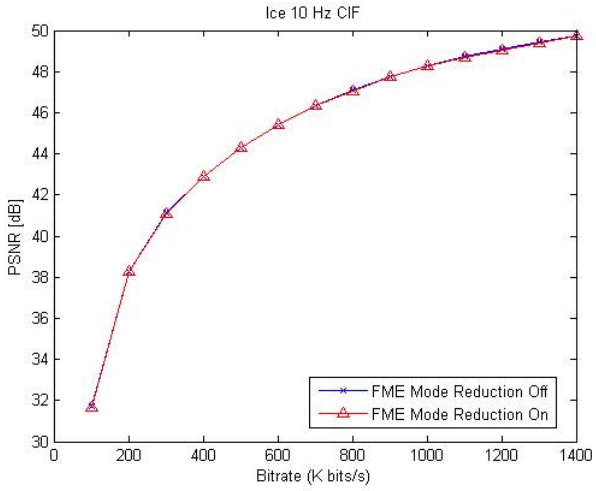

Figure 2.22: Luminance PSNR curve for "Ice"

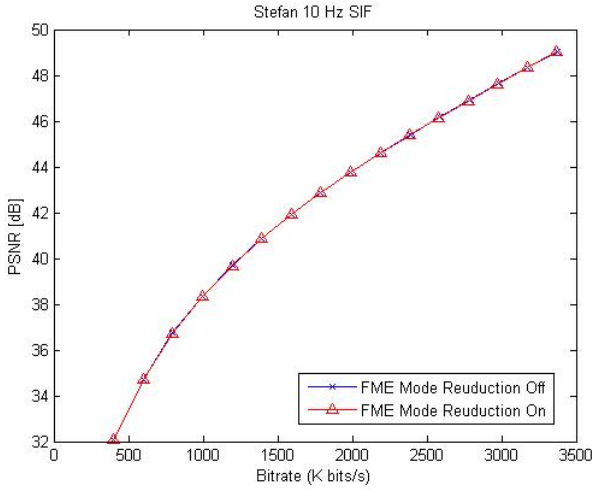

Figure 2.23: Luminance PSNR curve for "Stefan"

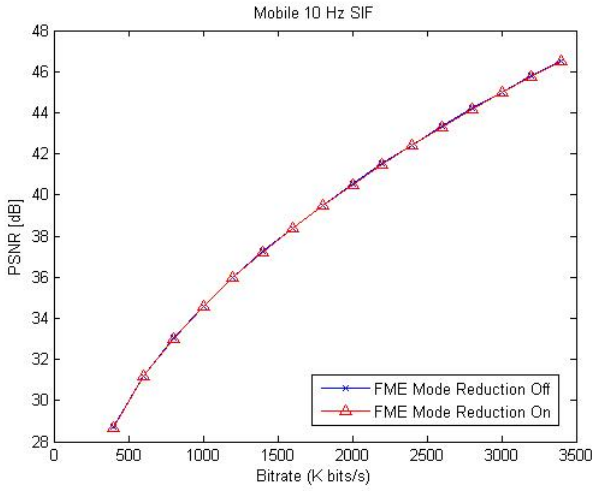

Figure 2.24: Luminance PSNR curve for "Mobile"

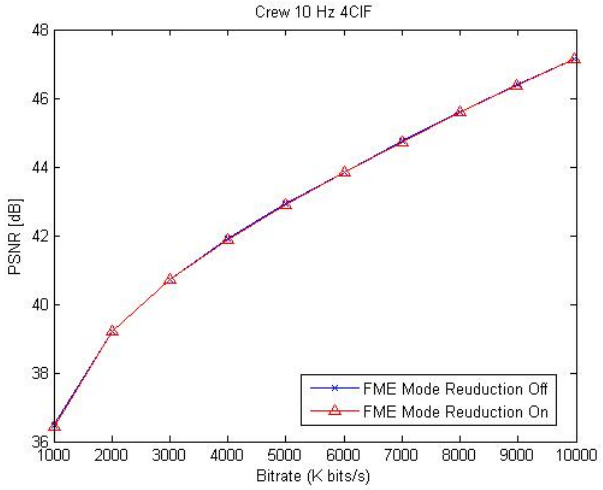

Figure 2.25: Luminance PSNR curve for "Crew"

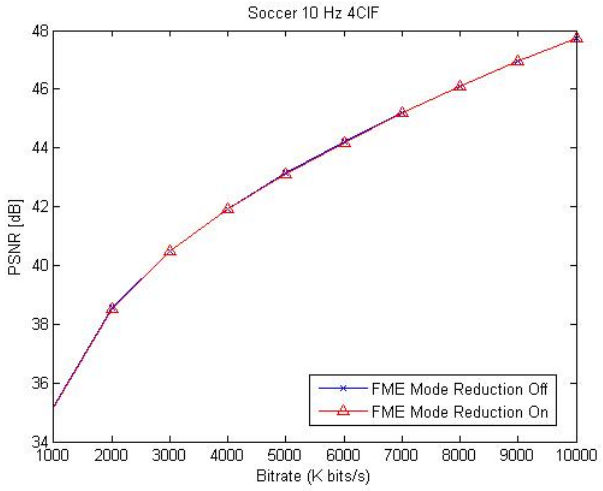

Figure 2.26: Luminance PSNR curve for "Soccer"

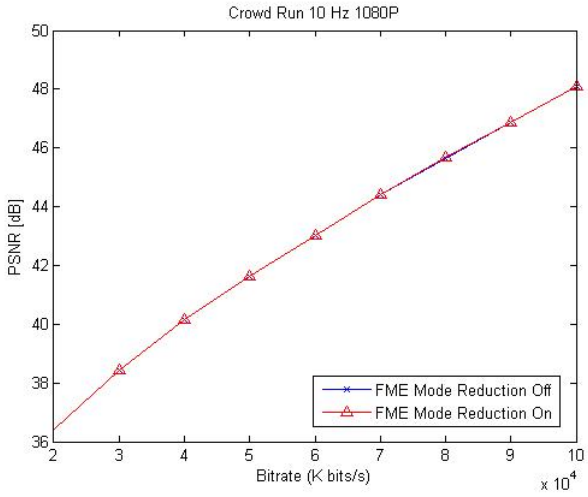

Figure 2.27: Luminance PSNR curve for "Crowd Run"

This leads to a simple scheme for FME mode reduction:

1. In IME stage, calculate the costs for all the modes, from 16×16 to $P8 \times 8$, using the criteria in (2.2).

2. Select the mode with the minimum cost as the best full-pixel mode.

3. Calculate R_{SAD}, the ratio between SAD of the second best mode and SAD of the best mode.

4. If R_{SAD} is above a certain threshold, $T_{R_{SAD}}$, take this best mode for IME as the best final mode and no other modes need to be considered in FME stage (in our experiments, we choose $T_{R_{SAD}} = 1.1$).

TABLE.2.2 shows part of our experimental data based on this scheme. All the test videos are from [2], covering videos with different resolution and motion intensities. Fig.2.19∼2.27 show the R-D curves in TABLE.2.2. The simulation is based on JM 9.0 (without RDOPT) and conducted under Windows XP operation system, with Intel Core TM2 Duo 2.2 GHz CPU and 2 GB RAM.

We compared the test results using this scheme with test results with regular flow, which calculates all the modes in FME. In this table, "MB Hit Rate" is the percentage of the MBs that satisfies the condition in step 4 above. To get the "Mode Hit Rate", we take all the MBs which satisfy the condition in step 4 above, and compare their best modes with the best modes we obtained from the regular motion estimation flow. If these two best modes are identical, it is marked as "Mode Hit".

The experimental data confirms our analysis that if the best mode in IME satisfies the condition in step 4, there is a high probability for the IME best mode to be the final best mode. In most cases, the smaller the QP, the higher the "Mode Hit Rate".

TABLE.2.2 also shows the speed-up for the FME process and the entire encoding process when the proposed scheme is used. In TABLE.2.2, we define "FME Speed-Up" as the ratio of FME processing time without vs. with the mode reduction scheme proposed in this paper, and define "System Speed-Up" as the ratio of overall encoding processing time without vs. with the mode reduction scheme proposed in this paper. As is well known, the "System Speed-Up" also depends on the algorithms used in other modules, such as the search strategies and early termination schemes of IME, entropy coding selection, etc. In our simulation, fast search algorithm UMHexagonS (Unsymmetrical-cross Multi-Hexagon-grid Search) [48] with a searching range of 16 was used in IME and CAVLC was used for entropy coding. Since these non-FME algorithms are not the focus of this paper, the Speed-Up comparison focuses on the speed-up provided by FME mode reduction alone.

As shown in TABLE.2.2, the speed-up factor achieved depends on video content and quantization parameter. it is important to note that the sacrifice for gaining these

Table 2.2: Experimental Data for FME Mode Reduction

Video Name	Resolution	QP	MB Hit Rate	Mode Hit Rate	Speed-Up	
					FME	System
Foreman	QCIF	24	36.4%	94.0%	1.42	1.16
		12	60.0%	91.8%	1.83	1.23
Trevor	QCIF	24	32.6%	95.9%	1.34	1.15
		12	46.2%	98.9%	1.53	1.18
Ice	CIF	24	22.8%	93.7%	1.21	1.10
		12	32.4%	97.6%	1.29	1.13
Highway	CIF	24	10.4%	96.3%	1.09	1.05
		12	60.5%	99.5%	1.75	1.25
Stefan	SIF	24	42.5%	98.0%	1.44	1.15
		12	53.6%	98.9%	1.69	1.20
Mobile	SIF	24	35.1%	95.9%	1.36	1.12
		12	43.6%	96.9%	1.49	1.15
Crew	4CIF	24	18.4%	89.7%	1.16	1.07
		12	80.6%	99.4%	2.62	1.40
Soccer	4CIF	24	21.4%	96.9%	1.18	1.08
		12	46.4%	98.7%	1.53	1.18
Crowd Run	1080P	24	44.3%	99.3%	1.48	1.18
		12	71.5%	99.8%	2.22	1.31

speed-ups is almost negligible: (a) the computational cost is just the comparison of the SADs for different prediction modes; (b) video quality drop is also very small. Fig.2.19∼2.27 show example R-D curves. These figures show that for the same bitrate, most of the PSNR drops are within 0.02 dB.

Considering Fig.2.18 and TABLE.2.2 together, it can be seen that when QP is small, a large portion of the MBs satisfy step 4 and can skip FME. Mode estimation in FME can be performed on only these MBs that do not satisfy step 4.

The method is also applicable when $QP = 24$, but the FME speed-up will be smaller than applications with a smaller QP. Actually, even when "MB Hit Rate" is 0, the extra computational cost introduced by this method is almost negligible since it is just to compare the SADs for different prediction modes.

This scheme can be extended by creating a subset of all the possible modes, and comparing the minimum SAD in a subset with the SAD with other modes outside this subset. If R_{SAD} satisfy the condition in step 4, we only need to compute the modes in this subset in FME. This way, we only need the modes in the subset to get a "Mode Hit" to save part of the mode decision computation.

2.3 Parallel Architecture for FME

This section presents a new VLSI architecture for fractional motion estimation (FME) in H.264/AVC. Statistical characteristics of the motion vectors of different inter-prediction modes are analyzed. The FME architecture explored block-level parallelism and can process multiple blocks with the same prediction mode simultaneously, external memory data accesses are dramatically reduced. Simulation results show that the proposed architecture can support 1080P (1960x1088) at 30fps with a frequency of

80MHz.

As the newest international video coding standard, H.264/AVC, provides much better image quality and compression ratio, compared with previous standards. This mainly comes from many new techniques, such as variable block size motion estimation (VBSME), pixel fractional motion estimation, multiple reference frame (MRF), in-the-loop deblocking filter and so on.

However, the intensive computation of H.264/AVC is the major obstacle for real-time encoding systems, especially consider that H.264/AVC requires almost 10 times the computation of previous coding standards. In H.264/AVC hardware encoder[8], the ME consists of 2 stages, the integer motion estimation (IME) for 7 block modes, and FME with 1/2 and 1/4 pixel accuracy.

In the IME stage, all the 7 block modes are processed in parallel with the SAD-reusing technique, and the integer motion vectors (IMV) of the 41 sub-blocks are then transferred to FME. In the FME stage, because these 41 sub-blocks have different IMVs, all the sub-blocks should be separately processed. Moreover, because 4x4 block size is the smallest unit for all the block modes, usually, a 4x4 based PU is adopted and all bigger blocks were divided into 4x4 to fully utilize the hardware, e.g. see [10]. Then the FME compuatation is in direct proportion to the number of block modes. For H.264/AVC with 7 block modes and a single reference frame, there are 7x16 = 112 4x4 blocks. Lack of block level parallelism, blocks like 16x8, 8x16, 8x8, etc, have to be processed sequentially, e.g. [31], [10] and [54]. To overcome this difficulty, some fast algorithms are brought forward, e.g. [37], bilinear filters are used to replace 6-tap FIR filters, and only single-iteration FME is needed. However, this inevitablly introduced prediction error propagation. Considering other factors, like irregular memory accessing, obtaining FME from IME becomes the bottleneck for the whole encoding system.

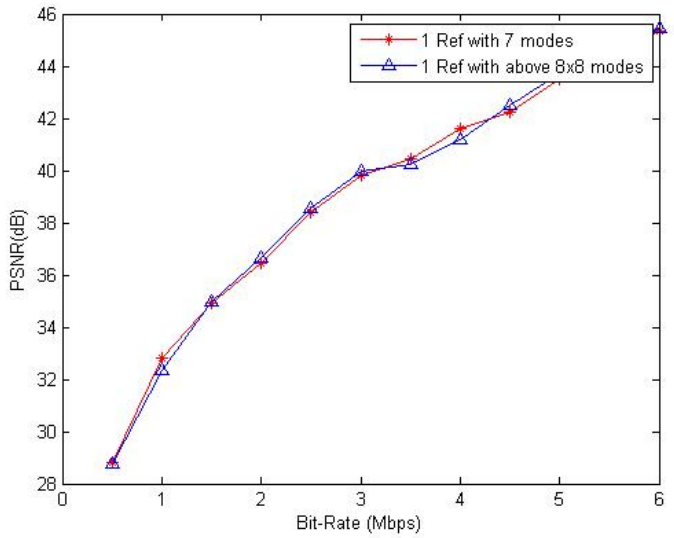

Figure 2.28: Mode statistics for STEPHEN.SIF

In order to improve the FME processing speed and keep the same encoding performance, in this paper, we explored block-level parallelism and a hardware architecture for parallel processing of two blocks (equal or greater than 8x8) is proposed. Also, by adopting a shifting register array, intensive access to reference memory is significantly alleviated.

2.3.1 H.264/AVC FME Observations

2.3.1.1 Encoding with INTER_8x8 mode or above

Our experiments show that for natural images, modes above 8x8 dominate the final MB modes. Moreover, experiments have shown that the image quality even has slightly better without the small block modes. Fig.2.28 shows a case with high motion intensities and complex content details. Similar observations can also be found in [31]. Therefore, in our realized FME engine, the block modes below 8x8 are all removed, which saves about 40% of the compuatational complexity.

2.3.1.2 Statistic charactistics of motion vectors

For FME, the fact that different blocks may have different IMVs prevents the encoder from parallel processing. The relative position of neighboring blocks is not fixed, which adds difficulties to multi-FME engines data fetching.

However, natural images are likely to have high correlations among neighboring motion vectors (MV). Fig.2.29 is one of our experiment results. Delta MVs are the absolute difference values between the motion vectors of neighboring blocks. Eqs (2.29) and (2.30) are the formula for calculating delta MVs for mode 16x8 and mode 8x16. Delta MVs for mode 8x8 is more complicated. For that mode, neighboring blocks include both horizontal and vertical neighbors. With the reference codec, JM16, we did experiments for most of the test medias in [2], we found more than 90% of the delta MVs are less than 4. This is also the motivation of our FME architecture.

$$dmv\_x = |blk0\_mv\_x - blk1\_mv\_x| \tag{2.29}$$

$$dmv\_y = |blk0\_mv\_y - blk1\_mv\_y| \tag{2.30}$$

2.3.2 The Proposed Architecture

For H.264/AVC FME design, three important issues need to be considered. The first is the data fetching, accesses to external memories need to be minimized. The second is the local memory size for FME, it should be minimized because about 80% of the chip area is occupied by on-chip memories in an ASIC. The third is the data sharing and parallelism, which determines the FME processing speed and system frequency. The proposed architecture improves on all three issues over prior art designs.

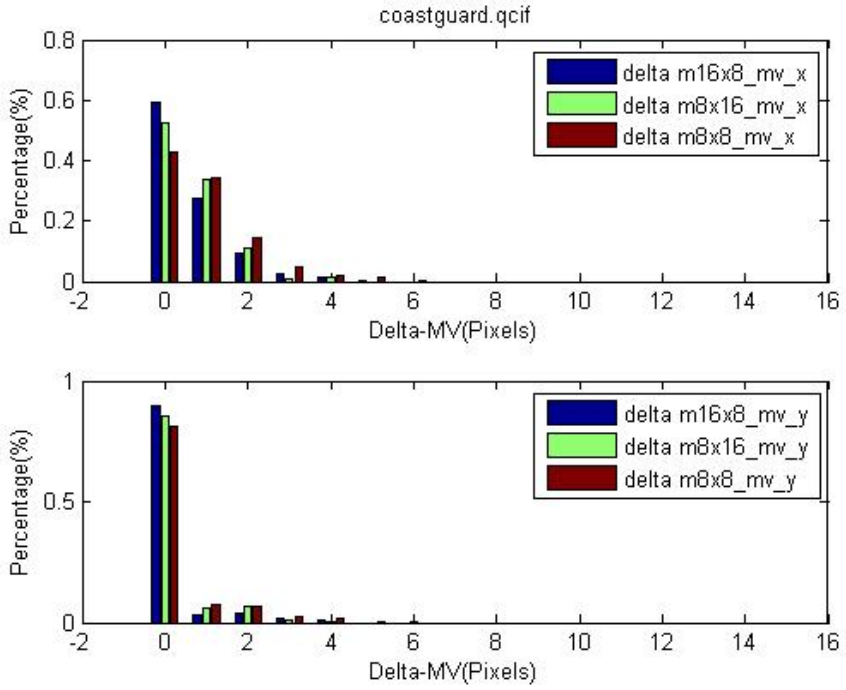

Figure 2.29: IMV Statistics for COASTGUARD.QCIF

Fig.2.30 shows our proposed FME system architecture. It consists of a control FSM, a reference pixel array, a sampler line mux, a current MB pixel array, two FME engines for interpolation and cost calculation, a unit for comparator and output buffer for motion compensation (MC). The FME Control FSM gets optimal IMVs and current Ref-Array locations from the IME engine, it controls the shifting directions of both the Ref-Array and the Cur-MB Array, also the cooperation of all the internal FME sub-modules described below.

2.3.2.1 Reference Pixel Array

In our design, a 32x32 pixels array is used for the reference window. This array was used by IME before it goes to FME. Two such arrays are used to enable parallel ping-pong processing between IME and FME.

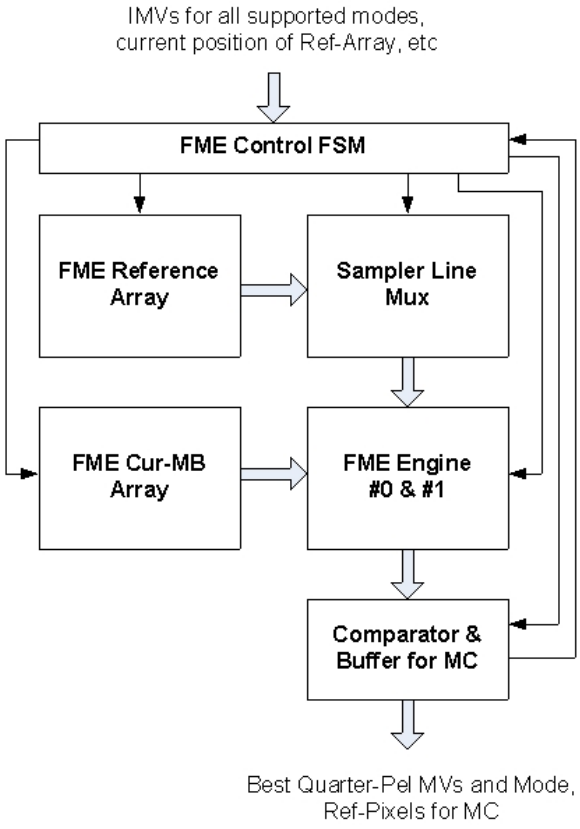

Figure 2.30: FME System Architecture

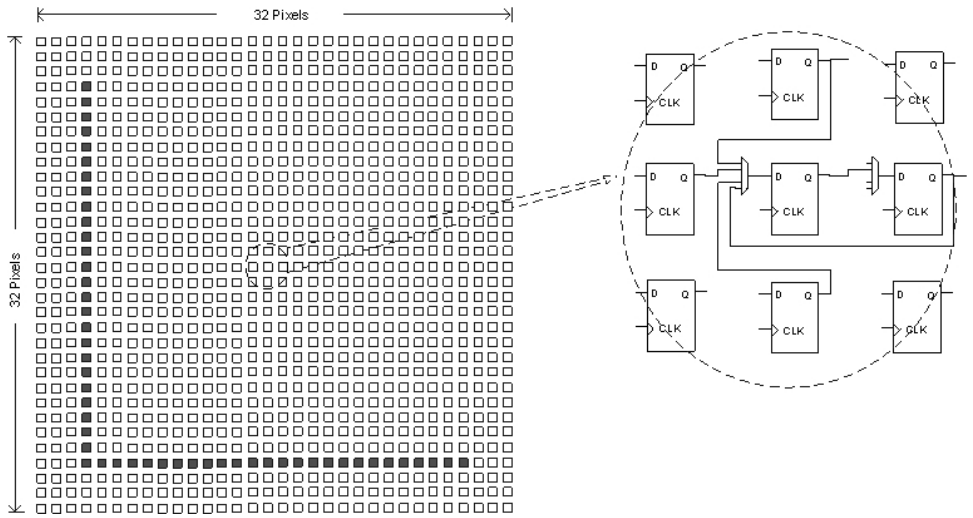

Figure 2.31: FME Reference Array

As shown in Fig.2.31, the reference array can shift in both horizontal and vertical directions. Each cell consists of an 8-bit flip-flop and a 4-to-1 mux. The mux is programmed by the FME control FSM, determining the shifting direction of the reference array.

This reference array is fetched by the IME engine and is handed over to FME engine once IME is done. So, the reference data is only loaded once and used by both IME and FME. This reduces intensive memory access and irregular memory addressing issues.

2.3.2.2 Integer Pixel Sampler in Reference Array

In the reference array, we have a horizontal and a vertical sampler lines, both with 26 samplers. The positions of these two sampler lines are fixed in the array. Only pixels in these lines can be sampled and used for FME.

At any time, only one of these two sampler lines is active. That is, when the array is shifting horizontally, the vertical sampler line will be used; and when the array

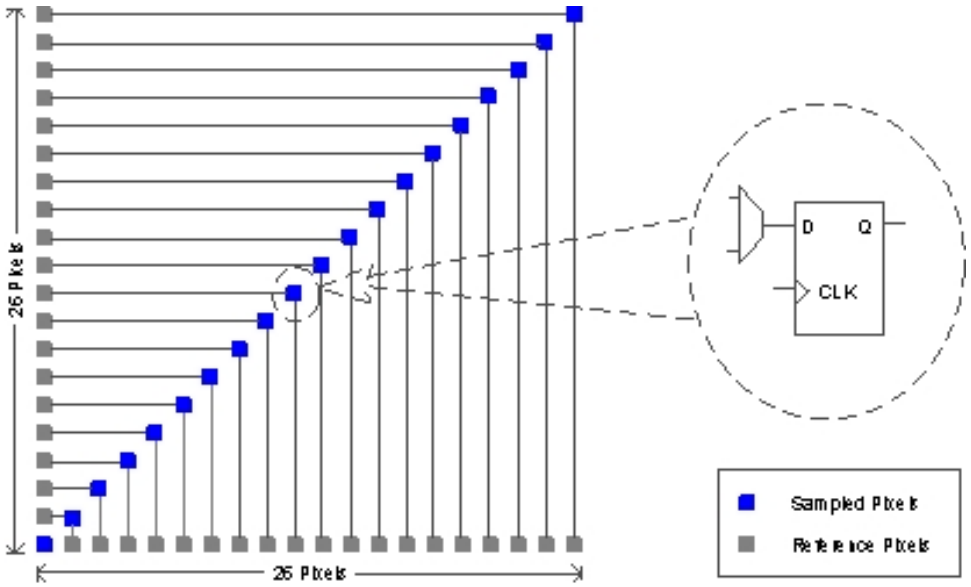

Figure 2.32: Sampler Line Mux

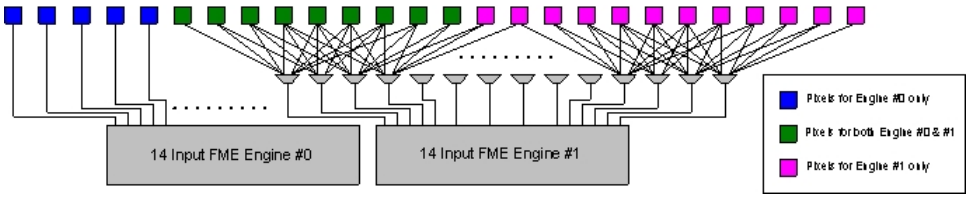

Figure 2.33: Two 14-Input FME Engine

is shifting vertically, the horizontal sampler line will be used.

As shown in Fig.2.32, 2-to-1 muxs are used to select either the horizontal sampler line or the vertical sampler line. Muxs are configured by the FME control FSM.

2.3.2.3 14-Input FME Engine

The two 14-Input (Ref-pixels) FME Engine working in parallel are the key parts of the proposed architecture. As show in Fig.2.33, these colored input units are from the mux outputs in Fig.2.32.

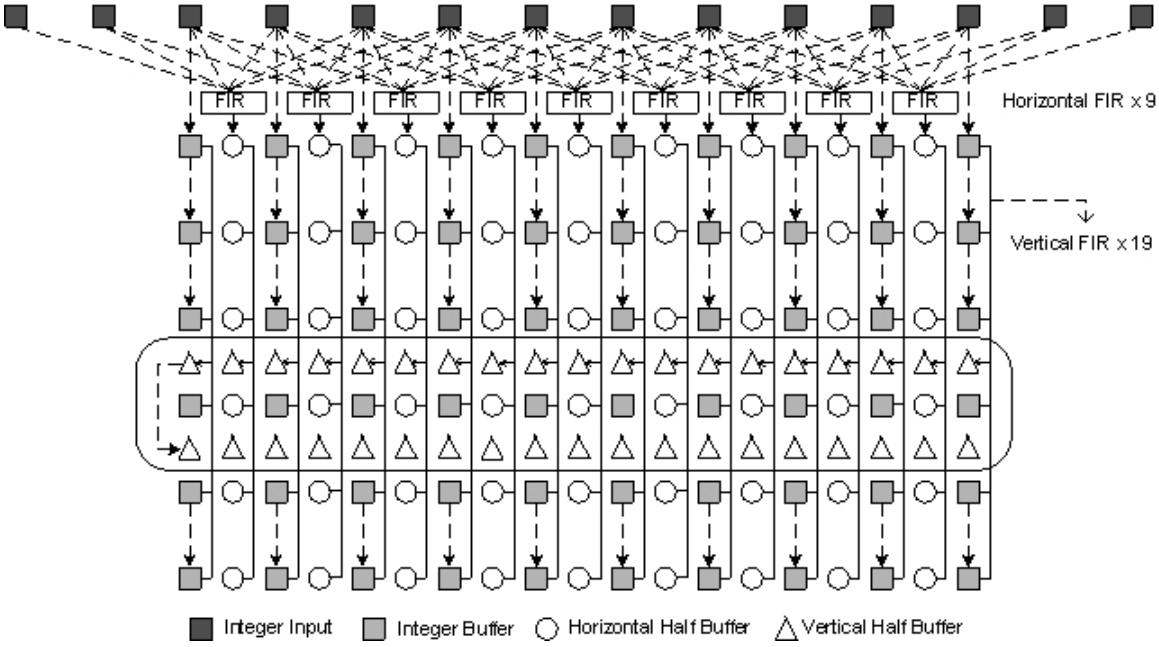

Figure 2.34: Interpolation Unit inside FME Engine

The FME engines make use of the statistical characteristics of IMVs which we discussed above. 14 8-to-1 muxs are adopted for the inputs of the second FME engine. So, the delta IMVs between neighboring blocks do not have to be zero, they can be any values from 0 to 4, which counts more than 90% of the cases. For INTER8x8 mode, the situation is a little bit more complicated, but we still can benefit form this architecture.

Inside every FME engine, an interpolation engine shown as Fig.2.34 is used. It takes 14 integer reference pixels as inputs, and in every clock cycle, it will produce 8 half-pels around every selected 8 integer pixels pointed by IMVs, or 8 quarter-pels around selected 8 half-pels pointed by half-pel motion vectors (bilinear filters were not drawn in Fig.2.34).

Once we picked a sampler line, the input pixels to FME engine 0 are always from the fixed locations (pixel_0 to pixel_13). On the other side, we used 14 8-to-1 muxs for FME engine 1 to select input pixels from 21 locations. The usage of muxs is to

accommodate the different motion vectors among neighboring blocks.

Briefly, the tasks for these 2 FME engines include: half-pel interpolation and motion search among 8 half-pel positions, quarter-pel interpolation and motion search among 8 quarter-pel positions, dump reference pixels (integer-pel, half-pel or quarter-pel) to MC stage. It will output SATD values and corresponding motion vectors for different prediction modes, from INTER16x16 to INTER8x8.

2.3.2.4 Data Processing Order

The FME will always start from INTER8x8 mode, since it can be processed either horizontally or vertically. As shown in Fig.2.35 and Fig.2.36, there are two data processing orders. Depending on the MV distribution of 4 blocks in INTRA8x8 mode, FME control unit will compute which direction will consume less processing cycles before the Ref-Array shifting starts.

In Fig.2.35 and Fig.2.36, the blue bars stand for the active sampler lines. Once the FME control FSM decides to shift 4 INTER8x8 reference blocks left, the data processing order will be like Fig.2.35, followed by INTER16x8 mode, INTER8x16 mode and INTER16x16 mode. If the FME control FSM decides to shift 4 INTER8x8 reference blocks down, the data flow will be like Fig.2.36, followed by INTER8x16 mode, INTER16x8 mode and INTER16x16 mode.

In the best case, FME engine 0 and FME engine 1 can work simultaneously. It can process block0 and block 1 in INTER16x8 mode, INTER8x16 mode, two half parts in INTER16x16 mode or neighboring 8x8 blocks in INTER8x8 modes. However, the parallel processing has to satisfy certain conditions. For INTER8x16 mode, Eq.(2.31) should be satisfied and for INTER16x8 mode, Eq. (2.32) should be satisfied, where

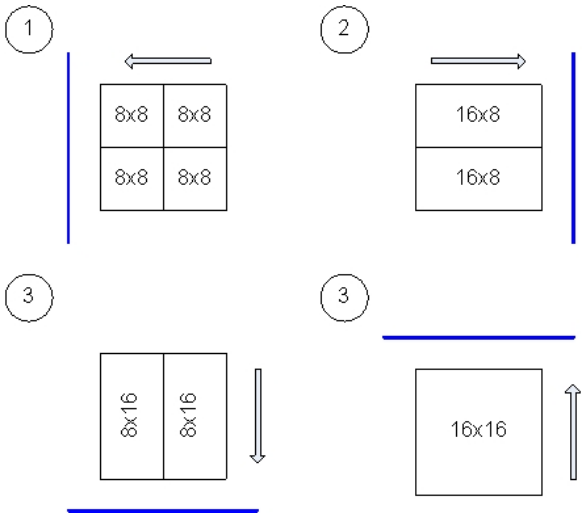

Figure 2.35: FME Processing Flow Option 0

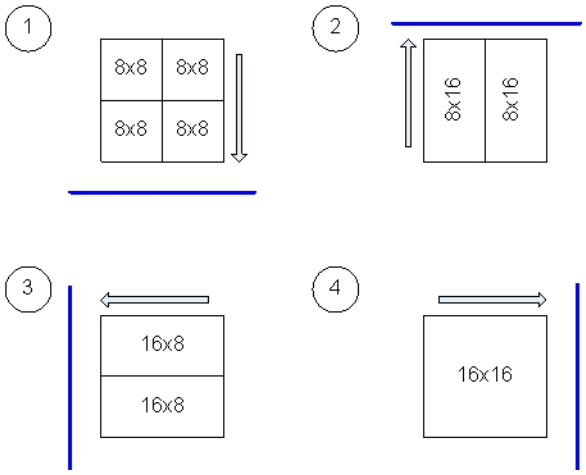

Figure 2.36: FME Processing Flow Option 1

MUX_DIST is 4 in our experiments. INTER_8x8 mode has a much more complicated algorithm, which will not be discussed in here.

$$|blk0\_mv\_x - blk1\_mv\_x| \leq MUX\_DIST \tag{2.31}$$

$$|blk0\_mv\_y - blk1\_mv\_y| \leq MUX\_DIST \tag{2.32}$$

If the current processing mode does not meet the parallel processing requirement, a *miss* penalty will be applied. For example, if INTER8x16 mode does not satisfy Eq.(2.31), then after it finishes FME of block 0 in engine 0, it has to scan back for block 1 and shifts it to the desired position for FME processing in engine 1. In this scenario, a certain number of bubble cycles are generated.

2.3.2.5 3-Stages Processing

The interpolation structure in Fig.2.34 can generate both half-pels and quarter-pels for FME. However, if we do both of them in a single iteration, a lot of memory cells will be used to store these sub-pels. In order to save expensive on-chip memory space, a hierarchical processing flow consisting of 3 stages is adopted in our design.

In the first stage, half-pel FME will be conducted as stated in section D. The best half-pel MVs (HMV) will be generated. In the second stage, quarter-pel FME is conducted and the best quarter-pel MVs (QMV) and the best INTER mode is generated. In the third stage, the reference pixels based on best mode and best QMV is output to MC buffer.

Table 2.3: PART OF THE SIMULATION RESULTS

Video Name	Motion Intensity & Content Detail	Avg Cycles per MB
Container	Less	198
Coastguard	Median	198
Foreman	Median	226
Highway	High	225
Stephen	Very High	319

2.3.3 Simulation Results

The proposed FME architecture and data processing order have been simulated in a SystemC implementation. Test videos with different content details and motion intensities have been tested. In most of the experimental results, the average FME processing periods are around 200 clock cycles as TABLE.2.3 shows. Compared with most of state of the art architectures, like [10] [39] and [1], it reduced about 67% memory accesses and about 50% processing cycles. Fig.2.37 shows the possibility distributions of number of processing cycles per MB for several video sequences with different motion intensity and content details. Our simulation also shows that for almost all the test videos in [2] (up to 1080P 30fps), a clock frequency of 80MHz will be sufficient.

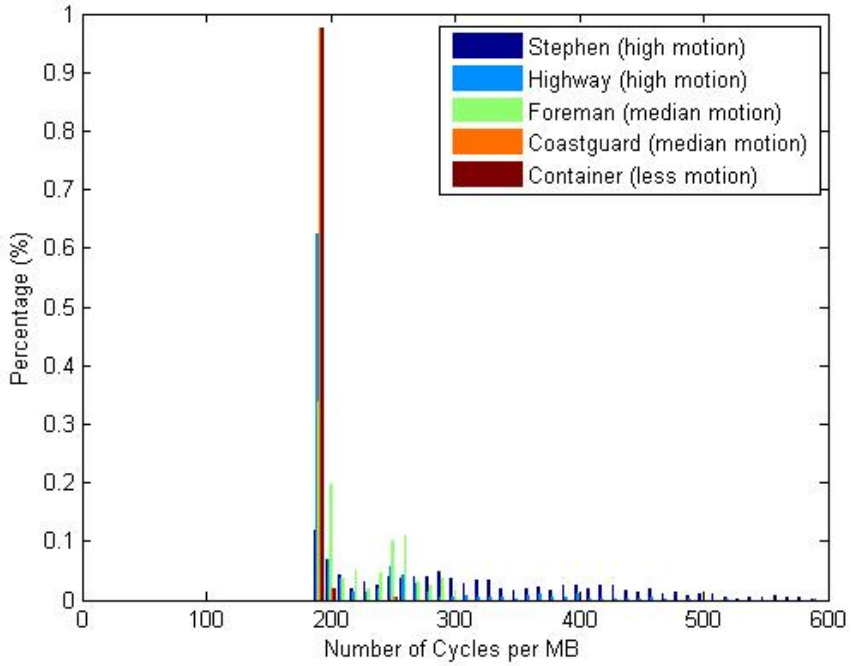

Figure 2.37: Processing Cycles Distributions

Chapter 3

Typical Parallel Architecture for Video Encoding

3.1 Task-Level Decomposition

In a task-level decomposition the functional partitions of the algorithm are assigned to different processing units. Pipeline based architectures, such as .[8], belong to this category.

Task-level decomposition requires significant communication between the different tasks in order to move the data from one processing stage to the other, and this may become the bottleneck. This overhead can be reduced using double buffering and blocking to maintain the piece of data that is currently being processed in cache or local memory. Additionally, synchronization is required for activating the different modules at the right time. This should be performed by a control processor and adds significant overhead.

The main drawbacks, however, of task-level decomposition is scalability. If the application requires higher performance, for example by going from standard to

high definition resolution, it is necessary to re-implement the task partitioning which is a complex task and at some point it could not provide the required performance for high throughput demands. Finally from the software optimization perspective the task-level decomposition requires that each task/processor implements a specific software optimization strategy, i.e., the code for each processor is different and requires different optimizations.

3.2 Data-Level Decomposition

In a data-level decomposition the work (data) is divided into smaller parts and each assigned to a different processor. Each processor runs the same program but on different (multiple) data elements (SPMD).

In H.264 data decomposition can be applied at different levels of the data structure (see Figure.3.1). At the top of the data structure there is the complete video sequence. This video sequence is composed out of Group of Pictures (GOPs) which are independent sections of the video sequence. GOPs are used for synchronization purposes because there are no temporal dependencies between them. Each GOP is composed of a set of frames, which can have temporal dependencies when motion prediction is used. Each frame can be composed of one or more slices. The slice is the basic unit for encoding and decoding. Each slice is a set of MBs and there are no temporal or spatial dependencies between slices. Further, there are MBs, which are the basic units of prediction. MBs are composed of luma and chroma blocks of variable size. Finally each block is composed of picture samples. Data-level parallelism can be exploited at each level of the data structure, each one having different constraints and requiring different parallelization methodologies.

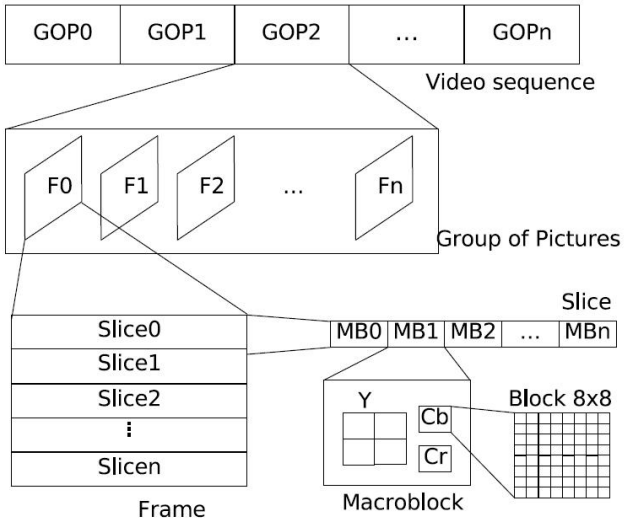

Figure 3.1: H.264/AVC Data Structure

3.2.1 GOP-Level Parallelism

The coarsest grained parallelism is at the GOP level. H.264 can be parallelized at the GOP-level by defining a GOP size of N frames and assigning each GOP to a processor. GOP-level parallelism requires a lot of memory for storing all the frames, and therefore this technique maps well to multi-computers in which each processing node has a lot of computational and memory resources. However, parallelization at the GOP-level results in a very high latency that cannot be tolerated in some applications. This scheme is therefore not well suited for multi-core architectures, in which the memory is shared by all the processors, because of cache pollution.

3.2.2 Frame-Level Parallelism for Independent Frames

After GOP-level there is frame-level parallelism. In a sequence of I-B-B-P frames inside a GOP, some frames are used as reference for other frames (like I and P frames) but some frames (the B frames in this case) might not. Thus in this case the B

frames can be processed in parallel. To do so, a control processor can assign independent frames to different processors. Frame-level parallelism has scalability problems due to the fact that usually there are no more than two or three B frames between P frames. This limits the amount of thread-level parallelism (TLP) to a few threads. However, the main disadvantage of frame-level parallelism is that, unlike previous video standards, in H.264 B frames can be used as reference [15]. In such a case, if the decoder wants to exploit frame-level parallelism, the encoder cannot use B frames as reference. This might increase the bit-rate, but more importantly, encoding and decoding are usually completely separated and there is no way for a decoder to enforce its preferences to the encoder.

3.2.3 Slice-Level Parallelism

In H.264 and in most current hybrid video coding standards each picture is partitioned into one or more slices. Slices have been included in order to add robustness to the encoded bitstream in the presence of network transmission errors and losses. In order to accomplish this, slices in a frame should be completely independent from each other. That means that no content of a slice is used to predict elements of other slices in the same frame, and that the search area of a dependent frame can not cross the slice boundary [44] [23]. Although support for slices have been designed for error resilience, it can be used for exploiting TLP because slices in a frame can be encoded or decoded in parallel. The main advantage of slices is that they can be processed in parallel without dependency or ordering constraints. This allows exploitation of slice-level parallelism without making significant changes to the code.

However, there are some disadvantages associated with exploiting TLP at the slice level. The first one is that the number of slices per frame is determined by the

encoder. That poses a scalability problem for parallelization at the decoder level. If there is no control of what the encoder does then it is possible to receive sequences with few (or one) slices per frame and in such cases there would be reduced parallelization opportunities. On the other hand, although slices are completely independent of each other, H.264 includes a deblock filter that can be applied across slice boundaries. This is also an option that is selectable by the encoder, but means that even with an input sequence with multiple slices, if the deblock filter crosses slice boundaries the filtering process should be performed after the frame processing in a sequential order inside the frame. This reduces the speed-up that can be achieved from slice-level parallelization. Another problem is load balancing. Usually slices are created with the same number of MBs, and thus can result in an imbalance at the decoder because some slices are decoded faster than others depending on the content of the slice.

Finally, the main disadvantage of slices is that an increase in the number of slices per frame increases the bit-rate for the same quality level (or, equivalently, it reduces quality for the same bit-rate level). [24] shows the increase in bit-rate due to the increase of the number of slices for four different input videos at three different resolutions. The quality is maintained constant (PSNR=40). When the number of slices increases from one to eight, the increase in bit-rate is less than 10%. When going to 32 slices the increase ranges from 3% to 24%, and when going to 64 slices the increase ranges from 4% up to 34%. For some applications this increase in the bit-rate is unacceptable and thus a large number of slices is not possible. As shown in the figure the increase in bit-rate depends heavily on the input content. The river bed sequence is encoded with very little motion estimation, and thus has a large absolute bit-rate compared to the other three sequences. Thus, the relative increase in bit-rate is much lower than the others.

3.2.4 Macroblock-Level Parallelism

There are two ways of exploiting MB-level parallelism: in the spatial domain and/or in the temporal domain.In the spatial domain MB-level parallelism can be exploited if all the intra-frame dependencies are satisfied. In the temporal domain MB-level parallelism can be exploited if, in addition to the intra-dependencies, inter-frame dependencies are satisfied.

In this thesis, we proposed a Macroblock-Level parallel architecture named "Wavefront architecture". It exploits both spatial and temporal parallelism and will be discussed in chapter 4 and chapter 5. According to [24], wavefront architecture so far has been the most scalable approach to H.264 coding.

3.2.5 Block-Level Parallelism

Since "variable block size" is one of the primary features of H.264/AVC, block-level parallelism becomes possible. As in section 2.3, we proposed a block-level parallel architecture for FME. However, scalability is still an issue due to the limited number of blocks in a MB.

Chapter 4

Wavefront Configurable Parallel Architecture

In this chapter, we will presents a new method for parallel processing of H.264 video encoder using data partition and task scheduling that fully exploits all data dependencies for maximal compression. The new method achieves the optimal compression at a frame rate that increases approximately linearly as the number of parallel processing elements. This is a significant improvement over prior art parallel encoders for H.264 which invariably sacrifice data dependency and/or optimal coding mode. It is shown that the new method outperforms prior approaches in both encoding speed and compression efficiency. This paper also gives the relation between the number of parallel processing elements and the theoretical encoding time and the relation between the number of processors and the number of concurrently processed frames. Software simulation shows that this parallel processing method achieves the same compression quality as a sequential processing encoder, e.g., the JM series.

Compared with MPEG-4 Simple Profile, up to 50% bitrate reduction is achieved

at the cost of more than four times of computational complexity [9]. Therefore, hardware or software acceleration, especially parallel structure, is a must for real-time applications.

Multiple-processor and multiple-threading encoding system have been used for real-time video encoding [12]. However, up to now, no satisfactory solution has been found that can partition video data for parallel processing while at the same time maximally exploit the temporal and spatial data dependencies for optimal coding efficiency.

Parallel algorithms are discussed in many papers,e.g., [12][18][3][35][53]. A single chip encoder for H.264 in [18] used a four-stage macroblock pipeline architecture. Although satisfactory R-D tradeoff is reported, it made approximations of neighboring encoding information in finding the optimal coding mode of the current MB. Therefore, its coding result can only be sub-optimal. An H.264 encoder using the Intel hyper-threading architecture is reported in [12]. It splits a frame into several slices and these slices are processed by multiple threads. However, these extra slices bring heavy overheads because of the slice header and the impairments to data dependencies among MBs. Y.K.Chen,et al. performed experiments to find relations between bit-rate and the number of slices in a picture [12]. In another word, the approach in [12] sacrifices coding efficiency in exchange for parallel threading. In [3], a frame is divided into many small partitions with overlapping areas and these partitions are processed concurrently. Unfortunately, this kind of partition is not feasible for H.264, because the encoder needs previous MVs in raster-scan order.

We will show in this chapter that how our new method does not suffer from the problems faced by prior art approaches.

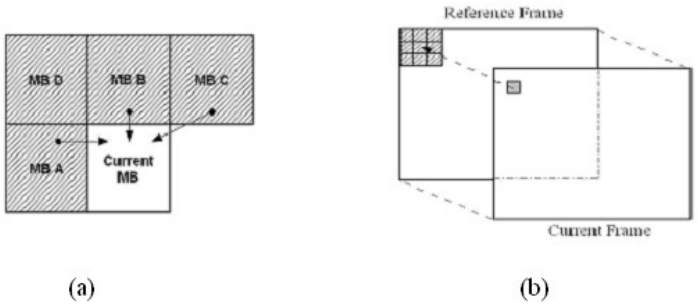

Figure 4.1: Intra- & inter-frame data dependencies

4.1 Primary Data dependencies in H.264/AVC

The reference software JM 9.0 for H.264/AVC provided by JVT adopts sequential processing of each macroblock (MB) and creates data dependencies that makes parallel processing difficult. However, by exploring these data dependencies, the JM encoders can produce the optimal bitstream in terms of coding efficiency, therefore, the highest compression ratio. For easy comparison with the JM, this paper only considers coding efficiency. We realize that optimal subjective quality may have different requirements, however it is not dealt in this paper.

Our objective is to explore elements of the encoder that can be processed in parallel and at the same time, maximally exploit the temporal and spatial data dependencies for optimal coding efficiency.

4.1.1 Predicted Motion Vector & Inter-prediction

In inter-prediction, predicted motion vector (PMV) defines the search center of motion estimation. This variable is very useful in maintaining continuity of the motion field. It is determined by the MVs of its neighboring subblocks (MV_A, MV_B and MV_C) and the corresponding reference indexes [2], as shown in Fig.4.1 (a). Only the difference

(MVD) between the final optimal motion vector (MV') and PMV will be encoded.

Similar to previous standards, H.264 also needs the reconstructed images from encoded frames as reference to exploit temporal redundancy. Thus, at least the co-located MB and its eight neighboring MBs in the reference frames must be available before current MB can be encoded, as shown in Fig.4.1 (b).

4.1.2 Quarter-pel interpolation and deblock-filtering

In H.264, traditional half-pel accuracy prediction is extended to quarter-pel. Before the reconstructed result of current MB can be used as reference for its next frame's coding, it must be interpolated to get these pixel values in quarter-/half-pel positions. This operation in the boundary area of current MB needs 3 rows/columns of pixels from its neighboring MBs (A, B, C, D, see Fig.4.1 (a)).

Also, in order to compensate the block artifacts brought by block-based ME and transform. H.264 uses a adaptive deblock-filter. The filter-strength factor is also determined by the prediction mode and pixel values of neighboring blocks [14]

4.1.3 4×4 & 16×16 intra-prediction & mode decision

H.264 has 9 4×4 intra-predictions and 4 16×16 intra-predictions. For detail, see [14]. As shown in Fig.4.2, the dark pixels from neighboring blocks of current MB are needed for intra-prediction.

4.1.4 Context-adaptive variable length coding (CAVLC)

CAVLC is another powerful tool which makes H.264 efficient. The first VLC, coeff_token, encodes both the total number of non-zero transform coefficients (TotalCoeffs) and the number of trailing $\pm 1 (T1)$. There are 4 choices of look-up table to use for

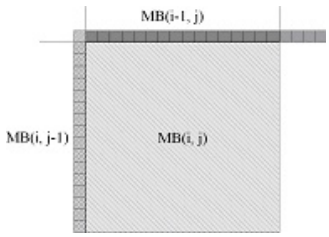

Figure 4.2: Intra-prediction data dependencies

encoding coeff_token, described as Num-VLC0, Num-VLC1, Num-VLC2 and Num-FLC (3 variable-length code tables and a fixed-length code). The choice of table depends on the number of non-zero coefficients in upper and left-hand previously coded blocks N_U and N_L.

4.2 Data partition and task priority

4.2.1 Data partition

From section 4.1, we observe the following:

- MBs in different frames can be processed concurrently, only if its necessary reconstructed MBs from reference frame are all available.
- MBs from different MB rows in the same frame can also be processed concurrently, only if its neighboring MBs in its top MB row all have been encoded and reconstructed.

Fig.4.3 is an illustration of this observation. This simple but powerful observation is the basis of our new data partition and task scheduling method, hereafter referred to as Wavefront Parallelization. In this example, three frames are concurrently processed. The gray areas represent MBs which have been encoded and checkered MBs

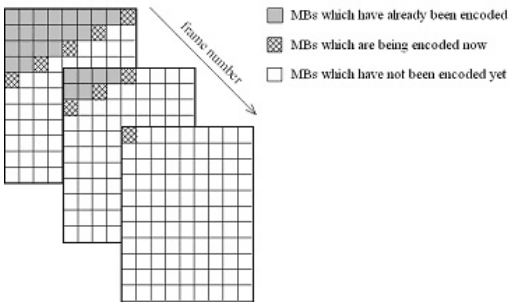

Figure 4.3: Concurrently processed MBs

are being encoded concurrently now.

A distinguishing property of Wavefront Parallelization is that if there are sufficient numbers of processors and the video sequence is long enough, Wavefront Parallelization can achieve a constant frame rate regardless how large the video format is, e.g., QCIF, CIF or HDTV720. Taking Fig.4.3 for instance, the encoding process of a new frame can be started as soon as the 2x2 MBs in the top-left corner of its previous frame are all encoded, because this is the minimum requirement for motion search of inter-prediction. This means, with the increase of frame number, the average encoding time for a frame approaches to only 4 TMB (where TMB is the encoding time for a macroblock). The number of processor units needed to achieve this is:

$$P_n = \lfloor frame\_size\_in\_MB/4 \rfloor \tag{4.1}$$

In practice, we cannot have a large number of processor units. We will show that majority of the benefit can be achieved with only a small number of processor units.

In Wavefront Parallelization, each frame is first partitioned into MB rows as shown in Fig.4.4. This is because a MB cannot be processed until its left neighbor in the same MB row is encoded. All MBs in the same MB row will be processed by the

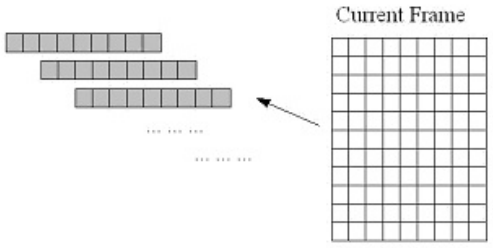

Figure 4.4: Processing units in a frame

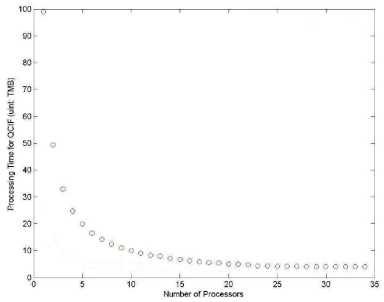

Figure 4.5: Theoretical processing time per frame for QCIF

same processor or thread to reduce data exchanges between processors.

4.2.2 Task assigning and priorities

Fig.4.7 gives the task assigning timing diagram for the ideal case (number of processors = P_n). Here, $T = TMB$. However, this requires 25 processors for QCIF to achieve the optimal encoding speed; 99 processors for CIF, and 900 for HDTV720P. This is not practical.

Fig.4.5 is our simulation result of the theoretical encoding time of a video frame with QCIF format. The results for CIF, HDTV720P are also similar. This figure shows the relation between the number of processors (horizontal axis) and the encoding time of

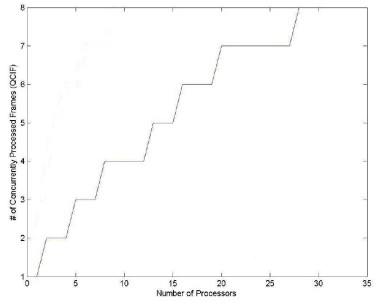

Figure 4.6: Number of concurrently processed frames (QCIF)

Table 4.1: Simulation Result for "Grandma.yuv" (QCIF)

	Avg enc time per frame	SnrY	SnrU	SnrV	# of bits	Speed Up
Wavefront simulator	273 ms	37.157	39.869	40.450	61464	3.17
JM 9.0	865 ms	37.157	39.869	40.450	61464	1

Table 4.2: Simulation Result for "Paris.yuv" (CIF)

	Avg enc time per frame	SnrY	SnrU	SnrV	# of bytes	Speed Up
Wavefront simulator	1272 ms	35.729	39.181	39.279	128419	3.08
JM 9.0	3914 ms	35.729	39.181	39.279	128419	1

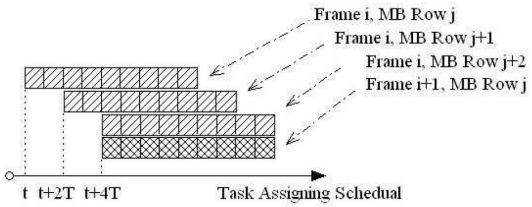

Figure 4.7: Task assignment timing diagram

a frame (vertical axis and with TMB as the unit). As can be seen, most of the speedup benefit can be achieved using a small number of processors.

If only a small portion of processors are used, not all the checkered MBs in Fig.4.3 that are ready for processing can be processed concurrently. Thus, priority based task scheduling is necessary to guarantee that the processors can be fully utilized.

Here we defined the priorities in two levels: the first is the inter-frame level, and the second is the intra-frame level. The priority of the inter-frame level is higher than the intra-frame level. The inter-frame level priority means that in the video source buffer, if several MBs belonging to different frames are ready to be encoded concurrently, the MBs in the frame with smaller frame number should be encoded first. The intra-frame level priority means that if several MBs belonging to different MB rows in the same frame are ready to be encoded concurrently, the MBs in the row with smaller row index should be encoded first.

4.3 Software simulation

Our wavefront simulator is developed using C language and implemented in a PC with a P4 2.8GHz processor and a 512MB memory. The simulation results are compared with those from JM 9.0, a sequential encoding structure.

In our software simulation of an encoder for H.264 baseline profile, four processors are simulated. Some of the encoder parameters are: one reference frame, searching range for ME: ± 10, Hadamard transform is used for motion cost estimation, full R-D optimization, CAVLC for entropy coding.

We also performed simulations to obtain the relationship between the number of processors (horizontal axis) and the number of concurrently processed frames (vertical

axis) for different video formats. Due to space limitation, only result for QCIF is given in Fig.4.6.

In order to simplify the encoder, we choose to encode 2 frames simultaneously at the most. From Fig.4.6, we found that 4 processors are the upper limit.

The simulator collects the maximal encoding time among every 4 concurrently processed MBs and the corresponding time spent on data partition and communication. Some of the simulation results are presented in Table.4.1 and Table.4.2. In Table.4.1, we used "Grandma.yuv" (QCIF) as video source, 50 frames are encoded as "IPPP" structure. In Table.4.2, we used "Paris.yuv" (CIF) as video source, 50 frames are encoded as "IPPP" structure. Experiments show that a speedup of more than 3 times is achieved and the encoding quality is the same as JM 9.0.

Chapter 5

System Simulation using Tensilica XTMP

5.1 XTENSA Processors

5.1.1 Processor Architectures

The Xtensa synthesizable processor core is based on Tensilica's Xtensa technology, the first configurable and extensible DPU architecture designed specifically to address embedded System-On-Chip (SOC) applications. The Xtensa architecture, as illustrated in Figure.5.1, was designed from the start to be configurable, which allows designers to precisely tailor each processor implementation to match the target SOC's application requirements. Members of the Xtensa processor family are unlike conventional embedded processor cores, they change the rules of the SOC game. Using Xtensa technology, the system designer molds each processor to fit its assigned tasks on the SOC by selecting and configuring predefined architectural elements and by inventing completely new instructions and hardware execution units that can deliver application-

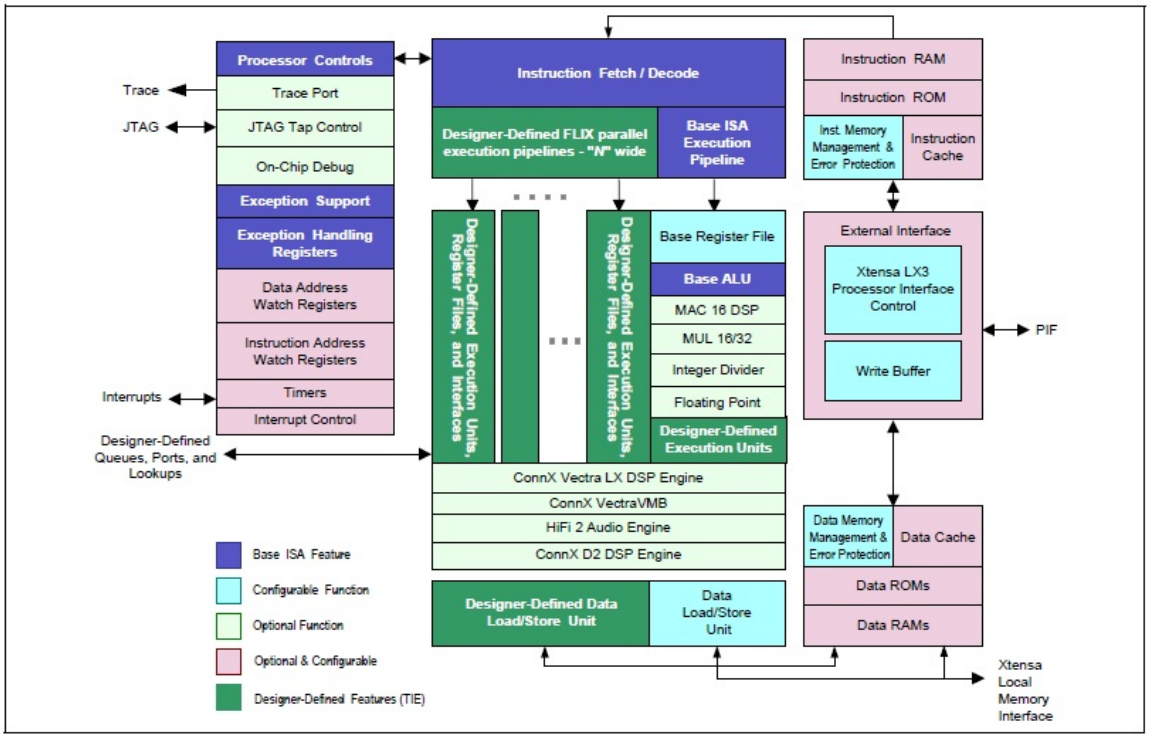

Figure 5.1: Xtensa LX3 Processor Architectural Block Diagram

specific performance levels that are orders of magnitude faster than alternative solutions. In addition to the processor core, the Xtensa Processor Generator automatically generates a complete, optimized software-development environment that includes customized operating system support for each processor configuration. The power and flexibility of the configurable Xtensa processor family make it the ideal choice for complex SOC designs.

Tensilica's Xtensa architecture consists of various standard and configurable building blocks. Configurable function blocks are elements parameterized by the system designer. Optional function blocks indicate elements available to accelerate specific applications. The optional and configurable blocks have optional elements (such as timer interrupts and interrupt levels) that can be individually scaled to fit specific applications. SOC hardware and firmware designers can add advanced functions to the processor's

architecture in the form of hardware execution units and registers to accelerate specific algorithms. Common to all configurations is the Xtensa base instruction set architecture (ISA).

Tensilica delivers five technologies to help designers build SOCs for embedded applications:

- The Xtensa processor architecture based on a highly configurable, extensible, and synthesizable 32-bit processor. Many designer-defined, application-specific families of processors can be built around the base Xtensa ISA to optimize factors such as code size, die size, application performance, and power dissipation. Designers define new processor instructions, execution units, and registers using the Tensilica Instruction Extension (TIE) language.

- A generated software tool suite to match the configured processor architecture. This tool suite includes the Xtensa C/C++ compiler (XCC), a macro assembler, linker, debugger, diagnostics, reference test benches, and a basic software library. XCC provides C++ capabilities equivalent to the GNU C++ compiler version 4.2. It improves code performance relative to GCC in many cases and provides vectorization support for the ConnX Vectra LX DSP Engine, for the ConnX D2 DSP Engine, and for TIE processor extensions generated by Tensilica's XPRES compiler.

- Xtensa Xplorer, which serves as a cockpit for single- and multiple-processor SOC hardware and software design. Xtensa Xplorer integrates software development, processor optimization and multiple-processor SOC architecture tools into one common design environment. It also integrates SOC simulation and analysis tools. Xtensa Xplorer is a visual environment with a host of automation tools that makes

creating Xtensa processor-based SOC hardware and software much easier. Xplorer serves as a cockpit for basic design management, invocation of Tensilica processor configuration tools (the Xtensa Processor Generator and the TIE compiler), and software development tools. Xtensa Xplorer is particularly useful for the development of TIE instructions that maximize performance for a particular application. Different Xtensa processor and TIE configurations can be saved, profiled against the target C/C++ software, and compared. Xtensa Xplorer includes automated graphing tools that create spreadsheet-style comparison charts of performance.

- A multiple processor (MP)-capable instruction set simulator (ISS) and C/C++ callable simulation libraries.

- The XPRES compiler: a software tool capable of automatically generating performance-boosting processor extensions based on the analysis of target C or C++ application source code. The XPRES compiler sets a new benchmark for the automation of processor design and vastly improved designer productivity. This tool analyzes the target software source code, quickly generates thousands or millions of alternative processor designs, evaluates these designs for performance and gate count, selects the optimal design based on designer-defined criteria, and finally produces a description of the selected processor in TIE, which is then submitted to the Xtensa Processor Generator.

All development tools are automatically built to match the exact configuration specified in the Xtensa Processor Generator. Together, these technologies establish an improved method for rapid design, verification, and integration of application-specific hardware and software.

5.1.2 Primary Features

The Xtensa processor offers complete and robust development tools, system building blocks, and packages to develop the target embedded system-on-chip solution:

- Xtensa Processor Generator
  - Automatic and rapid generation of RTL, companion software development tools, and simulation models.
- Tensilica Instruction Extension (TIE) language
  - Designer-defined instructions are easy to construct and are automatically integrated with the base processor.
  - The Xtensa LX3 processor features FLIX (flexible-length instruction extensions) technology, which allows designers to create wide instruction words (32 or 64 bits) with multiple operation slots. These multiple slots can contain operations that use the base Xtensa general-purpose register file and drive the base Xtensa ISA function units as well as a second load/store unit added through processor configuration and new registers, register files, and execution units added through designer-defined TIE descriptions. FLIX instructions can be freely and modelessly intermixed in the processor's instruction stream along with the processor's native 24- and 16-bit instructions.
  - The Xtensa LX3 processor also features TIE ports, queues, and lookups, which allow designers to create high-speed I/O channels that communicate directly with TIE-based execution units and registers within the processor.
- Optional function units to fit each application

  - 16x16-bit and 32x32-bit multipliers
  - 16x16-bit MAC
  - Integer divider
  - CLAMPS, MIN/MAX, NSA, and Sign Extension

- Floating-point options

  Xtensa processors have two floating-point configuration options: a single-precision floating-point unit and a double-precision floating-point accelerator. One, or both of these floating-point configuration options can be selected.

- HiFi 2 Audio Engine with optional software audio codecs
- ConnX D2 DSP Engine with 16-bit DualMAC SIMD on a 2-way FLIX architecture
- ConnX Vectra LX DSP Engine with QuadMAC SIMD on a 3-way FLIX architecture
- Memory-Management Options
  - Region-based memory protection
  - Region-based memory protection with translation
  - Memory-management unit with Translation Lookaside Buffer (TLB) and Autorefill
- Configurable processor attributes to fit the application
  - Relocatable addresses for exception and reset vectors
  - Big- or little-endian byte ordering
  - 5- or 7-stage pipeline to accommodate different memory speeds

  - Exceptions: non-maskable interrupt (NMI), as many as 32 external interrupts, as many as six interrupt priority levels, and as many as three 32-bit timer interrupts
  - General-purpose 32-bit register file with 16, 32, or 64 entries
  - Write buffer: 1/2/4/8/16/32-entry write buffer
  - No write buffer (for configurations with no PIF)
  - Wide Instruction Fetch Width (64-bit) option

- Configurable Interfaces
  - 32/64/128-bit Processor Interface (PIF) width to main system memory or to an on-chip system bus. Tensilica provides a complete Vera-based tool kit for PIF bridge implementation and verification
  - Optional AHB-Lite and AXI bus bridges included with each processor instance
  - "No PIF" configuration option for gate-count reduction
  - Optional high-speed Xtensa Local Memory Interface (XLMI)
  - Inbound-PIF requests allow external access to the processors local memories (local instruction- and data-RAM memories and the XLMI port)
  - TIE ports, queues, and lookup interface ports

- On-Chip Memory Architecture
  - Optional 1, 2, 3, and 4-way set-associative caches
  - Write-through or write-back cache-write policy
  - Cache locking per line for set-associative cache

  - Size of instruction cache: 0/1/2/4/8/16/32/64/128 kbytes (for 1, 2, or 4-way set associative) 0/1.5/3/6/12/24/48/96 kbytes (for 3-way set associative)
  - Instruction cache line size: 16, 32, or 64 bytes
  - Size of data cache: 0/1/2/4/8/16/32/64/128 kbytes (for 1, 2, or 4-way set associative) 0/1.5/3/6/12/24/48/96 kbytes (for 3-way set associative)
  - Data cache line size: 16, 32, or 64 bytes
  - Optional data RAMs and ROM and optional instruction RAMs and ROM Maximum number of each type of memory: two data RAMs, two instruction RAMs, one data ROM, and one instruction ROM
  - Size of data RAM or ROM: 0/0.5/1/2/4/8/16/32/64/128/256/512 kbytes or 1, 2 or 4 Mbytes
  - Size of instruction RAM or ROM: 0/0.5/1/2/4/8/16/32/64/128/256/512 kbytes or 1, 2, or 4 Mbytes
  - One optional XLMI port, mapped to an address space of size: 0.5, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 kbytes or 1, 2, or 4 Mbytes.
  - Optional busy signals for all local memory ports and the XLMI port
  - Optional parity or ECC for local memory interfaces
  - Local memory and PIF widths independently configurable

- Multiple-processor (MP) development and debug capabilities
  - MP synchronization instructions (L32AI, S32RI, and S32C1I)
  - MP on-chip debug (OCD) capability: Trace and instruction/data breakpoint support (1 or 2 hardware-assisted instruction breakpoints and 1 or 2 hardwar-

eassisted data breakpoints). Support for synchronously stopping, stepping, and resuming execution on multiple processors.

  - Special processor ID (PRID) register
  - GDB or Xtensa Xplorer debugger support
  - Trace compressor with JTAG interface

- Software development tools (automatically generated)
  - High-performance Xtensa C/C++ compiler (XCC) and companion GNU tool chain

- Processor and system simulation environments
  - Cycle-accurate, pipeline-modeling instruction set simulator (ISS) with a fast functional simulator module called TurboXim available as a separate cost option
  - Xtensa Modeling Protocol (XTMP), a C-based API to the instruction set simulator
  - Xtensa SystemC (XTSC) package, which provides SystemC interfaces for both transaction-level and signal-level system modeling
  - XTSC-based SystemC-Verilog co-simulation, available as a separate cost option

- Robust EDA environment support
  - Standard and physical synthesis design flow

- Verification support
  - Diagnostics for the Xtensa core and designer-defined TIE verification

- Broad support for real-time operating systems
- System Integration Building Blocks
  - AHB-Lite and AXI bus bridges.
  - Xtensa Bus Designer's Toolkit (a.k.a., PIF Kit) for bus-bridge design.

5.2 XTMP Introduction

The stand-alone simulator allows you to simulate and verify the behavior of a single Xtensa processor connected to simple memories. However, the stand-alone ISS is not appropriate for designs that consist of multiple Xtensa cores, TIE ports, queues, and lookups, custom memories or other hardware devices.

For system simulation, Tensilica provides the ISS application programming interface (API), also referred to as the Xtensa Modeling Protocol (XTMP). XTMP allows you to write your own customized, multi-threaded simulators to model more complicated hardware systems. Because such a simulator is written in C, it runs much faster than an HDL simulator. Figure.5.2 shows the comparison of the simulation speeds when using FPGA, XTMP, CSM and RTL simulation. It is also much easier to modify and update than a hardware prototype. Thus, XTMP allows you to create, debug, profile, and verify the combined hardware and software systems early in the design process.

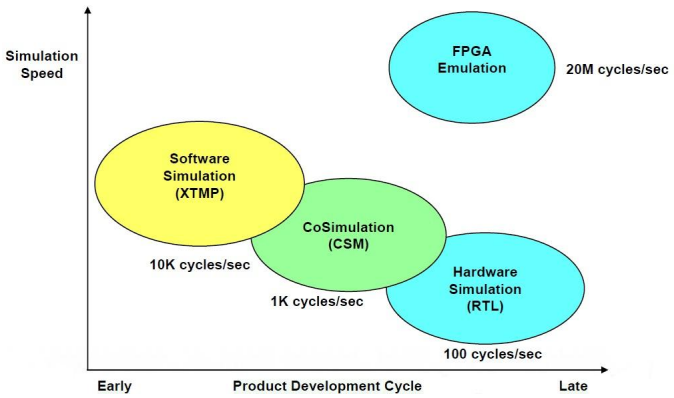

Figure 5.2: System Simulation Methods

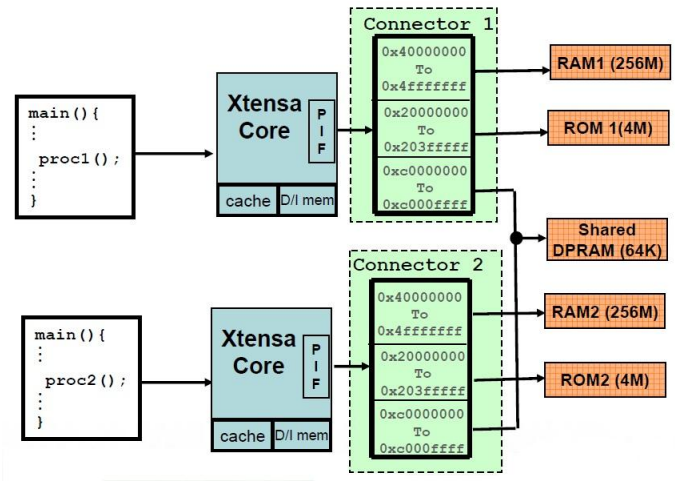

Figure 5.3: A Typical XTMP MP System

5.2.1 Basic XTMP Components and Connections

5.2.1.1 Simulation Clocks

Each XTMP simulation has a single global clock. To find out how many cycles have elapsed since the simulation started, people can use the following function: "XTMP_time XTMP_clockTime(void)". This is also what we used in the following sections for simulation time stamping.

In each simulation cycle, the global clock advances after all the cores finish the cycle and increment their local clocks.

5.2.1.2 TIE Ports, Queues and Lookups

There are several TIE language constructs that you can use for this purpose: state export, import wire, queue (input or output), and lookup. In XTMP, each of these TIE constructs is referred to as an XTMP_tieGroup. The TIE compiler translates each of these constructs into one or more interface signals: one signal for a state export or an import wire, multiple signals for a queue or a lookup. In XTMP, each of the TIE interface signals is referred to as an XTMP_tiePort.

5.2.1.3 Memory-Mapped Devices

A memory-mapped device refers to any component in your system that is accessed by loads or stores from an Xtensa processor. The XTMP device models use transaction-oriented interfaces. When a simulated Xtensa core issues a load or store request to a device attached to the PIF or a local port, the callback function associated with that device model is invoked. The device callback function receives the information about the transaction (transfer record), processes this information, and sends back a

response to the core.

Figure.5.3 shows a typical XTMP MP system consists of two cores with their own RAMs and ROMs which are connected to the core through a PIF connector. A DPRAM is shared by these two cores.

5.3 System Architecture

Our target in the simulation is to prove that the wavefront architecture can improve system performance by increasing the number of processors with small overhead. In this thesis, we will try to verify a quad-core system which can encoded real-time (25 FPS) videos with YUV420 CIF format, the system frequency is 150MHz.

5.3.1 HW/SW Partition

With a pure software solution, we found it may take more than 500000 clock cycles to encode a MB in I-Slice and more than 1500000 clock cycles to encode a MB in P-Slice (searching range = ± 16). However, The real-time requirement needs each MB to be encoded in $\sim$ 60000 clock cycles. It is obvious that pure software solution in a quad-core system can not meet the requirement. So, we will have to do HW/SW partition and implement hardware modules around each core to accelerate the encoding process.

In our implementation, we have two kinds of cores: "System Control Core (SC-Core)" and "Peripheral Core (P-Core)". SC-Core is in charge of task scheduling and interrupt service routing (ISR). P-Core is carrying on the real encoding task. When we talk about "core" alone, we referred to P-Core.

Figure.5.4 shows our proposed architecture for XTMP wavefront simulation.

Task scheduling is implemented by software running in SC-Core. Communications between SC-Core and P-Core are interrupt based.

Peripheral modules (P-Module) around each P-Core are used to accelerate the encoding tasks and are controlled by their master P-Core. Peripheral modules in our design include (1) intra_luma_4×4 module, (2) intra_luma_16×16 module, (3) intra_chroma module, (4) inter_prediction module, (5) post processing module and (6) stream packer module. In Figure.5.4, we gave a generic name to them: "PERI_A∼PERI_N".

To make it easier for data transfer and communication, we also implemented DMAC and INTC modules for both SC-Core and P-Cores. Softwares running at P-Cores are in charge of the following: (1) ISR, (2) coordinate the data transfers and communications among P-Core, shared memories and these P-Modules.

Each core, either SC-Core or P-Core, needs to use bus bridge to talk to their P-Modules and other shared resources. The interface between core and bus bridge is called "Processor Interface (PIF)" which uses a protocol defined by Tensilica.

5.3.2 Development Flow

The development of the XTMP simulation environment consists of two parts: hardware and software. These two parts are developed separately using different tools.

5.3.2.1 Hardware Development Flow

XTMP Modeling Protocol provides a model of Xtensa processors for software system simulation and corresponding hardware components. It also provides Application Programming Interface (API) to the ISS, available in library form.

The primary features of the hardware part includes: (1) Model and debug systems with one or more Xtensa processors and custom peripherals. (2) System simulation

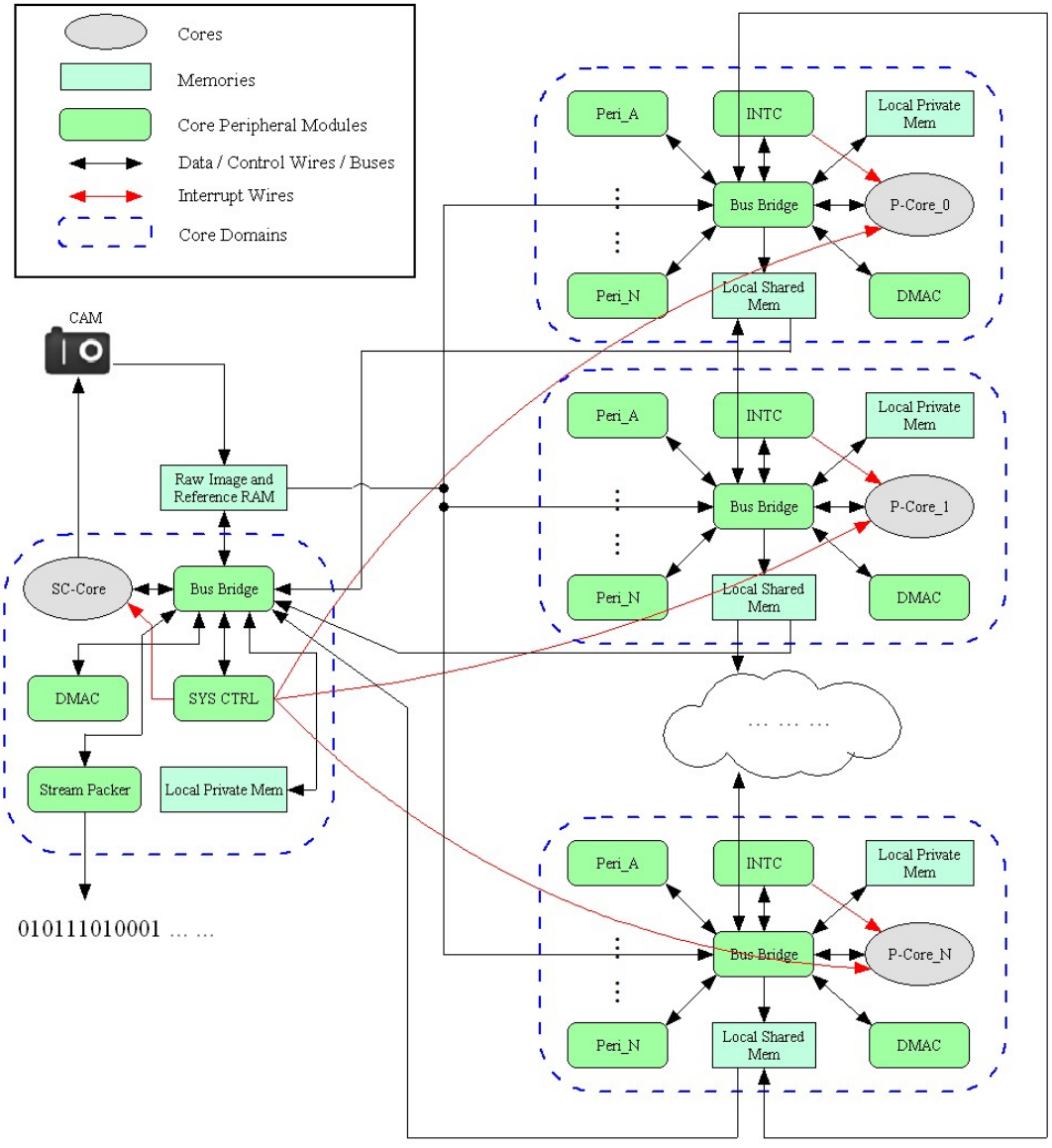

Figure 5.4: Architecture for XTMP Wavefront Simulation

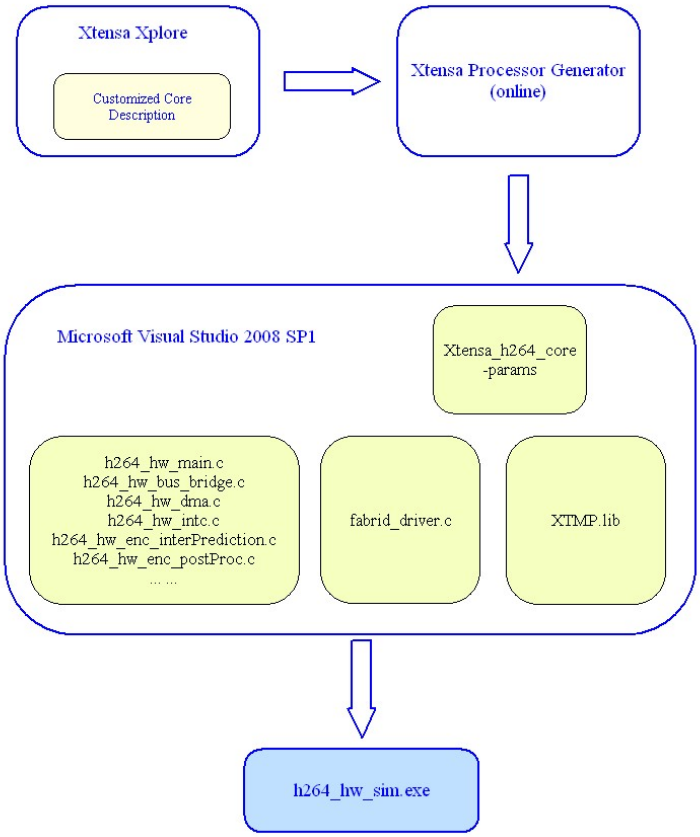

Figure 5.5: XTMP Hardware Development Components

is cycle accurate. (3) Provides a transactional level model of Xtensa.

Fig.5.5 shows the components and flows we used to build the simulation executable file "h264_hw_sim.exe":

- First, we used "Xtensa Xplore" (GUI for Xtensa System Development) to customize the processors, like the bus width, cache sizes, little/big endian, interrupt configuration etc. Primary configurations of our implementation is as Table.5.1.
- Then, we upload these configurations to Xtensa Processor Generator. It may take couple of hours to build the core. After that, we download a parameter file (in our case, named "Xtensa_h264_core-params") which will be used in the compilation.

Table 5.1: Primary Core Configurations

PROTOTYPE	PIF BITS	ENDIANESE	HAS_PRID
DC233L	32	LITTLE	YES
DCACHE _SIZE	DCACHE _WAYS	DCACHE _WRITE_BACK	DCACHE _LINE_BYTES
4096	2	YES	32
ROM _SIZE	LOCAL _RAM_SIZE	NUM_EXT _INTERRUPTS	WITH_MMU
64KB	4MB	17	YES

- The compilation and link tools are from "Microsoft Visual Studio 2008 SP1". "h264_hw_xxx.c" files described the system architecture and hardware functionalities of the system modules, like SC-Core and P-Cores, memories, bus bridges, DMAC, interrupt controllers and some other hardware acceleration blocks for the encoder, etc. "Xtensa_h264_core-params", "fiber-driver.c" and "XTMP.lib" are all XTMP related system files.

- The build target is "h264_hw_sim.exe". To run the simulation, we will need "h264_hw_sim.exe" work together with two software images built for SC-Core and P-Cores respectively. Software development flow will be discussed in next section.

5.3.2.2 Software Development Flow

The software development flow will generate two images with ELF format: "h264_sw_core_ctrl_main.out" and "h264_sw.out". The first image will be used for the SC-Core, the second will be used for P-Cores.

Figure.5.6 shows details of this flow. In this figure, LSP stands for "linker support package". A LSP specifies object files to pull into an executable using a specific memory map, and is used as a convenient short-hand for telling the linker what it

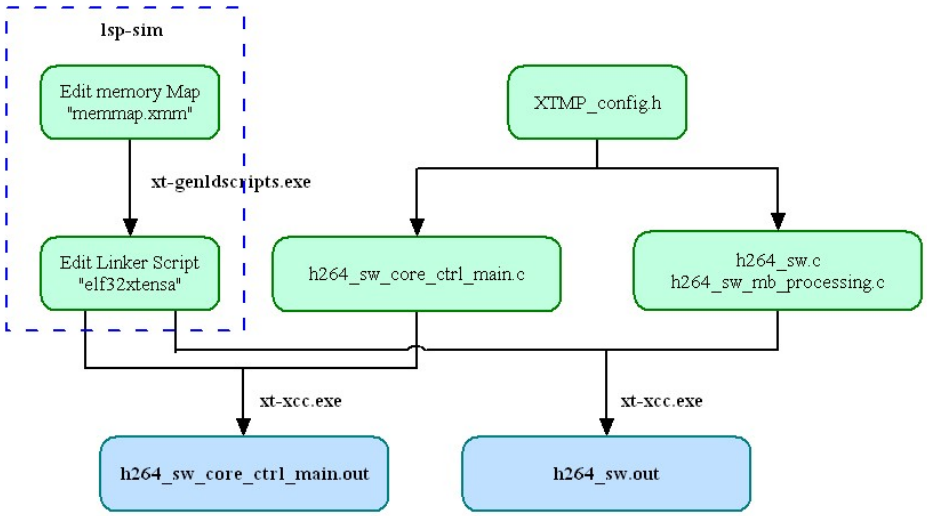

Figure 5.6: XTMP Software Development Components

needs for a particular target environment. "lsp-sim" is the default LSP when we do compilations.

After we defined the hardware system as previous section, we may need to modify the memory map "memmap.xmm" since we may have added new memory modules. "xt-genldscripts.exe" was used to convert "memmap.xmm" to a linker script file "elf32xtensa". In most applications, we may want to modify the properties of certain pieces of memories. For example, we may want to make shared memory regions to be unbufferable and uncacheable to avoid coherence issues. In those cases, we have to modify the linker script "elf32xtensa".

Among the source files, "XTMP_config.h" was generated by the Xtensa Processor Generator. It contains the configuration information of the generated cores. We also put our register definitions of all the P-Modules to this file. "h264_sw_core_ctrl_main.c" is the source file for SC-Core software, it implemented the task scheduling algorithms and ISR. "h264_sw.c" and "h264_sw_mb_processing.c" are the source files for P-Core software. They implemented the algorithms for ISR in P-Core and controls over all the

P-Modules around that P-Core.

Xtensa compiler "xt-xcc.exe" can do both the compiling and linking if you use it alone.

5.3.3 Components in System Architecture

As in Figure.5.4, the system consists of the following components. In this section, we will discuss the functionalities, register definitions of these components and how they are communicating with each other.

5.3.3.1 SC-Core and P-Cores

Our implementation used five cores for simulation: one SC-Core for the system control and task scheduling among the other four P-Cores for parallel encoding. When we are talking about quad-core system, it means four P-Core system. Both SC-Core and P-Cores are using the same configuration described in Table.5.1.

5.3.3.2 Bus Bridge

SC-Core and P-Cores communicate with their P-Modules through Processor Interfaces (PIF). In our configuration, each core only has one PIF and more than one P-Modules. So, we designed a "bus bridge" to solve this issue.

In our implementation, the bus bridge can also be called bus matrix. In XTMP, it analyzes the incoming transactions and forwards them to corresponding destinations.

By using the bus bridge, "Core-to-Peripherals" and "Peripherals-to-Peripherals" transactions can be processed in parallel, then makes DMA operations become possible.

In our implementation, each core has a bus bridge with it. The only case that a core can bypass the bus bridge is to access I/D-Caches.

5.3.3.3 Memories

In order to simplify the design and make debug easier, we used XTMP default memory type for all the memories in the system. In our implementation, both SC-Core and P-Cores have their own local private memories which can only be accessed by themselves. Also, we have two kinds of shared memories: (1) Shared memories between every two P-Cores, which can also be accessed by SC-Core but are invisible for other P-Cores. We define them as "Local Shared Memories". (2) Global shared memories, which can be accessed by all the cores (both SC-Core and P-Core). We define them as "Global Shard Memories".

- **Local Private Memories**

  Each core has a local memory which is used to store software images for that core, stack/heap and some temporary data.

  In order to improve the core performance, local private memory is cacheable and bufferable by default. That means, the I/D-Caches of the corresponding core will keep a small copy of the frequently used data in the caches and save memory access time.

  With the bus bridge, the local private memory is also accessible by some of the P-Modules, like DMAC. Sometimes, we may need to use DMAC to transfer data from local private memory to other P-Modules. However, if the data is still in the caches, it may cause a cache coherence issue, which means, the data in local memory is not the latest. In order to solve this issue, we utilized one of the Xtensa's Linker Support Packages (LSP) called "lsp-sim". By modifying the linker script in LSP, we made a part of the local private memory to be non-cacheable and non-bufferable. All the shared data between cores and their P-Modules is put in these

regions, cache coherence issues are avoided.

- **Local Shared Memories**

  Local shared memories are used for data exchange between two P-Cores. Every two neighboring P-Cores have such a memory. This kind of memories are not accessible by other P-Cores, but visible for SC-Core. They are configured as non-cacheable and non-bufferable in the linker scripts mentioned above.

- **Global Shared Memories**

  An example is the "Raw image and reference memory" in Figure.5.4, which is used to store raw images generated by the camera module (raw image buffer) and reference frames (reference buffer). In order to improve the system throughput, a ping-pong mechanism is adopted for the raw image buffer. We split this buffer into two parts: a "Even frame buffer" and a "Odd frame buffer". When raw images with even frame index are being processed, camera module fetches next frame with odd frame index and vice-versa. By doing this, the time spent on raw image fetching is hidden. So, for the encoder, whenever the raw images are needed, they are already there.

5.3.3.4 Camera Module

Camera module is controlled by the SC-Core. After sending out a frame, it has to be re-programmed. As shown in Table.5.2, before a frame starts, the SC-Core needs to program the desired image width/height, frame_idx. Since ping-pong buffer mechanism is used, we also need to let the camera module know about the output addresses for Y/U/V components.

Camera module itself has an output engine. So, once the CTRL.BIT0 is set,

Table 5.2: Register Definitions of Camera Module

Addr Offset	Register Name	Description
0x00	CTRL	Camera Control Register BIT0: Camera starts BIT1: Camera reset
0x04	WIDTH	Image Width Register
0x08	HEIGHT	Image Height Register
0x0C	FRM_IDX	Image Index Register
0x10	Y_ADDR	Y Component Address Register
0x14	U_ADDR	U Component Address Register
0x18	V_ADDR	V Component Address Register
0x1C	OUT_CNT	Output Frame Count Register
0x20	STATUS	Status Register BIT0: Busy BIT1: Ack

camera module will start to send raw image data to designated memory addresses without the interference of DMAC. Once a frame was sent, STATUS.BIT1 will be set to 1, reset will clear it.

In our simulation, the camera module will read from a raw data file based on the requirement of the SC-Core.

5.3.3.5 PIF System Control Unit (SCU)

The synchronization between SC-Core and other P-Cores are interrupt based. SCU was designed to handle interrupts and some of the information exchange between SC-Core and P-Cores. Interrupts between a core and its own P-Modules are handled by corresponding INTCs.

Figure.5.7 illustrates the process for the interrupt based system control flow in our design. Figure.5.7(A) stands for the flow in SC-Core side and Figure.5.7(B) stands for the flow in the P-Core side. Table.5.3 shows the register definitions of SCU.

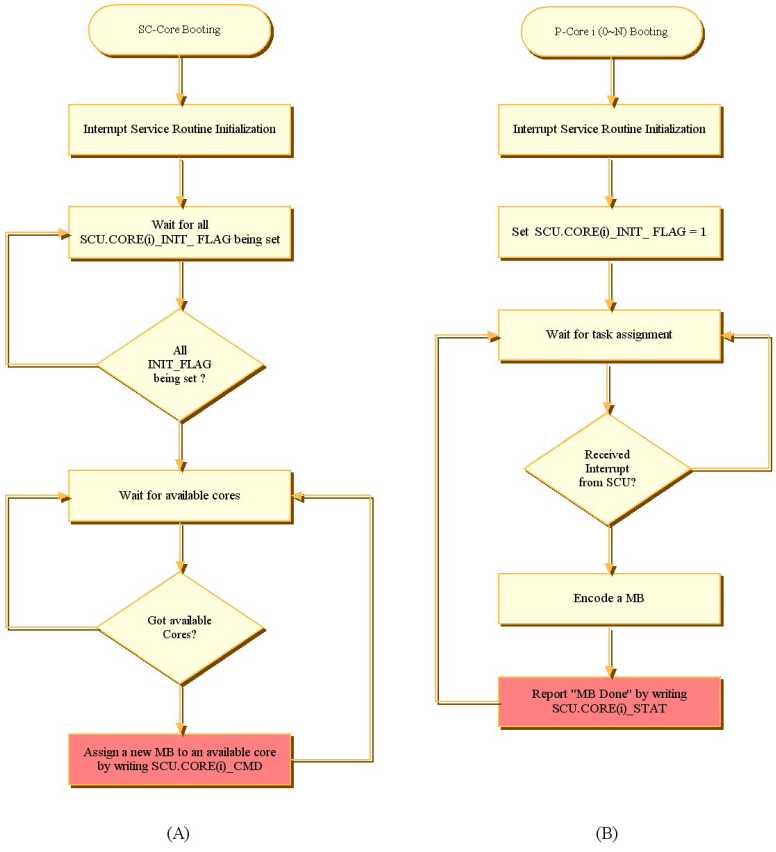

Figure 5.7: Interrupt Based System Control Flow

In Figure.5.7, red blocks stand for an action which may cause SCU to assert an interrupt. For example, writing SCU.CORE0_CMD register will make SCU assert an interrupt to CORE0. In the interrupt service routine (ISR), CORE0 knows that a new task has been assigned and then read register SCU.CORE0_CMD to get the new task information, like "frm_idx", "mb_y" and "mb_x", etc. Similarly, once CORE0 finished encoding a MB, it will write SCU.CORE0_STAT register and make SCU assert an interrupt to SC-Core. After receiving this interrupt, the SC-Core will check SCU.INT_STAT register to see which bit has been set (each bit represents a specific P-Core) and then mark corresponding core to be "available". If all conditions are met, a new task will be assigned to that available P-Core very soon.

In our system design, each core only has one interrupt port being used. If multiple P-Cores send interrupts to SC-Core through SCU in a very short time window, we need to make sure every interrupt gets processed. For this purpose, we implemented a counter named "irq_cnt" in the SCU design. Whenever a P-Core writes the corresponding SCU CORE STAT register, this counter will increment by 1 and whenever the SC-Core writes any SCU CORE CMD register, this counter will decrease by 1. Only if $irq\_cnt > 0$, the interrupt signal line between the SC-Core and SCU will remain high. When $irq\_cnt = 0$, the interrupt signal line will be pulled low and ISR in SC-Core software will not be called any more.

As we can see in Table.5.3, SCU is also used to deliver some side information from SC-Core to P-Cores, such as raw image buffer addresses and reference buffer addresses.

Table 5.3: Register Definitions of PIF SCU

Addr Offset	Register Name	Description
0x00	INT_STAT	Interrupt Status Register BIT0: 1 - core0 interrupt asserted BIT1: 1 - core1 interrupt asserted BIT2: 1 - core2 interrupt asserted BIT3: 1 - core3 interrupt asserted
0x04	INT_CLR	Interrupt Clear Register BIT0: 1 - clear Bit0 in INT_STAT BIT1: 1 - clear Bit1 in INT_STAT BIT2: 1 - clear Bit2 in INT_STAT BIT3: 1 - clear Bit3 in INT_STAT
0x08	CORE0_CMD	Core0 Command Register BIT0: mb_start_flag BIT1: mb_type BIT2-7: rsvd BIT8-15: frm_idx BIT16-23: mb_y BIT24-32: mb_x
0x0C	CORE0_STAT	Core0 Status Register BIT0: mb_done BIT1: mb_type BIT2: in_sync BIT3-7: rsvd BIT8-15: frm_idx BIT16-23: mb_y BIT24-32: mb_x
0x10	CORE1_CMD	Core1 Command Register
0x14	CORE1_STAT	Core1 Status Register
0x18	CORE2_CMD	Core2 Command Register
0x1C	CORE2_STAT	Core2 Status Register
0x20	CORE3_CMD	Core3 Command Register
0x24	CORE3_STAT	Core3 Status Register
0x28	CFG	Configuration Register BIT0: reset
0x2C	CORE0_INIT_FLAG	Core0 initialization finish flag
0x30	CORE1_INIT_FLAG	Core1 initialization finish flag
0x34	CORE2_INIT_FLAG	Core2 initialization finish flag
0x38	CORE3_INIT_FLAG	Core3 initialization finish flag
0x3C	BUF_EVEN_ADDR	Address for even frames in the raw image ping-pong buffer.
0x40	BUF_ODD_ADDR	Address for odd frames in the raw image ping-pong buffer.
0x44	REF_Y_ADDR	Address for the Y component in the reference buffer.
0x48	REF_UV_ADDR	Address for the UV components in the reference buffer.

5.3.3.6 Direct Memory Access Controller (DMAC)

Each cores in the system has its own DMAC. DMACs are used to transfer data between their master cores and other P-Modules. Single and burst transfers are both supported in our system, the bus width is 32-bit.

DMACs also support linked transfers. For that purpose, a table for Linked List Items (LLI) needs to be used. Each item consists of 4 parts: (1) Source Address, (2) Destination Address, (3) Next LLI Address, (4) Transfer Size. For the last transfer, "Next LLI Address " has to be "0". Due to the possible cache coherence issue, we put all LLIs in non-cacheable and non-bufferable regions in the local private memory.

Table.5.4 shows the register definitions of DMAC. The finish of a transfer can be checked in two ways: (1) Once the transfer finished, DMAC will inform INTC module by writing a 1 to INTC_STATUS.BIT1 in Table.5.5. Then INTC will assert an interrupt to its master core, (2) Master core can check DMAC.STATUS.BIT1, which is the acknowledge bit of the DMA transfer.

5.3.3.7 Interrupt Controller (INTC)

INTC is a very important module to free its master core from keep polling the status of a P-Module. When a module wants to inform its master core by interrupt, it only needs to set the corresponding bit in INTC.STATUS register, then INTC will assert an interrupt to the master core. ISR in the the master core will, in turn, check INTC.STATUS to figure out the interrupt source. As in Table.5.5, writing to INTC.CLR will clear desired interrupt bit.

Table 5.4: Register Definition of DMAC

Addr Offset	Register Name	Description
0x00	CTRL	DMAC Control Register BIT0: start DMA BIT1: reset DMA BIT2: set timer
0x04	IN_ADDR	Source Memory Address
0x08	OUT_ADDR	Destination Memory Address
0x0C	TFR_SIZE	Transfer Size Register
0x10	TFR_CNT	Transfer Count Register
0x14	STATUS	Status Register BIT0: DMA Busy BIT1: DMA Ack
0x18	TIMER_H	System Timer High 32-bit
0x1C	TIMER_L	System Timer Low 32-bit
0x20	LLI_ADDR	Address for next LLI Item
0x24	PRIORITY	Transaction Priority Register

Table 5.5: Register Definition of INTC

Addr Offset	Register Name	Description
0x00	CTRL	INTC Control Register BIT0: INTC Enable BIT1: INTC Reset
0x04	STATUS	INTC Status Register BIT0: CAM IRQ BIT1: DMAC IRQ BIT2: ENC_POST BIT3: ENC_INTRA_LUMA_4x4 BIT4: ENC_INTRA_LUMA_16x16 BIT5: ENC_INTRA_CHROMA BIT6: ENC_INTER_PRED BIT7: ENC_STREAM_PACKER
0x08	CLR	INTC Clear Register Clear corresponding bit in status register

5.3.3.8 Intra_Luma_4 × 4 Module

This module is one of the P-Modules of the P-Core. It does the intra_4 × 4 prediction and output the optimal intra_luma_4×4 mode, the coded block pattern (CBP) and the prediction cost which will be used to compare with the cost in intra_luma_16×16 prediction.

In our implementation, there is an internal memory inside this module, which is used to store the current MB data and the information of neighboring MBs. Before this module starts, all these data needs to be transferred from outside to this internal memory.

The register definitions are as Table.5.6. After the reset (CTRL.BIT1), intra_luma_4×4 module will put the addresses for current MB and neighboring MBs in internal memory to register ADDR_CUR_MB, ADDR_MB_A, ADDR_MB_B, ADDR_MB_C and ADDR_MB_D respectively. Here, MB_A, MB_B, MB_C and MB_D refer to the neighboring MBs in the same frame.

Its master P-Core then load related data to these addresses using DMAC and set CTRL.BIT0 to start the processing. Once the task finished, the master P-Core will read out the value in COST register and compare it with the cost from intra_luma_16×16 module. For the winner (with the smaller cost), its CBP and intermediate encoding information will be copied out to core local private memory from ADDR_CUR_MB. CBP is a syntax element which will be encoded in the CAVLC stage.

5.3.3.9 Intra_Luma_16 × 16 Module

The design and register definitions of intra_luma_16 × 16 module are similar with intra_luma_4 × 4. These two modules are running in parallel. When both of these

Table 5.6: Register Definitions of INTRA_4 × 4 Module

Addr Offset	Register Name	Description
0x00	CTRL	Control Register BIT0: Start BIT1: Reset
0x04	ADDR_CUR_MB	Loading Address of Current MB
0x08	ADDR_MB_A	Loading Address of MB_A
0x0C	ADDR_MB_B	Loading Address of MB_B
0x10	ADDR_MB_C	Loading Address of MB_C
0x14	ADDR_MB_D	Loading Address of MB_D
0x18	STATUS	Status Register BIT0: Busy BIT1: Ack
0x1C	CBP_LUMA	Coded Block Pattern Register
0x20	COST	Cost Register

two modules finished, their encoding costs will be compared, the P-Core will pick the smaller one and the corresponding encoding information will be copied out from the internal memory of that module.

5.3.3.10 Intra_Chroma Module

Intra_Chroma module starts to run after two intra_luma modules finished their tasks since intra_chroma module needs to write Coef_AC and Coef_DC data structures which have to be updated by intra_luma modules first. The design and register definitions of intra_chroma module are similar with intra_luma_4 × 4 as well.

The modeling of these three intra modules referred to the design in [47].

5.3.3.11 Inter-Prediction Module

Tasks conducted by the inter-prediction module include IME, FME, DCT, IDCT, quantization and inverse-quantization. Deblock filtering and entropy coding will be in Post-Processing (PP) module discussed in next section.

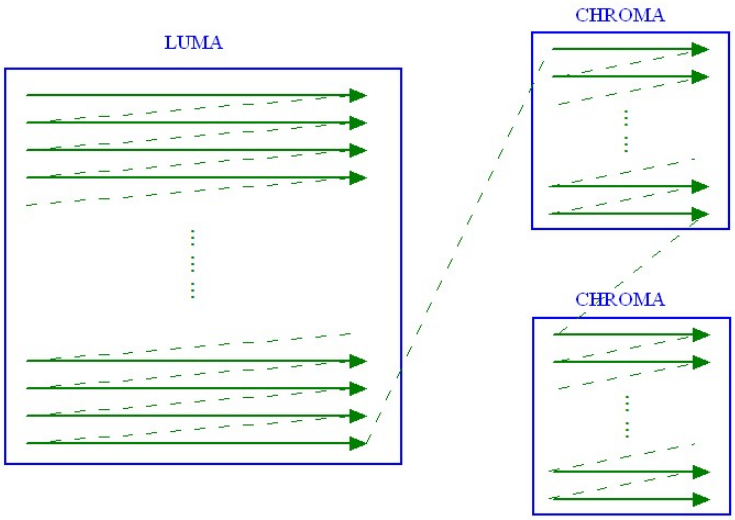

Figure 5.8: Reference Pixels Scan Order

The register definitions of inter-prediction module are as Table.5.7. Similar with intra-prediction modules, master P-Core will first use DMAC to transfer current MB data and information of neighboring MBs to the internal memory of this module. Also, the reference data will transferred from reference buffer in the global shared memory. Different from traditional memory data layouts, we read/write the reference buffer MB-by-MB and inside each MB, the pixels are scanned in a order described in Figure.5.8. This way, DMAC can do a continuous transfer of all the data in a MB without re-programming.

The best inter-prediction mode and some intermediate encoding results, like AC/DC coefficients will be read back from ADDR_CUR_MB. Coded block patterns (CBP) for both Luma and Chroma components can be accessed from STATUS register. The value in COST register is used by the master P-Core to decide whether skip mode should be used or not.

Table 5.7: Register Definitions of Inter-Prediction Module

Addr Offset	Register Name	Description
0x00	CTRL	Control Register BIT0: Start BIT1: Reset
0x04	ADDR_CUR_MB	Loading Address of Current MB
0x08	ADDR_MB_A	Loading Address of MB_A
0x0C	ADDR_MB_B	Loading Address of MB_B
0x10	ADDR_MB_C	Loading Address of MB_C
0x14	ADDR_MB_D	Loading Address of MB_D
0x18	ADDR_REF	Loading Address of Reference Pixels
0x1C	STATUS	Status Register BIT0: Busy BIT1: Ack BIT2-7: RSVD BIT8-15: CBP_LUMA BIT16-23: CBP_CHROMA
0x20	COST	Cost Register

5.3.3.12 Post Processor (PP)

The Post Processor module conducted two tasks: (1) deblock-filter and (2) Entropy Coding. These two tasks are processed in parallel.

As we know, for deblock-filter and entropy coding, except the current MB itself, the information about its neighboring MBs: MB_A, MB_B, MB_C and MB_D may also be needed. After reset, the default values of register PP.ADDR0 $\sim$ PP.ADDR4 will be the PP internal memory addresses for current MB, MB_A, MB_B, MB_C and MB_D.

The software in the master P-Core will read the addresses from PP.ADDR0 to PP.ADDR4 and use DMAC to transfer current MB and its neighbors to corresponding memory locations inside PP module. Setting CTRL.BIT0 will trigger PP module to start. Then the P-Core need to either wait for interrupt requested by PP to happen, or keep polling PP.STATUS.ACK to see if it finishes or not.

Even though in [1], deblock filtering is conducted in a MB-by-MB order, intra-

prediction for the whole frame has to be finished before that. The reason is, intra-prediction needs non-filtered neighboring pixels and deblock filtering will obviously modify these data.

However, for mobile/embedded applications, the data flow in JM may not be applicable since we don't have enough memory space to hold the non-filtered pixels for that long time. Especially, in our proposed wavefront architecture, the encoding of next frame may need the filtered pixels of current frame even before the encoding process of current frame completely finished. So, we have to find a way to make deblock filtering really MB-based, which means, deblock filtering doesn't have to wait till all the MBs in current frame finishes their intra-prediction. It should be able to start when the intra-prediction of current MB finishes.

In Figure.5.9, the red area denotes the pixels which will be needed by the intra-prediction of current MB. These pixels should be non-filtered reconstructed pixels. Also, the shading area contains the pixels which may be used by the deblock-filtering process of current MB, they may be partially filtered pixels. In order to solve this conflict, we saved these non-filtered pixels as part of the neighboring MB information.

For both MB boundaries and internal edges, we do horizontal filtering first and then vertical filtering.

For the modeling of deblock filter itself, we referred to the design in [46]. After PP finished, master P-Core needs to use DMAC to transfer deblock-filtered results from PP internal memory to core local private memory.

The entropy coding results of this module will be transferred to local shared memory. This will be discussed in section 5.3.6.

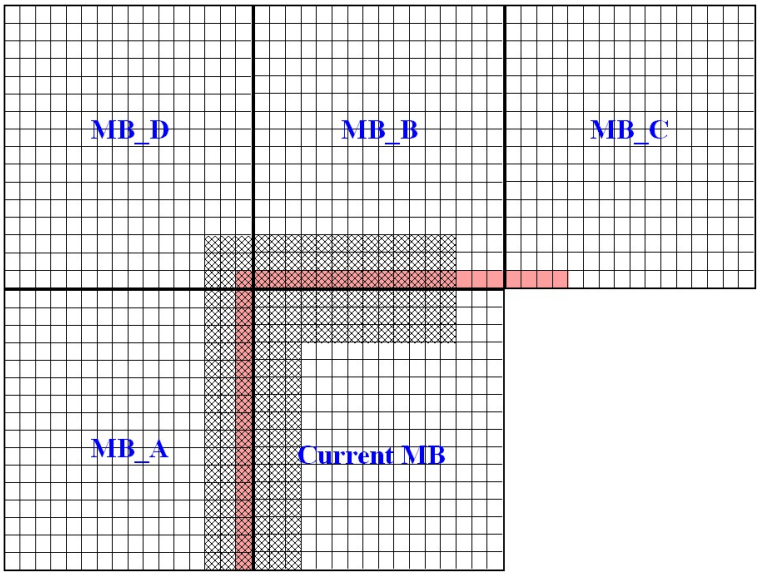

Figure 5.9: Neighboring MBs in Deblock-filtering

Table 5.8: Register Definition of PP

Addr Offset	Register Name	Description
0x00	CTRL	PP Control Register BIT0: PP Start BIT1: PP Reset
0x04	ADDR0	Current_MB Address in module internal Memory
0x08	ADDR1	MB_A Address in PP internal Memory
0x0C	ADDR2	MB_B Address in PP internal Memory
0x10	ADDR3	MB_C Address in PP internal Memory
0x14	ADDR4	MB_D Address in PP internal Memory
0x18	STATUS	PP Status Register BIT0: Busy BIT1: Ack

5.3.3.13 Stream Packer (SP)

In the standard [14] and JM model [1], MBs are processed in a raster-scan order, so does the entropy coding.

However, our proposed wavefront architecture is based on the availability, that means, the raster-scan order has been broken and we need a module to assemble these out of order MB coding results.

In our proposed architecture, SP gets the entropy coding results from the local shared memory of multiple P-Cores through the DMAC engine of the SC-Core and assemble them together. The output order of MBs from SP is identical with JM.

5.3.4 Task Scheduling Algorithm Design

Task scheduling is handled by the software running in the SC-Core. Figure.5.10 shows the flow for task scheduling.

5.3.5 Communications between Cores

The data transfer direction is show in Figure.5.11 and Figure.5.12. Each segment of the memory stores the necessary coding information of a MB since the encoding of its neighboring MBs may need it later. Its content is as Table.5.9. The size of local shared memory is:

$$local\_shared\_mem\_size = frame\_width\_in\_mb \times segment\_size \tag{5.1}$$

As in Figure.5.12, the read/write to the local shared memory by P-Core(i) and P-Core(i+1) are coordinated by the task scheduling algorithm in SC-Core. The

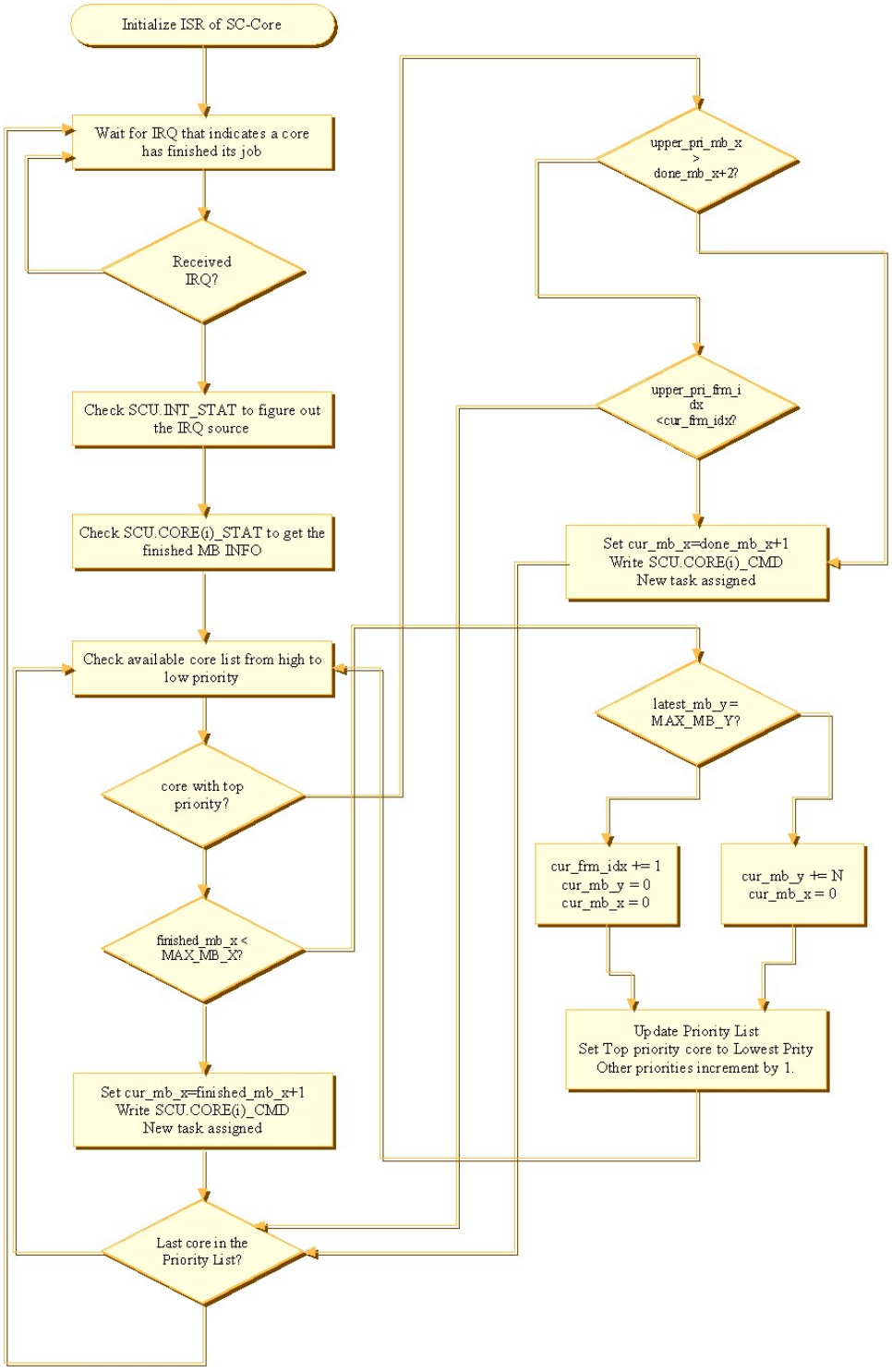

Figure 5.10: Task scheduling flow

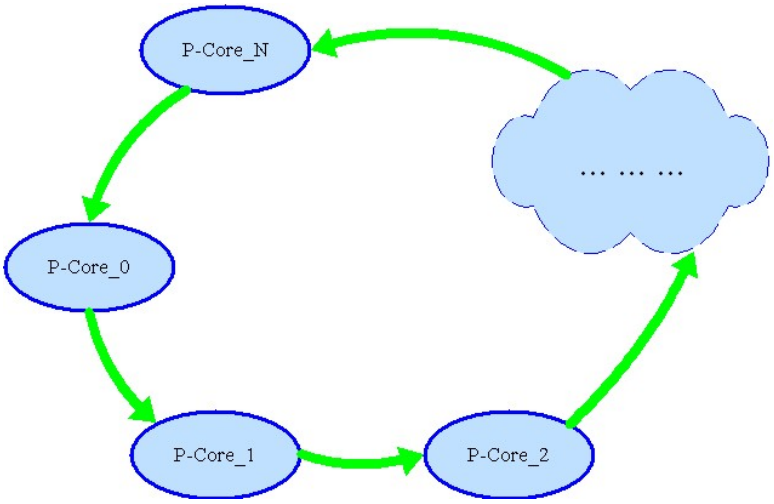

Figure 5.11: Inter-Core communication ring

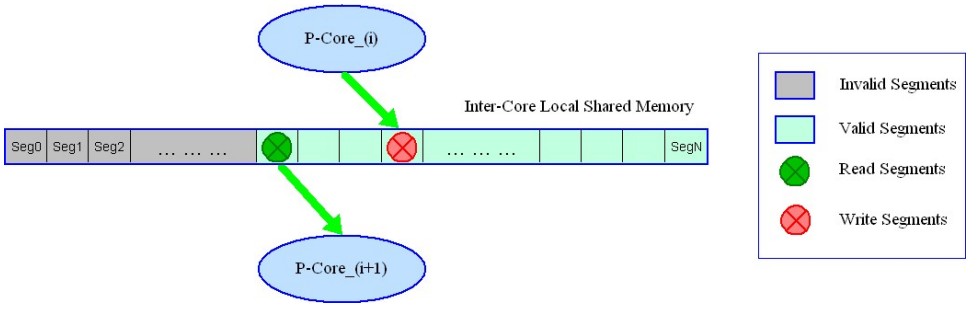

Figure 5.12: Shared memory between two peripheral cores

Table 5.9: Inter-Core Shared Information (YUV420)

Item	Size	Description
REC_Y	256 Bytes	Reconstructed Y component after deblock filtering
REC_U	64 Bytes	Reconstructed U component after deblock filtering
REC_V	64 Bytes	Reconstructed V component after deblock filtering
BK_Y	31 Bytes	Reconstructed right and bottom edge pixels before deblock filtering. This may be used for the intra-prediction of neighboring MBs.
BK_U	15 Bytes	see "BK_Y"
BK_V	15 Bytes	see "BK_Y"
NZ_COEF_LUMA	16 Bytes	Number of non-zero coefficients of each 4×4 Luma block.
NZ_COEF_CHROMA	8 Btyes	Number of non-zero coefficients of each 4×4 Chroma block.
MB_Y	1 Byte	Current MB coordinate
MB_X	1 Byte	Current MB coordinate
CBP	2 Byte	"Coded Block Pattern", will be used for deblock filter.
DIRTY	1 Byte	This item indicates if this memory segment can be overwritten or not.
BEST_MODE_MV	16 Bytes	Motion vectors for the best mode. indexed by [b8_y][b8_x][y/x=0/1]
BEST_INTRA4x4 _MODE	16 Bytes	
BEST_INTRA16x16 _MODE	1 Byte	
BEST_INTRA_CHROMA _MODE	1 Byte	
BEST_INTRA_MODE	1 Byte	
BEST_INTER_MODE	1 Byte	
SLICE_TYPE	1 Byte	

read and write locations never overlap. So the software running in both P-Core(i) and P-Core(i+1) don't need to handle any data synchronization issue.

For a video sequence with CIF (352×288) format, $frame\_width\_in\_mb = 22$, $segment\_size = 512$ (Bytes), so the overall size of $shared\_mem\_size = 11264$ Bytes (11 KBytes). Also, the traffic to access this memory will not increase with the increase of the number of P-Cores.

5.3.6 Data Reuse and Memory Savings

5.3.6.1 Reference Buffer Saving

For basic applications, H.264/AVC encoder needs at least one reference frame for inter-prediction of P/B frames. For single processor/thread solution, we may have to allocate dual-buffers for both the current reference frame and the reconstructed pixels of current frame (will be used as reference frame for next frame), as in Figure.5.13(A). In proposed wavefront architecture, we can merge these two buffers into a single one, as in Figure.5.13(B).

In our design, after encoding a MB, the reconstructed pixels (Y/U/V) will not be saved back to the reconstruction buffer right away. It stays in the local shared memory until that memory location is overwritten and the corresponding P-Core saves it back to the reference buffer. The reason is two-folded:

- The co-located MB in the reference buffer will still be needed by neighboring MBs, as in Figure.5.14. So, it cannot be overwritten right away and has to be in the reference buffer for a while till all MBs who need it for inter-prediction have been encoded.

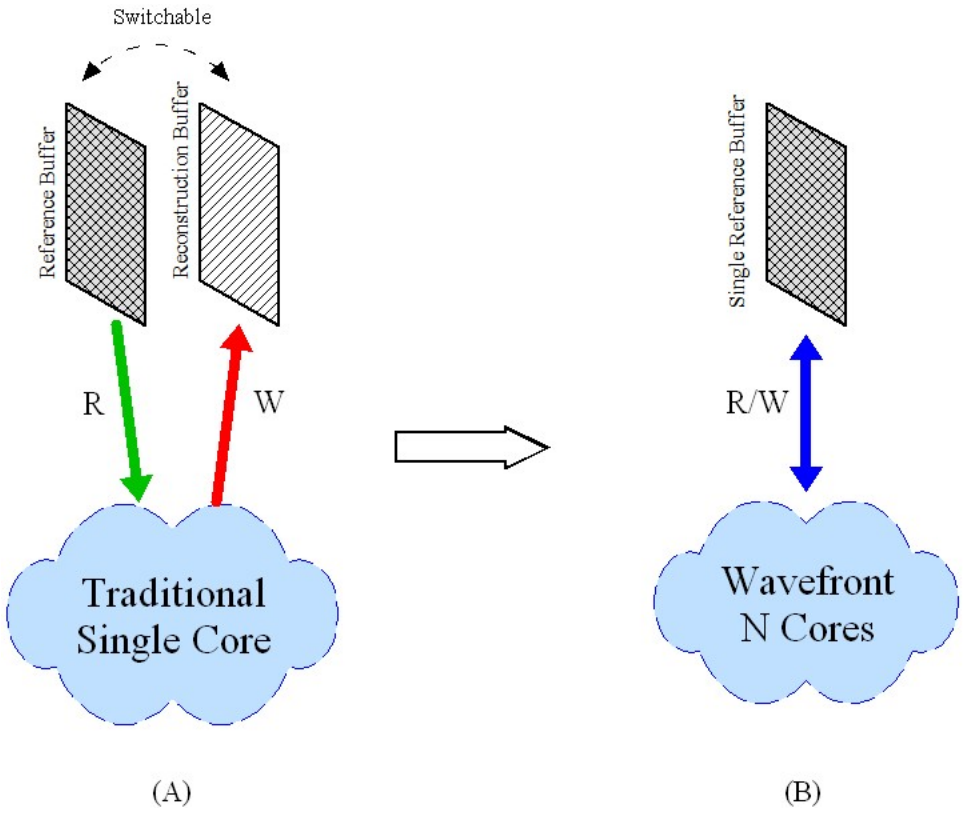

Figure 5.13: Reference Buffer Saving

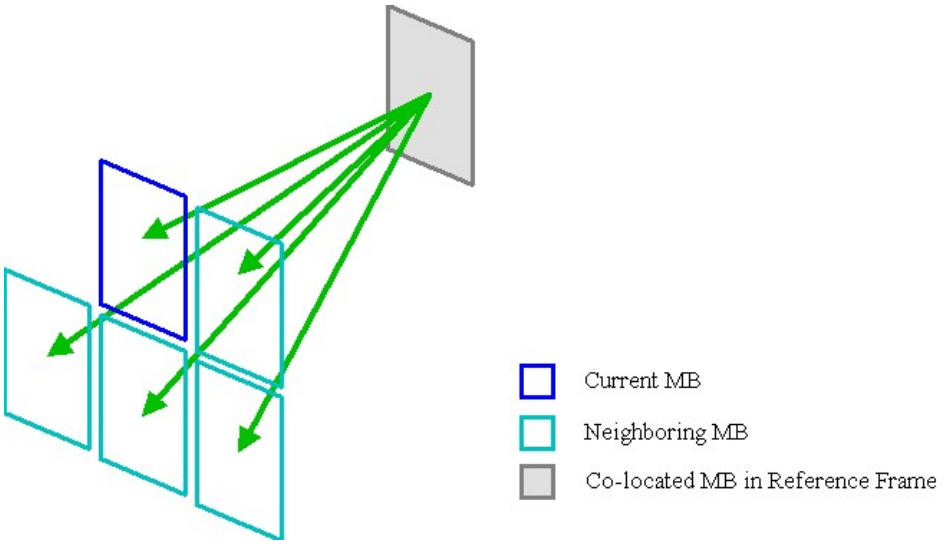

Figure 5.14: Co-located MB in the reference frame

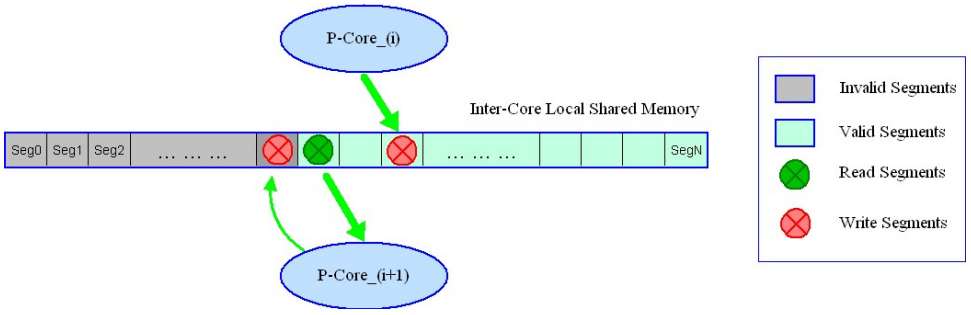

Figure 5.15: Local Shared Memory Saving

- Not all the edges in current reconstructed MB have been processed by the deblock filter, as in Figure.5.9. Pixels in right and bottom edges may be modified when we do deblock filtering for its neighboring MBs.

Assume we have N P-Cores which are encoding continuous MB lines in parallel. Here, we define "MB line" as a row of MBs. For example, if MB(x,y) has been encoded, the P-Core will empty segment x in the shared memory first and then put the reconstructed MB(x, y) into that location. Before the P-Core empties segment x, what's kept in there is the reconstructed MB(x, y-N) and some side encoding information. Also all of the edges in that MB have been deblock filtered. What's more, the co-located MB(x, y-N) in the reference frame will not be needed any more, so it's safe to be overwritten by MB(x, y-N) in current frame.

By doing so, we saved the bandwidth to access reference frame buffer.

5.3.6.2 Local Shared Memory Saving

In order to utilize these "invalid" segments in local shared memory, as in Figure.5.15, we allow P-Core(i+1) to put its entropy coding results in the invalid segments of local shared memory between P-Core(i) and P-Core(i+1). These data will be transferred to stream packer (SP) before P-Core(i) switches to other MB lines.

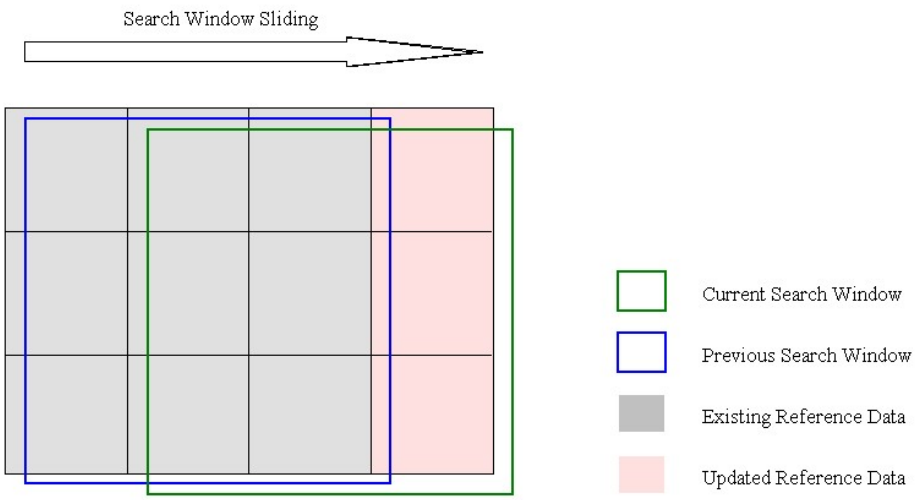

Figure 5.16: Sliding Window for Reference Data

5.3.6.3 Reference Data Reuse in Inter_Prediction Module

Reference data is a key part of inter-prediction. In order to save bandwidth to access the reference buffer in global shared memory, we used a sliding window to load reference data and only the updated data will be loaded to the internal memory of Inter_Prediction module, as in Figure.5.16.

5.3.7 Load Balancing

As we will see in next section, the fluctuation for the MB processing time in the neighboring area is not large and most of the time are spent on the memory access / data transferring.

To do the load balancing among different cores, the data which needs to transfer from one core to another includes:

- **Current MB Data:** Raw image data, AC/DC Coefficients and some intermediate data.

- **Reference Data:** This is for inter-prediction only. In our simulation, the reference data needs to be loaded varies from 5 to 15 macroblocks (1.9KB $\sim$ 5.6KB in size).

- **Neighboring Data:** For intra-prediction, we need neighboring pixels in current frame before deblock-filtering. For joint deblock-filtering with neighboring MBs, we need neighboring pixels in current frame after partial (horizontal or vertical, but not both) deblock-filtering. Also, we will need some of the neighboring encoding results (best modes, number of non-zero coefficients, motion vectors, LUTs used in entropy coding, etc).

Transferring these data will easily cost tens of thousands of cycles, which is far more than the difference of the processing time of neighboring MBs. So, introducing load balancing to our proposed wavefront architecture may highly deteriorate the system performance and cause even longer delays.

Based on this fact, we choose not to use load balance in our implementation.

5.4 Simulation Results

The wavefront idea we proposed in Chapter 4 was simulated using the architecture and tools discussed in this chapter. In order to evaluate the performance improvements, besides the quad-core system, we also simulated a dual-core and a three-core wavefront system with the same software and P-Modules. At the the end of this section we also discussed the upscaling schemes to expand the system to any number of cores.

During the simulation, we used "time stamps" (with a unit of simulation clock cycles) to monitor system performance / behavior. After SCU received a CMD from

SC-Core or after it received a STAT write from P-Cores, it will print a time stamp, together with corresponding MB information, to a simulation log file.

Time stamp is defined as:

$$TIME\_STAMP[core\_idx][frm\_idx][mb\_y][mb\_x][scu\_flag] \tag{5.2}$$

where $scu\_flag = 0$ denotes the start of a MB processing and $scu\_flag = 1$ denotes the end of a MB processing. Besides time stamp, we have other two important definitions: "MB Processing Gap" (MPG) and "MB Processing Time" (MPT).

$$\begin{aligned} &MB\_PROC\_GAP[frm\_idx][mb\_y][mb\_x] = \\ &TIME\_STAMP[core\_idx][frm\_idx][mb\_y][mb\_x][0]- \\ &TIME\_STAMP[core\_idx][frm\_idx][mb\_y][mb\_x-1][1] \end{aligned} \tag{5.3}$$

$$\begin{aligned} &MB\_PROC\_TIME[frm\_idx][mb\_y][mb\_x] = \\ &TIME\_STAMP[core\_idx][frm\_idx][mb\_y][mb\_x][1]- \\ &TIME\_STAMP[core\_idx][frm\_idx][mb\_y][mb\_x][0] \end{aligned} \tag{5.4}$$

An example of these two concepts can be found in Figure.5.17.

5.4.1 MB Processing Gaps

As in Figure.5.17, MB Processing Gap is a very important index to show how fast a P-Core can start its next job once it finished encoding current MB. Factors which may impact it include:

- Task scheduling algorithm implemented in the SC-Core.

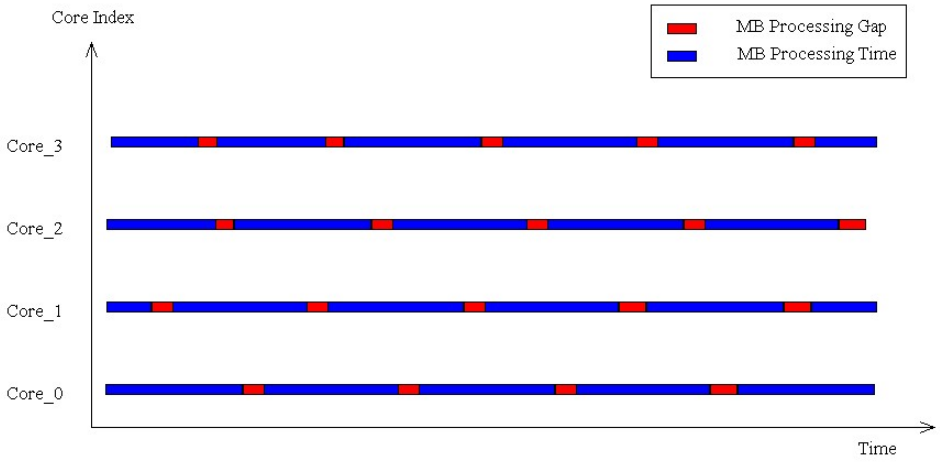

Figure 5.17: MB Processing Time and Gaps

Table 5.10: Average MB Processing Gaps (CIF YUV420)

	Dual-Core		Three-Core		Quad-Core	
	# Cycles	Δ%	# Cycles	Δ%	# Cycles	Δ%
Intra	435	0%	489	12.4%	542	24.6%
Inter	440	0%	484	10.0%	519	18.0%

- Whether the relied neighboring MB information in upper MB lines become available.

Figure.5.18∼5.23 show the MB processing gaps for a dual-core, three-core and a quad-core system with intra/inter-prediction. With the increase of the number of P-Cores, there is a small increase in the MB processing gap, as in Table.5.10. The Δ% is caused by the increased number of cores that the control software (running on the SC-Core) needs to screen and assign tasks. Also the increased data dependencies between neighboring P-Cores have an impact on the gaps. However, as what we will see in the

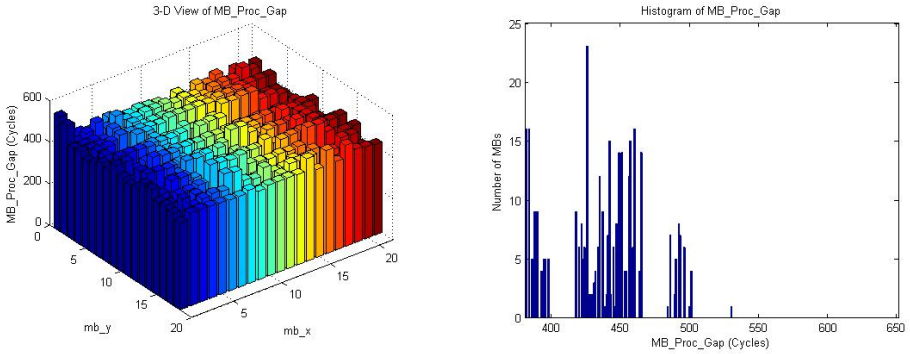

Figure 5.18: MPG of a Dual-Core System (CIF, YUV420, Intra-Frame)

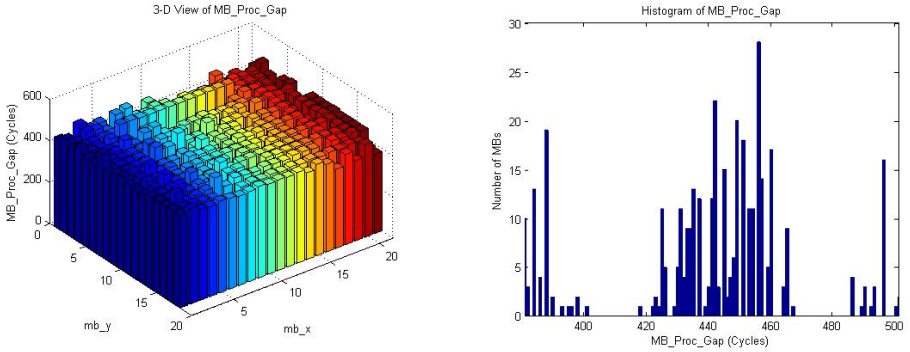

Figure 5.19: MPG of a Dual-Core System (CIF, YUV420, Inter-Frame)

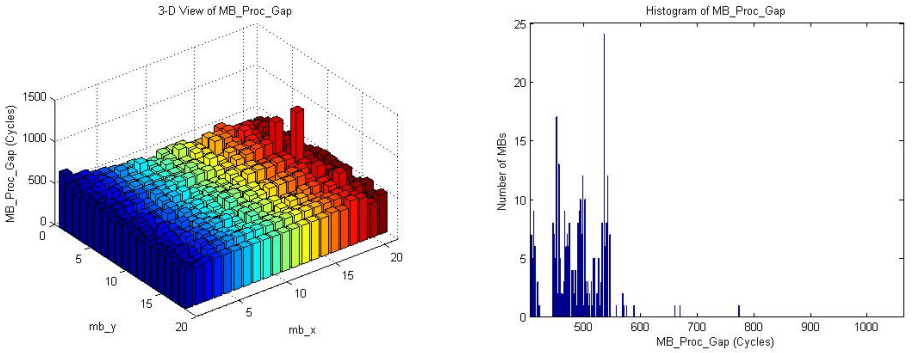

Figure 5.20: MPG of a Three-Core System (CIF, YUV420, Intra-Frame)

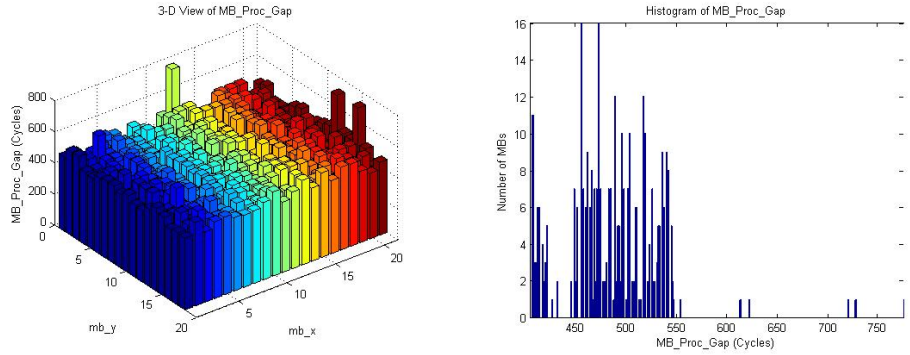

Figure 5.21: MPG of a Three-Core System (CIF, YUV420, Inter-Frame)

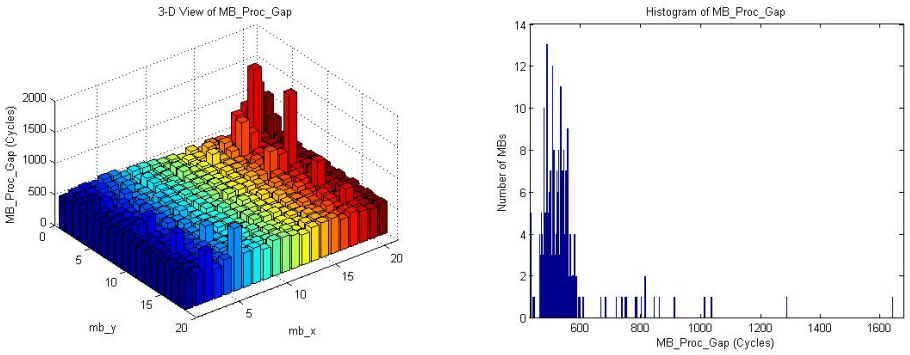

Figure 5.22: MPG of a Quad-Core System (CIF, YUV420, Intra-Frame)

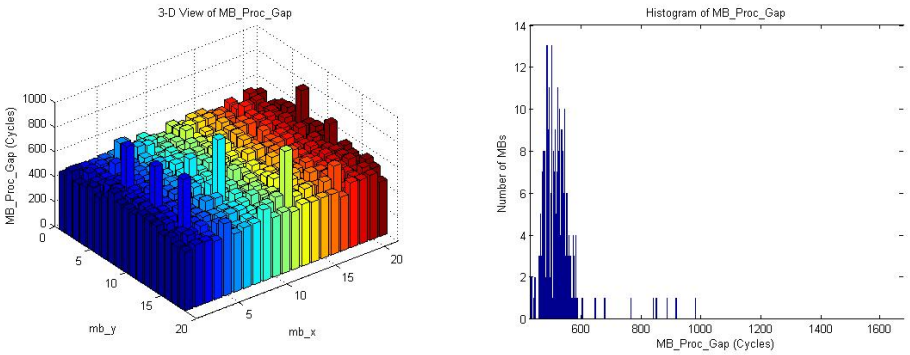

Figure 5.23: MPG of a Quad-Core System (CIF, YUV420, Inter-Frame)

following sections, this impact on system performance is very limited.

5.4.2 MB Processing Time

MB Processing Time (MPT) is a number which represents the duration a P-Core processes a MB. It has nothing to do with data dependencies and task scheduling since when the SC-Core issued a command to a P-Core to start processing a MB, all the needed neighboring data should be ready. All the P-Core needs to do is to transfer these data to its local memory and start to run.

Factors which may impact the MPT include:

- MB content and algorithms implemented in the peripheral core. This will not change with the increase of cores.
- Intensity of the memory accesses.
  - **Local Shared Memory accesses between every two P-Cores** The intensity of this kind of memory access will not increase with the increase of the number of P-Cores since the communication between every two peripheral cores has nothing to to do with the system size.
  - **Global Shared Memory accesses Raw** This includes accesses to raw image buffer and reference buffer. These kinds of buffers will be shared by all P-Cores. So, with the increase of the number of P-Cores, the intensity of memory accesses will inevitably increase and thus deteriorate the system performance. However, this problem can be solved in wavefront architecture by introducing extra pre-fetch units and system data caches. This will be discussed in next section.

Table 5.11: Average MB Processing Time (CIF YUV420)

	Dual-Core		Three-Core		Quad-Core	
	# Cycles	Δ%	# Cycles	Δ%	# Cycles	Δ%
Intra	46180	0%	46497	0.68%	46695	1.11%
Inter	40956	0%	41615	1.60%	41855	2.20%

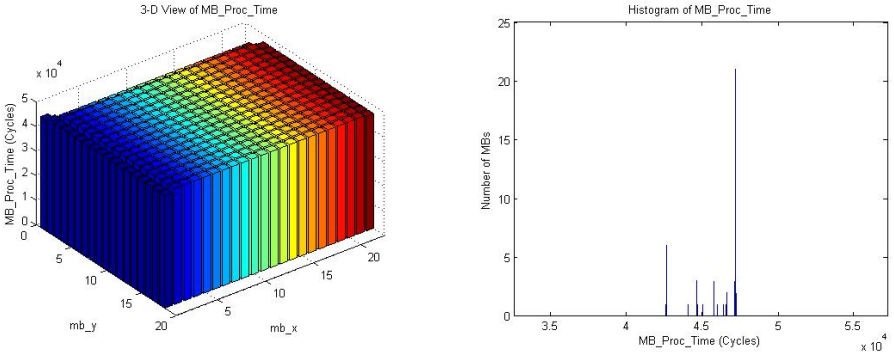

Figure 5.24: MPT of a Dual-Core System (CIF, YUV420, Intra-Frame)

Figure.5.24∼5.29 the MPT and its histogram for a dual-core, three-core and a quad-core system with intra/inter-prediction. Table.5.11 shows the average MPT. From this table, we observed a very small increase of MPT when increasing the number of

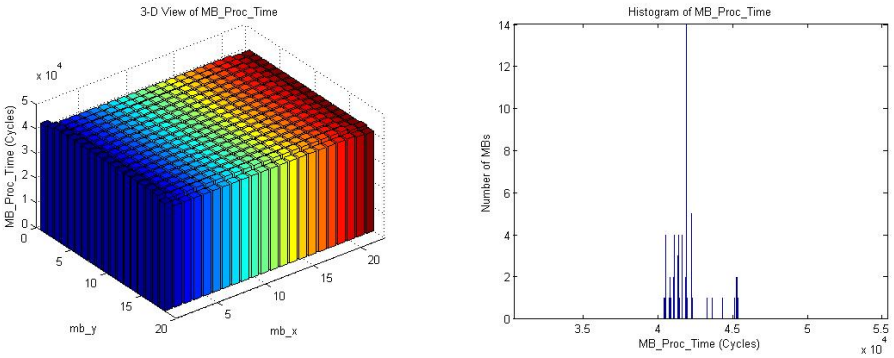

Figure 5.25: MPT of a Dual-Core System (CIF, YUV420, Inter-Frame)

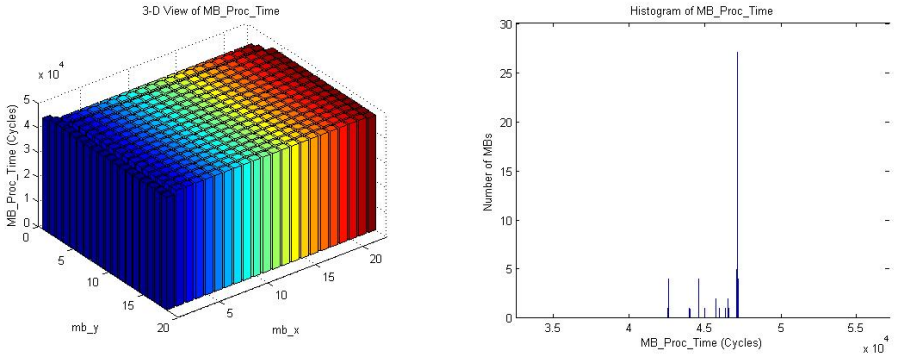

Figure 5.26: MPT of a Three-Core System (CIF, YUV420, Intra-Frame)

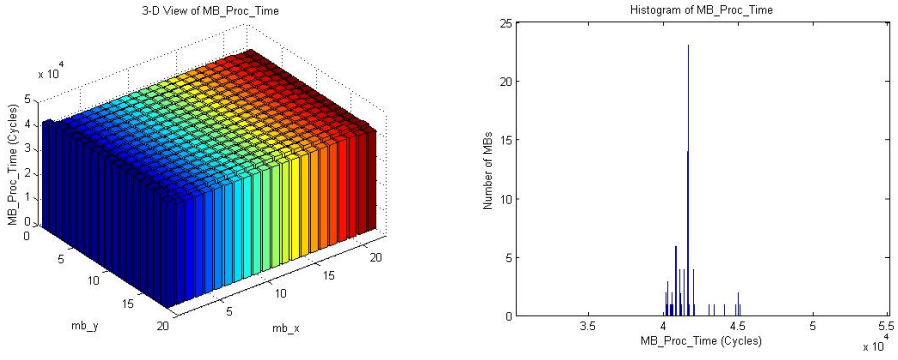

Figure 5.27: MPT of a Three-Core System (CIF, YUV420, Inter-Frame)

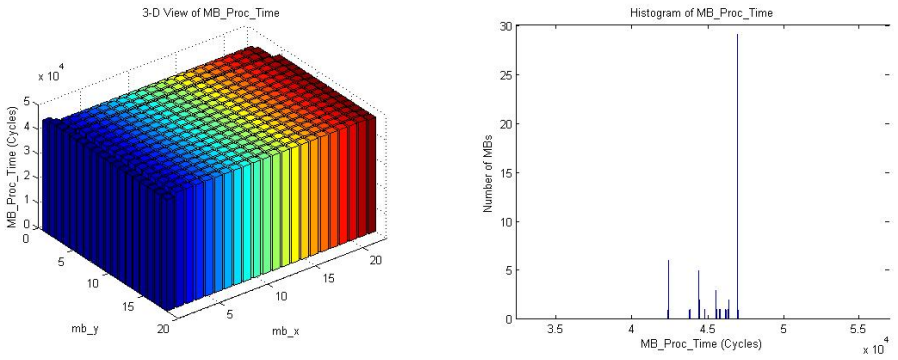

Figure 5.28: MPT of a Quad-Core System (CIF, YUV420, Intra-Frame)

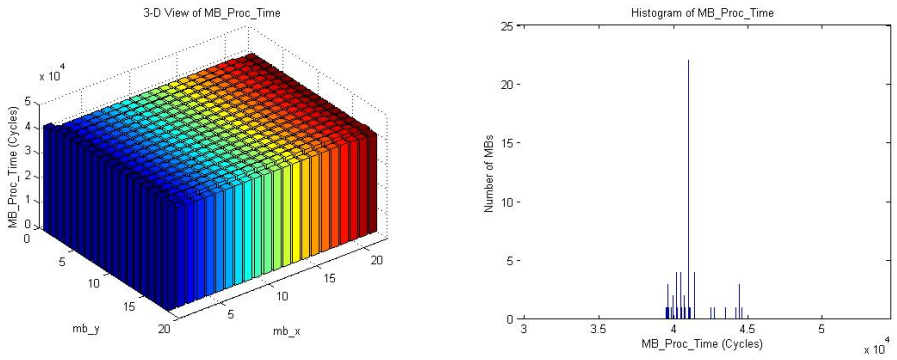

Figure 5.29: MPT of a Quad-Core System (CIF, YUV420, Inter-Frame)

P-Cores from two to four.

5.4.3 System Upscaling

From the simulation data we collected, we know when using a quad-core system to encode a CIF format video (like ICE.CIF) in 150MHz, the system throughput is going to be 32.1 FPS for I-frames and 35.8 FPS for P-frames. We meet the real-time requirement.

From dual-core to three-core and quad-core, the $\Delta\%$ of single P-Core performance (MPG+MPT) are 0.8% and 1.3% respectively for I-frames; 1.7% and 2.4% respectively for P-frames. The speed-ups from dual-core to three-core and quad-core are 1.49 and 1.97 respectively for I-frames, and 1.47 and 1.95 respectively for P-frames, which are very close to the theoretical values 1.5 and 2.0. This shows the overhead is very small compared with the system speed-up it achieves.

In some circumstances, our verified quad-core system may not satisfy performance requirement. For example, people may want to encode a real-time (25 FPS) HD1080P video sequence with 150MHz, then we will have to upscale the existing system.

Since the traffic to access local shared memory between P-Cores will not change with the increase of number of P-Cores, we now focus on the global shared memories and system control algorithm.

5.4.3.1 System Control Algorithm Up-scaling

In our simulation platform, the system control algorithm is implemented by software running in the SC-Core. Whenever a "MB finish IRQ" is asserted, the "system control ISR" will be called to assign a new task to an available P-Core. If the number of P-Cores increase, the ISR will have more P-Cores to check and the data dependencies will become more complex. This will thus deteriorate the performance, as what's shown in Table.5.10.

Even though the system performance drop caused by this is very small (compared with MPT), people may also want to use a pure hardware solution with the sacrifice of flexibility. In a typical hardware solution for the algorithm described in Figure.5.10, IRQ is replaced by signal lines and the SC-Core is replaced by a small state machine which only spends couple of cycles to assign a new task. So the performance fluctuation will be ignorable.

5.4.3.2 Global Shared Memory Up-scaling

In global shared memory, the reference buffer is shared among four P-Cores and the raw image buffer is shared by the camera module and four P-Cores. According to Table.5.11, the increase of number of cores will have impact on the system performance which is caused by the increased traffic on the memory interface.

When we expand the system with more P-Cores, the traffic to access this memory will increase as well. Also, in order to improve the efficiency of the memory

interface, we want the traffic distribution along the time axis to be more flat.

To solve this issue, a possible solution is as Figure.5.30. In this solution, we defined a system with N P-Cores and M Data-Caches. Each Data-Cache is only accessible by four P-Cores and stores raw image data and reference data which are needed by these four P-Cores in a short time window. The pre-fetch module is in charge of updating the content in all these Data-Caches and must make sure in any time window, all the needed data are in these Data-Caches.

It has been proved that in our quad-core simulation, one such Data-Cache with four P-Cores will satisfy the real-time requirement of CIF format videos. So, only if the Data-Cache can be pre-fetched on time, the system performance will be improved linearly with the number of P-Cores.

There is a M-Channel memory interface between the global shared memory and M Data-Caches. Access requests to the memory are buffered in the channel FIFOs. This will help to smooth the traffic on the memory interface.

With this solution, to encode a real-time (25 FPS) HD1080P video sequence with 150MHz, a system with 82 P-Cores and a global shared memory (32-bit) running at 233MHz will be sufficient.

This is just an up-scaling of our verified quad-core system. In real application, people may choose a pure hardware solution. In that case, much less P-Cores may be sufficient to meet the requirement.

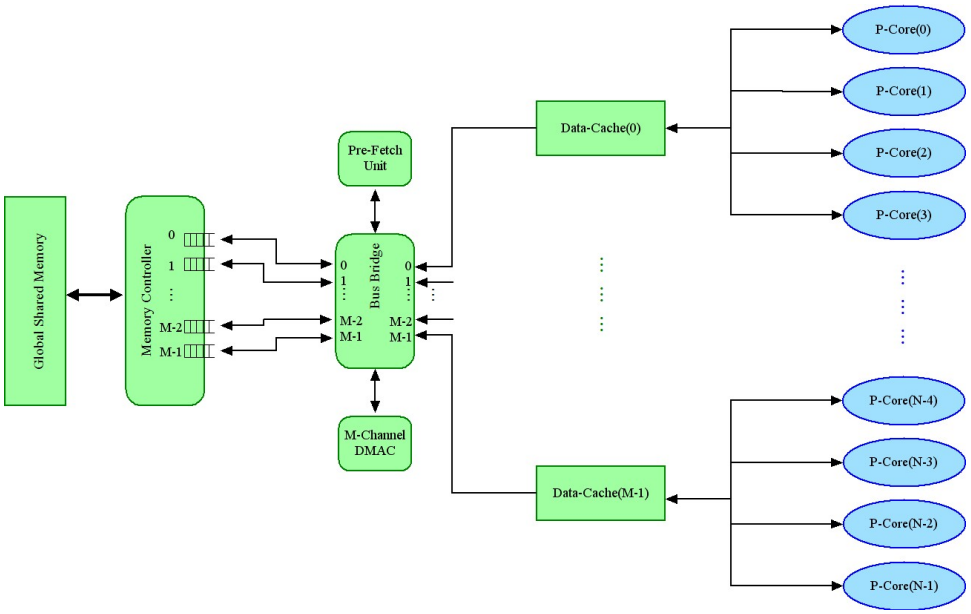

Figure 5.30: A Solution for Global Shared Memory Up-scaling

Chapter 6

Conclusions

6.0.4 Research Summary

In recent years, portable multimedia applications are playing a significant role in multimedia systems. It's becoming more and more popular to integrate video codec into personal mobile devices, like cell phones and PDAs, etc. As the newest video coding standards, H.264/AVC shows great performance and advantages over its ancestors. However, its encoding/decoding complexity also increased by several times and thus make it extremely hard for real-time applications in mobile devices.

In our research, we found the difficulties for the real-time application of H.264/AVC are in three aspects: (1) Memory access. A lot of the new features of H.264/AVC, like multiple reference frames, variable block sizes, quarter-pel fractional motion search, etc, make the power consumption on memory access increase dramatically. (2) Fast algorithm. A typical H.264/AVC encoder algorithm needs to explore all possible search points, block modes to make the optimal decision. As in [14], this process will take a very long time to finish. (3) Parallel Architectures. With the demand of real-time high-resolution video encoding using H.264/AVC, a single-processor solution may not

be sufficient to meet the requirement. People have to rely on parallel architectures. However, due to the complex data dependencies among neighboring blocks in the video content, it is very difficult to split the task among several processors and at the same time keep the same encoding efficiency (bit-rate vs. PSNR).

To address these difficulties, this thesis is focused on fast algorithms, data reuse and parallel architecture of H.264/AVC encoder. For data reuse, we proposed a partially forward processing algorithm (PFPA) in section 2.1 to reuse the reference information to avoid load the same reference data multiple times. For fast algorithms, we studied the statistical features of fractional motion estimation (FME) and proposed a FME mode reduction scheme in section 2.2. For parallel algorithms, we proposed two solutions for MB level and block level respectively. For MB level, we proposed wavefront architecture in chapter 4. Theoretically, this architecture can extend an encoder to any desired number of processors without sacrificing encoding quality. According to [24], our solution is so far the most scalable approach to H.264 coding. For block level, we proposed a FME parallel architecture in section 2.3.

We have used JM Model [1] to verify these proposed algorithms separately. Also, in chapter 5, we used XTMP, a simulation environment from Tensilica [29] to verify the wavefront architecture in more detail. Implementations are discussed in this chapter and cycle-accurate results show that this architecture has very small overhead when number of P-Cores increases. We also discussed the up-scaling solution in section 5.4.3.

6.0.5 Future Work

There are a number of additional activities that could extend the results of this research.

- **Algorithm:** in order to fulfill the solution we discussed in section 5.4.3, a good pre-fetch algorithm needs to be explored.

- **Architecture:** need to extend the architecture to support CABAC which creates more data dependencies. Also, we need to study the possibility to extend the architecture to support H.265, next generation of video compression standard.

- **Verification:** the parallel solution we proposed is in architecture level. Even though we verified it using XTMP, which is cycle accurate, we still need to verify it in more detail, like RTL level and use power analysis tools to get a relationship between the power number and the number of processors.

Bibliography

[1] H.264 jm reference model. *http://iphome.hhi.de/suehring/tml/.*

[2] Xiph.org test media. *http://media.xiph.org/video/derf/.*

[3] S.M. Akramullah, I. Ahmad, and M.L. Liou. Parallelization of mpeg-2 video encoder for parallel and distributed computing systems. In *Circuits and Systems, 1995., Proceedings., Proceedings of the 38th Midwest Symposium on*, volume 2, pages 834–837. IEEE, 2002.

[4] M. Alvarez, A. Ramírez, M. Valero, A. Azevedo, C. Meenderinck, and B. Juurlink. Performance Evaluation of Macroblock-level Parallelization of H. 264 Decoding on a cc-NUMA Multiprocessor Architecture. In *Proc. of the 4CCC: 4th Colombian Computing Conf.(April 2009)*. Citeseer.

[5] A. Azevedo, C. Meenderinck, and B. Juurlink. Performance Evaluation of Macroblock-level Parallelization of H. 264 Decoding on a cc-NUMA Multiprocessor Architecture.

[6] G. Bjøntegaard and K. Lillevold. Context-adaptive VLC (CVLC) coding of coefficients, JVT Document JVT-C028, Fairfax, VA, May 2002.

[7] S. Chen, S. Chen, and S. Sun. P3-CABAC: A Nonstandard Tri-Thread Parallel Evolution of CABAC in the Manycore Era. *Circuits and Systems for Video Technology, IEEE Transactions on*, 20(6):920–924, 2010.

[8] T.C. Chen, S.Y. Chien, Y.W. Huang, C.H. Tsai, C.Y. Chen, T.W. Chen, and L.G. Chen. Analysis and architecture design of an HDTV720p 30 frames/s H. 264/AVC encoder. *Circuits and Systems for Video Technology, IEEE Transactions on*, 16(6):673–688, 2006.

[9] T.C. Chen, Y.W. Huang, and L.G. Chen. Analysis and design of macroblock pipelining for H. 264/AVC VLSI architecture. In *Circuits and Systems, 2004. ISCAS'04. Proceedings of the 2004 International Symposium on*, volume 2. IEEE, 2004.

[10] T.C. Chen, Y.W. Huang, and L.G. Chen. Fully utilized and reusable architecture for fractional motion estimation of H. 264/AVC. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings.(ICASSP'04). IEEE International Conference on*, volume 5. IEEE, 2004.

[11] Y.J. Chen, C.L. Yang, and P.H. Wang. PM-COSYN: PE and memory co-synthesis for MPSoCs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 1590–1595. IEEE, 2010.

[12] Y.K. Chen, X. Tian, S. Ge, and M. Girkar. Towards efficient multi-level threading of H. 264 encoder on Intel hyper-threading architectures. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 63, 2004.

[13] Y. Cheng, Z. Wang, J. Guo, and K. Dai. Research on intra modes for inter-frame coding in H. 264. In *Computer Supported Cooperative Work in Design, 2005. Proceedings of the Ninth International Conference on*, volume 2, pages 740–744. IEEE, 2005.

[14] I. Draft. Recommendation and final draft international standard of joint video specification (ITU-T Rec. H. 264— ISO/IEC 14496-10 AVC). *Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, JVTG050*, 2003.

[15] M. Flierl and B. Girod. Generalized B pictures and the draft H. 264/AVC video-compression standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):587–597, 2003.

[16] W.T. Hsieh, J.C. Yeh, and S.Y. Huang. PAC duo system power estimation at ESL. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 815–820. IEEE, 2010.

[17] Z.M. Hsu, I.Y. Chuang, W.C. Su, J.C. Yeh, J.K. Yang, and S.Y. Tseng. System Performance Analyses on PAC Duo ESL Virtual Platform. In *2009 Fifth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, pages 406–409. IEEE, 2009.

[18] Y.W. Huang, T.C. Chen, C.H. Tsai, C.Y. Chen, T.W. Chen, C.S. Chen, C.F. Shen, S.Y. Ma, T.C. Wang, B.Y. Hsieh, et al. A 1.3 TOPS H. 264/AVC single-chip encoder for HDTV applications. In *2005 IEEE International Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC*, pages 128–588, 2005.

[19] Y.C. Kao, H.C. Kuo, Y.T. Lin, C.W. Hou, Y.H. Li, H.T. Huang, and Y.L. Lin. A high-performance VLSI architecture for intra prediction and mode decision in H. 264/AVC video encoding. In *Circuits and Systems, 2006. APCCAS 2006. IEEE Asia Pacific Conference on*, pages 562–565. IEEE, 2007.

[20] Y. Kim, J.T. Kim, S. Bae, H. Baik, and H.J. Song. H. 264/AVC decoder parallelization and optimization on asymetric multicore platform using dynamic load balancing. In *Multimedia and Expo, 2008 IEEE International Conference on*, pages 1001–1004. IEEE, 2008.

[21] P. List, A. Joch, J. Lainema, G. Bjontegaard, and M. Karczewicz. Adaptive deblocking filter. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):614–619, 2003.

[22] H.S. 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*, 13(7):598–603, 2003.

[23] 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*, 13(7):620–636, 2003.

[24] C. Meenderinck, A. Azevedo, B. Juurlink, M. Alvarez Mesa, and A. Ramirez. Parallel scalability of video decoders. *Journal of Signal Processing Systems*, 57(2):173–194, 2009.

[25] S. Momcilovic and L. Sousa. Development and evaluation of scalable video motion estimators on GPU. In *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, pages 291–296. IEEE, 2009.

[26] T. Moriyoshi and S. Miura. System IP Core Research Laboratories, NEC Corporation, Japan.

[27] T. Moriyoshi and S. Miura. Real-time H. 264 encoder with deblocking filter parallelization. In *Consumer Electronics, 2008. ICCE 2008. Digest of Technical Papers. International Conference on*, pages 1–2. IEEE, 2008.

[28] B. Pieters, C. Hollemeersch, P. Lambert, and R. Van de Walle. Motion estimation for H. 264/AVC on multiple GPUs using Nvidia CUDA. *Proceedings of SPIE on CD-rom*, page 1, 2009.

[29] X. Processor. Tensilica Inc.

[30] Y. Song, Y. Ma, Z. Liu, T. Ikenaga, and S. Goto. Hardware-oriented direction-based fast fractional motion estimation algorithm in H. 264/AVC. In *Multimedia and Expo, 2008 IEEE International Conference on*, pages 1009–1012. IEEE, 2008.

[31] Y. Song, M. Shao, Z. Liu, S. Li, L. Li, T. Ikenaga, and S. Goto. H. 264/AVC Fractional Motion Estimation Engine with Computation Reusing in HDTV1080P Real-Time Encoding Applications. In *Signal Processing Systems, 2007 IEEE Workshop on*, pages 509–514. IEEE, 2007.

[32] W.C. Su, J.K. Yang, K.C. Liu, S.Y. Tseng, and W.S. Wang. Waiting cycle analysis on H. 246 decoder run in PAC Duo platform. In *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, pages 922–925. IEEE, 2010.

[33] Y. Su and M.T. Sun. Fast multiple reference frame motion estimation for H. 264. In *Multimedia and Expo, 2004. ICME'04. 2004 IEEE International Conference on*, volume 1, pages 695–698. IEEE, 2005.

[34] S. Sun, D. Wang, and S. Chen. A highly efficient parallel algorithm for H. 264 encoder based on macro-block region partition. *High Performance Computing and Communications*, pages 577–585, 2007.

[35] P. Tiwari and E. Viscito. A parallel MPEG-2 video encoder with look-ahead rate control. In *Acoustics, Speech, and Signal Processing, 1996. ICASSP-96. Conference Proceedings., 1996 IEEE International Conference on*, volume 4, pages 1994–1997. IEEE, 2002.

[36] C.Y. Tsai, T.C. Chen, and L.G. Chen. Low power entropy coding hardware design for H. 264/AVC baseline profile encoder. In *Multimedia and Expo, 2006 IEEE International Conference on*, pages 1941–1944. IEEE, 2006.

[37] P.K. Tsung, W.Y. Chen, L.F. Ding, C.Y. Tsai, and L.G. Chen. Single-iteration full-search fractional motion estimation for quad full HD H. 264/AVC encoding. In *Multimedia and Expo, 2009. ICME 2009. IEEE International Conference on*, pages 9–12. IEEE, 2009.

[38] S.W. Wang, S.S. Yang, H.M. Chen, C.L. Yang, and J.L. Wu. A Multi-core Architecture Based Parallel Framework for H. 264/AVC Deblocking Filters. *Journal of Signal Processing Systems*, 57(2):195–211, 2009.

[39] Y.J. Wang, C.C. Cheng, and T.S. Chang. A fast algorithm and its VLSI architecture for fractional motion estimation for H. 264/MPEG-4 AVC video coding. *Circuits and Systems for Video Technology, IEEE Transactions on*, 17(5):578–583, 2007.

[40] Z. Wang, L. Liang, X. Zhang, J. Sun, D. Zhao, and W. Gao. A Novel Macro-Block Group Scheme of AVS Coding for Many-Core Processor.

[41] Z. Wang, L. Liang, X. Zhang, J. Sun, D. Zhao, and W. Gao. A Novel Macro-Block Group Based AVS Coding Scheme for Many-Core Processor. *Advances in Multimedia Information Processing-PCM 2009*, pages 356–367, 2009.

[42] H. Wei, Y. Junqing, and L. Jiang. The design and evaluation of hierarchical multi-level parallelisms for H. 264 encoder on multi-core architecture. *Computer Science and Information Systems/ComSIS*, 7(1):189–200, 2010.

[43] T. Wiegand, H. Schwarz, A. Joch, F. Kossentini, and G.J. Sullivan. Rate-constrained coder control and comparison of video coding standards. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):688–703, 2003.

[44] T. Wiegand, G.J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H. 264/AVC video coding standard. *IEEE Transactions on circuits and systems for video technology*, 13(7):560–576, 2003.

[45] C.L. Wu, C.Y. Kao, and Y.L. Lin. A high performance three-engine architecture for H. 264/AVC fractional motion estimation. In *Multimedia and Expo, 2008 IEEE International Conference on*, pages 133–136. IEEE, 2008.

[46] K. Xu and C.S. Choy. A five-stage pipeline, 204 cycles/MB, single-port SRAM-based deblocking filter for H. 264/AVC. *Circuits and Systems for Video Technology, IEEE Transactions on*, 18(3):363–374, 2008.

[47] K. Xu and C.S. Choy. A power-efficient and self-adaptive prediction engine for H. 264/AVC decoding. *IEEE Transactions on very large scale integration (VLSI) systems*, 16(3):302–313, 2008.

[48] X. Xu and Y. He. Comments on motion estimation algorithms in current JM software. *JVT of ISO/IEC MPEG and ITU-T VCEG, Doc. JVT-Q089, Oct*, 2005.

[49] X. Xu and Y. He. Improvements on fast motion estimation strategy for H. 264/AVC. *Circuits and Systems for Video Technology, IEEE Transactions on*, 18(3):285–293, 2008.

[50] L. Yang, K. Yu, J. Li, and S. Li. An effective variable block-size early termination algorithm for H. 264 video coding. *IEEE transactions on circuits and systems for video technology*, 15(6):784–788, 2005.

[51] S.S. Yang, S.W. Wang, and J.L. Wu. A Parallel Algorithm for H. 264/AVC Deblocking Filter Based on Limited Error Propagation Effect. In *Multimedia and Expo, 2007 IEEE International Conference on*, pages 1858–1861. IEEE, 2007.

[52] L. Yu-Sheng, L. Chin-Feng, H. Chia-Cheng, C. Han-Chieh, and H. Yueh-Min. Power-Aware DVB-H Mobile TV System on Heterogeneous Multicore Platform. *EURASIP Journal on Wireless Communications and Networking*, 2010, 2010.

[53] N.H.C. Yung and K.K. Leung. Spatial and temporal data parallelization of the H. 261 video coding algorithm. *Circuits and Systems for Video Technology, IEEE Transactions on*, 11(1):91–104, 2002.

[54] L. Zhang and W. Gao. Reusable architecture and complexity-controllable algorithm for the integer/fractional motion estimation of H. 264. *Consumer Electronics, IEEE Transactions on*, 53(2):749–756, 2007.

[55] Z. Zhao and P. Liang. A frame-level data re-use & mode decision strategy for H. 264/AVC encoders. In *Proceedings of the 2006 international conference on Wireless communications and mobile computing*, pages 57–60. ACM, 2006.

[56] Z. Zhao and P. Liang. A highly efficient parallel algorithm for H. 264 video encoder. In *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, volume 5. IEEE, 2006.

[57] Z. Zhao and P. Liang. Data partition for wavefront parallelization of H. 264 video encoder. In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pages 4–2672. IEEE, 2006.

[58] Z. Zhao and P. Liang. A Statistical Analysis of H.264/AVC FME Mode Reduction. *IEEE Transactions on circuits and systems for video technology*, To Appear.