SOFTWARE OPTIMIZATION OF H.263 VIDEO ENCODER ON PENTIUM PROCESSOR WITH MMX TECHNOLOGY

Pohsiang Hsu and K. J. Ray Liu

Department of Electrical and Computer Engineering
University of Maryland at College Park
College Park, Maryland, USA

ABSTRACT

A key enabling technology for the proliferation of multimedia PC's is the availability of fast video codecs, which are the basic building blocks of many new multimedia applications. Since most industrial video coding standards (e.g., MPEG1, MPEG2, H.261, H.263, etc.) only specify the decoder syntax, there are a lot of rooms for optimization in a practical implementation. When considering a specific hardware platform like the PC, the algorithmic optimization must be considered in tandem with the architecture of the PC. Specifically, an algorithm that is optimal in the sense of number of operations needed may not be the fastest implementation on the PC. This is because special instructions are available which can perform several operations at once under special circumstances. In this work, we describe a fast implementation of H.263 video encoder for the Pentium processor with MMX technology.

1. INTRODUCTION

Recent advances in the personal computer industry have provided the necessary computation power and storage required by many multimedia applications. These tremendous technological advances have enabled the PCs to perform image/video compression and decompression efficiently in software only. Some advantages of implementing the video codec in software for the PC are the elimination of expensive hardware, the ease of upgrade through replacement of software modules, and the wide availability of PCs.

Video coding requires tremendous amount of computations. There have been many fast algorithms proposed in the literature to ease the computation load for various components in a video codec. Typically these fast algorithms are proposed and compared with each other by using the total number of operations as a criterion assuming a general-purpose processor without considering the target hardware platform. However, the comparisons can be misleading when we consider a software implementation on a specific hardware platform. This is because each microprocessor has it's own strengths and weaknesses which places bias on certain operations. For example, some microprocessors may have dedicated hardware to execute the multiply-accumulate operation in one cycle. Then, it will be advantageous to arrange an algorithm such that the multiply-accumulate operation occurs frequently. Thus, we see that the design of a fast software only video codec is highly dependent on the hardware platform. Each component of the video codec must be properly selected to take maximum advantage of the underlying hardware.

In this paper, we present a fast software implementation of a H.263 video encoder on the Intel Pentium with MMX technology processor, which powers a vast majority of the computers in the world. The optimization of the encoder is performed iteratively through profiling and recoding to speed up the inner loops. Traditional optimization techniques were used along with the MMX instructions to achieve speed-up. Optimization techniques such as removals of loop invariant computation, strength reduction, loop jamming, loop unrolling, and table lookup were used. Loop unrolling was used often in tight loops with MMX instructions to achieve speed-up through software pipelining.

2. MMX TECHNOLOGY

MMX technology is an extension to the Intel Architecture whose aim is to improve the performance of multimedia and communications algorithms. With the addition of the MMX technology comes fifty-seven new instructions, and eight new 64-bit registers. The MMX instruction set was designed by analyzing a broad range of software applications in the field of multimedia and communications. In the analysis of these applications from different domains, it is found that certain common characteristics exist for a majority of the core time-consuming code sequences. From on these observations, it was found that a salient feature of many multimedia algorithms was the execution of the same set of operations on a large number of small data elements. Therefore, the MMX technology adopted the SIMD (Single Instruction, Multiple Data) architecture to enable exploitation of the data parallelism inherent in these applications.

The new set of SIMD instructions defined by MMX technology performs parallel operations on multiple data elements packed into the 64-bit register. Three new packed data types and the 64-bit quad-word are defined. The packed data types contain several smaller fixed-point data elements. The three packed data types are packed byte, packed word, and packed doubleword. Basically, packed byte, word, or double word contains eight bytes, four words, or two doublewords, respectively. The data inside a MMX register is interpreted as one of these four new types depending on the executed instruction. New instructions introduced by the MMX technology includes packed arith-

0-7803-6536-4/00/$10.00 (c) 2000 IEEE

metic instructions, saturating arithmetic instructions, data manipulation instructions, and logical instructions.

The packed arithmetic allows the same arithmetic operations to be applied to each individual data element of a packed data type in parallel. Another key feature provided by the MMX instructions is the ability to perform signed or unsigned saturating arithmetic on each data elements of a packed data type in parallel. In conventional fixed-point arithmetic, we can only obtain the correct lower order bits when overflow occurs. On the other hand, saturating arithmetic clips the result to the largest or the smallest possible value for the given data type when overflow occurs. Saturating arithmetic is found to be very useful in image processing since it eliminates the clipping operations found at the end of most image processing operations. The data manipulation instructions provided by MMX technology are for conversion between the new data types. These instructions are very important when an algorithm requires higher fixed-point precision in its intermediate stages. The pack instructions convert a bigger packed data type to a smaller packed data type while the unpack instructions convert a smaller packed data type to a bigger packed data type. In addition, the unpack instruction can perform an interleaved merge operation which can be used efficient to perform insertion, transposition, and other data manipulation operations.

3. H.263 ENCODER IMPLEMENTATION DETAIL

The major computational blocks in a H.263 video encoder are motion estimation, motion compensation, DCT / IDCT, quantization/de-quantization, entropy coding, and inter/intra-coding. Among these functional blocks, motion estimation and DCT/IDCT are typically the most computational intensive portion of the encoder. To get an idea of the computational load distribution of the functional blocks from a typical H.263 encoder, we encoded a video sequence using the ITU TMN H.263 video encoder provided by Telenor to obtain a profile of the encoding computational load. Intel's V-Tune software package, which is a visual optimization/profiling tool, was used to monitor the encoding process. In Fig 1, we show the distribution of CPU load obtained by the profiling. Indeed, we can see that the motion estimation and the DCT/IDCT are the most time consuming portions where the motion estimation occupies a majority of the CPU power.

H.263 uses block matching motion estimation and compensation to exploit the temporal correlation between adjacent frames. Various block matching algorithms has been proposed in the literature and basically they differ in the matching criteria, search strategy, or block size. The method that the TMN H.263 Encoder employs is the full search block matching algorithm using sum of absolute difference (SAD) as the matching criterion. This method guarantees a global minimum by exhaustively comparing all possible candidates in the search space. However, the complexity of such an search is prohibitively high as we can see from Fig. 1 which makes it impractical for real time software implementation.

Many fast blocking matching techniques have been proposed in the literature to reduce the complexity of the motion vector search by trading off the prediction efficiency. These techniques can be divided into two categories namely fast matching or fast search. In fast matching, different matching criteria that requires fewer computations [6] than the sum of absolute difference (SAD) or the mean square error (MSE) are used. In fast search, the SAD or MSE criteria is typically still used but the average number of points searched is smaller the total number of points in the entire search space [5] [4] [7].

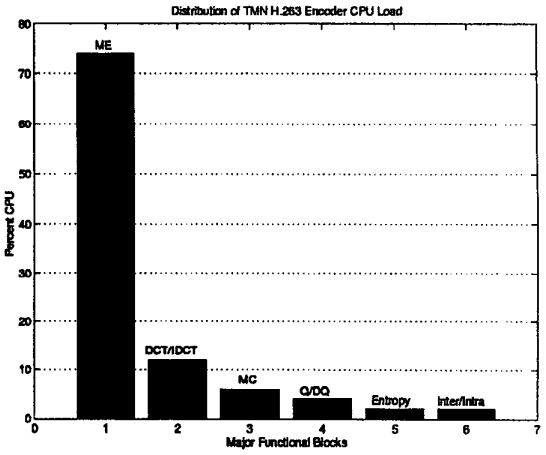

Figure 1: Distribution of CPU Load for TMN H.263 Video Encoder. The abbreviations ME, MC, Q/DQ, stands for Motion Estimation, Motion Compensation, and Quantization/Dequantization, respectively.

In our video encoder, we employed a fast search block matching that is based on the three step search [5]. In this scheme, we start our search at the center of the search region. From the starting point, we search its eight surrounding neighbors to find the best matching out of all nine points. If the starting point was found as the best match, we stop the process and declare it as the motion vector. Otherwise, we set the newly found best match as the new starting point and repeating the process over again. We note that the computation of the matching criteria for the eight neighboring points of a starting point might be need in the search process of future starting points due to overlap. Therefore, the computed matching scores are stored so that they can be access instead of computed later if needed. Along with the searching strategy, we tried several different matching criterions including the MSE, MAD, and the error variance. In terms of computational complexity, the MAD matching criterion required the least amount of computation. However, the error variance matching criterion which is the variance of the difference between the block and its prediction resulted in better prediction among the three. We have implemented the MAD and error variance matching measure using MMX instructions, which significantly improved the speed of these operations. The computation of the MAD matching criterion involves evaluation of the

following types of arithmetic operation:

$$MAD = \sum_{m=0}^{15} \sum_{n=0}^{15} |y(m,n) - x(m,n)|, \qquad (1)$$

where each element of x and y are eight-bit quantities. Since the same arithmetic operation is applied to each element independent of other elements, we can take advantage of the inherent instruction level parallelism through MMX instructions. We note that the resulting dynamic range of subtraction between two eight bits unsigned numbers is nine bits. Therefore, if we perform full precision subtraction with x and y, we must work with 16-bits quantities which reduces the parallelism to four instead of eight. However, the absolute difference between two 8-bits numbers x and y can be performed in 8-bits precision using saturating arithmetic as follows. We first compute $x - y$ and $y - x$ using saturating arithmetic and then we logically OR the two difference together to form the absolute difference. If x equals to y, then the computation produces the correct result. If x does not equals to y, we note that one of the two quantities $x - y$ and $y - x$ is the absolute difference while the other one will be saturated to zero. Thus the correct result can be obtained by logically OR the two differences together. Therefore, we can perform the absolute difference using eight-bit precision, which will allow us to work on eight elements at a time.

In order to perform motion estimation, we must generate each candidate blocks through motion compensation followed by computation of the matching criterion. Typically, the motion vector search is done using full integer accuracy until the best match is found. Then, a half pixel (i.e. 0.5) motion vector search is done centered on the best integer motion vector. Thus, we must generate the eight candidate half pixel blocks using bilinear interpolation at the end of the best integer motion vector search to find the best half pixel accurate motion vector. This process involves averaging two pixels or four pixels to find the missing pixels where special attention must be paid to ensure proper rounding is performed. We can organize the computation into three cases according to the motion vector. In the first case, only the vertical component of the motion vector contains a half pixel component. In the second case, only the horizontal component of the motion vector contains a half pixel component. In the third case, both the horizontal and vertical components of the motion vector contain half pixel components. For the first case, we need to perform averaging across adjacent rows to find the predicted value. For the second case, we need to perform averaging across adjacent columns to find the predicted value. For the third case, we need to perform averaging across the adjacent rows and columns to find the predicted value.

We take advantage of the MMX technology to perform the bilinear interpolation in the motion compensation process. Let's consider the first case where we are averaging across the rows to obtain the predicted value. Ideally, we want to take advantage of the instruction level parallelism by performing the averaging process on eight pixels on two adjacent rows at a time. However, We first note that the additions can not be performed on the eight elements in parallel because of possible overflow. Furthermore, the MMX technology does not support parallel shift on byte elements (the smallest size it support is on word elements). Thus, in a straightforward implementation, we will have to convert the pixel from an eight-bit quantity to a sixteen-bit quantity, which reduce the parallelism from eight to four.

Fortunately, there is another way to perform this operation while preserving the parallelism to eight and achieve proper rounding at the same time. Let's consider the case of averaging two integers X_1 and X_2 to form Y. Suppose we simply perform the following operations to form Y_1, $Y_1 = X_1 >> 1 + X_2 >> 1$, where $>>$ indicates a right shift. Comparing Y and Y_1 reveals that the following relationship hold, $Y = X_1 >> 1 + X_2 >> 1 + Z$, where Z is the logical OR of the least significant bit of X_1 and X_2. Based on this observation, we can perform the averaging of two arrays of eight pixels in the following way to preserve the maximum parallelism of eight. First, we construct a new array whose element contains the logical OR of the least significant bit for the corresponding elements of the two arrays using the 64 bit logical OR and 64 bit logical AND instructions provided by MMX technology. In essence, this step generates an array Z. Next, we need to perform the shift operation on each byte element of the array. As we have pointed out, we can not perform parallel shift on byte elements directly. However, this can be done in two steps since we are working on eight bytes at a time. First, we zero out the least significant bit of each byte element in the two input arrays. Then, we simply regard the eight bytes as one 64-bit quantity and perform a 64 bit logical shift by one to obtain the desired result. Afterwards, the three arrays are added in parallel to get the final result. Similarly, the averaging process for the second case can be done in the same way by first transposing the block of data. Furthermore, the third case can be computed in a similar manner by separating the computation between the two least significant bits and the rest.

The H.263 encoder uses DCT to reduce the spatial redundancy of the video sequence. The DCT is popular in image compression because it achieves good energy compaction and it has many fast algorithms available. For our encoder, the direct fast 2-D DCT developed by Feig [8] and the scaled 1-D DCT developed by Arai, Agui, and Nakajima [9] were considered as candidates. The Feig 2-D DCT is the most efficient algorithm proposed in the literature in terms of number of operations. It is a true 2-D method that requires 54 multiplications, 462 additions, and six multiplications by 1/2 which can be done by arithmetic shifts. On the other hand, the most efficient 1-D DCT proposed in the literature is by Arai, Agui, and Nakajima. This method requires 13 multiplications and 29 additions. However, eight of the thirteen multiplications can be absorbed into the quantization stage and thus the 1-D DCT can be computed with 5 multiplications and 29 additions. Therefore, the 2-D DCT can be computed by applying this fast 1-D DCT on the rows and the columns using a total of 80 multiplications and 464 additions. This is the best known approach for computing a separable 2-D scaled DCT. Therefore, the computational complexity of the separable approach is higher than the direct 2-D approach. However, in terms of implementation, the 1-D DCT approach was better suited for MMX instructions than the direct 2-D ap-

Enc:	Seq:	File Size: (KB)	PSNR: (dB)	Time 1: (sec)	Time 2: (sec)
TMN	A	11.3	37.1	46.6	14.2
Enc.	B	33.8	33.2	42.3	13.8
Opt.	A	17.3	37.1	4.3	1.5
Enc.	B	40.8	33.1	4.5	1.6

Table 1: Speed Comparison between H.263 TMN encoder and our optimized H.263 encoder. The amount of time needed to encode 100 frames on Pentium 200 MHz with MMX is shown in Time 1 and on Pentium II 400 MHz with MMX is shown in Time 2. Sequence A denotes Miss America and sequence B denotes Carphone.

proach due to the computational flow. The separable 2-D DCT was implemented in assembly and takes advantage of the MMX instructions. The capabilities to multiply several elements together in parallel and multiply/accumulate were provided by the MMX technology, which we found to be extremely useful.

We compared the speed of our optimized implementation of a H.263 encoder against the TMN H.263 encoder. For comparison, both encoders were used to compress 100 frames of two test sequences and the total time needed was recorded. The hardware platforms used were a Pentium 200 MHz with MMX and a Pentium II 400 MHz computer and the results are summarized in Table 3. As we can see, the TMN encoder compresses only two frames per second on the Pentium 200 MHz. Moreover, the number of frames compressed per second will be significantly lowered on real communications applications due to the fact that the CPU power have to be distributed among many processes instead of just on the video encoder. On the other hand, our optimized encoder compresses about 23 frames per second on the same platform, which is about 10 times faster the TMN encoder. The distribution of the CPU load for our optimized H.263 encoder is shown in Fig. 2.

As we can see, the optimized encoder is fast enough so that the CPU can be shared with other processes and we still can obtain good frame rate. The drawback of our optimized encoder is lowered compression efficiency. This is because we traded compression efficiency with computation complexity.

4. CONCLUSION

In this paper, we considered the problem of software optimization of video codecs on the Pentium with MMX platform. We described an actual implementation of a fast H.263 video encoder utilizing the MMX technology. The H.263 video encoder is composed of several different components. The optimization is done by selecting fast algorithms for each component that takes advantage of the underlying hardware platform. We compared the speed-up with the H.263 standard TMN video encoder and found that the speed up is about tenfold at the expense of compression efficiency.

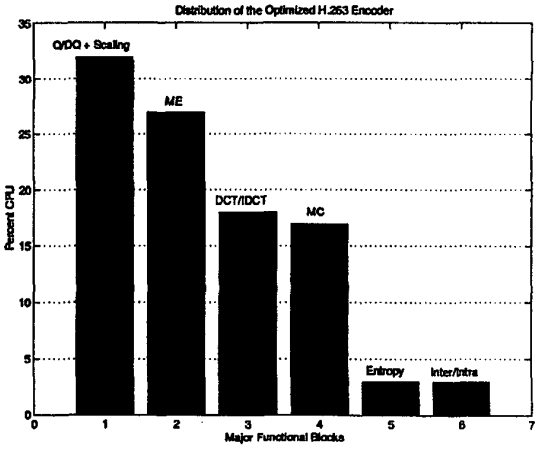

Figure 2: Distribution of CPU Load for our Optimized H.263 Video Encoder. The abbreviations ME, MC, Q/DQ + Scaling, stands for Motion Estimation, Motion Compensation, and Quantization/Dequantization plus Scaling for IDCT/DCT, respectively.

5. REFERENCES

[1] Telenor Research. H.263 encoder/decoder tmn1.5 ver. 1.7. *ftp://bonde.nta.no/pub/tmn*,1996.

[2] Intel Corporation, *The Complete Guide to MMX™ Technology*, McGraw Hill, 1997.

[3] V. Allan, R. Jones, and R. Lee, S. Allan, "Software Pipelining," *ACM Computing Surveys*, Sept. 1995.

[4] L. Liu and E. Feig, "A Block-based Gradient Descent Search Algorithm for Block Motion Estimation," *IEEE Trans. on Circuit and Systems for Video Technology*, vol. 6, pp. 419-422, Aug. 1996.

[5] R. Li, B. Zeng and M. L. Liou, "A New Three-Step Search Algorithm for Block Motion Estimation," *IEEE Trans. on Circuit and Systems for Video Technology*, vol. 4, pp. 438-442, Aug. 1994.

[6] X. Lee and Y. Zhang, "A Fast Hierarchical Motion-Compensation Scheme for Video Coding Using Block Feature Matching," *IEEE Trans. on Circuit and Systems for Video Technology*, vol. 6, pp. 627-634, Dec. 1996.

[7] J. Chalidabhongse and C. Kuo, "Fast Motion Vector Estimation Using Multiresolution-Spatio-Temporal Correlations," *IEEE Trans. on Circuit and Systems for Video Technology*, vol. 7, pp. 477-488, 1997.

[8] E. Feig and E. Linzer, "Discrete Cosine Transform Algorithms for Image Data Compression," *Proceedings Electronic Imaging '90 East*, pp. 84-87, 1990.

[9] Y. Arai, T. Agui and M. Nakajima, "A Fast DCT-SQ Scheme for Images," *Trans. of the IEICE*, vol. 71, pp. 1095-1097, Nov. 1988.