Performance Analysis of Intel MMX Technology for an H.263 Video Encoder

Ville Lappalainen
Nokia Research Center, Speech and Audio Systems Laboratory[1]
P.O. Box 100,
FIN-33721 Tampere, Finland
ville.lappalainen@research.nokia.com

1. ABSTRACT

The purpose of this study was to find out what performance gains an H.263 video encoder can achieve using Intel MMX technology. This study resulted in an overall speedup of over 65%.

1.1 Keywords

Performance, Intel MMX, video encoder, H.263

2. INTRODUCTION

The International Telecommunication Union, Telecommunication Standardisation Sector (ITU-T), has specified the first version of the H.263 standard for video coding for low bit rate communication enabling compression of video sequences to bit rates below 64 kbps. The standard specifies a hybrid coding method containing discrete cosine transformed (DCT) intra frames, and motion compensated and DCT transformed inter frames. Applications for which the standard is suitable include video phone applications in the current Public Switched Telephone Network (PSTN), and in future mobile networks with data rates similar to the data rates of PSTN. An H.263 encoder is one component of a video phone application.

Intel MMX technology is designed to accelerate multimedia and communications applications. The technology exploits the parallelism typical of many of the algorithms in these applications by offering new instructions and data types. MMX technology enables parallel operation on small data elements.

Amdahl's Law [1] can be used to analyse the speedup of different routines. Amdahl's Law (defined in Equation 2-1) states that the performance enhancement possible with a given improvement is limited by the fraction (of the execution time) that the improved feature is used.

A routine containing MMX code can perform independent operations on eight elements in parallel, for example. This implies that the performance enhancement of this parallel operation ($Speedup_{enhanced}$ in Equation 2-1) is 8. However, the overall speedup of the routine depends on the fraction that the parallel operation is used ($Fraction_{enhanced}$ in Equation 2-1).

$$Speedup_{overall} = \frac{1}{\frac{Fraction_{enhanced}}{Speedup_{enhanced}} + (1 - Fraction_{enhanced})}$$

Equation 2-1. Amdahl's Law.

Some examples of the performance enhancement of MMX technology can be found in [12], [13] and [11]. There are different ways to find out the speedup achieved using MMX technology. Firstly, one can observe the speedup of time-consuming routines. Secondly, one can observe the overall speedup of the whole application. The speedup of an individual routine can be found out by calculating the number of instructions in it, and then, by making some assumptions about, e.g., cache misses, the total number of clock cycles for the routine can also be calculated. Another way to find out the speedup of an individual routine is to measure its actual execution time. The only reasonable way to find out the overall speedup is to measure the actual execution time of the whole application.

In this study, all speedups were found out by measuring the actual execution times. The overall speedups of a few multimedia applications are presented in [13]. These speedups were also found out by measuring the actual execution times.

An optimised routine utilising MMX technology is written in assembly language. To find out the speedup achieved using MMX technology, one can compare this MMX routine to an integer assembly routine, a floating-point assembly routine, or to a C routine. Different comparisons yield different speedups. Different levels of optimisation may also have a considerable influence on the speedup.

In this study, the performance comparisons were made between optimised MMX code and optimised scalar code.

ACM Multimedia'98, Bristol, UK
© 1998 ACM 1-58113-036-8/98/0008 $5.00

[1] The author is also a postgraduate student at Tampere University of Technology (Department of Information Technology, Software Systems Laboratory), Finland.

In this context, scalar code means assembly code that does not contain MMX instructions, but can contain either integer or floating-point instructions.

The kind of comparison used in this study gives more information about the performance enhancement of MMX technology, because when comparing optimised MMX code and non-optimised C code, only the fraction of the speedup is caused by MMX technology. The rest of the speedup is caused by the assembly implementation of MMX code and different optimisation levels between MMX and C code.

To find out what performance gains an H.263 encoder can achieve using MMX technology, two performance comparison tests were executed. One resulted in overall speedup on the encoder; the other resulted in the speedup of each of the most time-consuming routines. Amdahl's Law is used to explain why the overall speedup of the routine performing a parallel operation is lower than the performance enhancement of this parallel operation.

Chapter 3 presents an overview of the H.263 encoder as well as an optimised implementation of the time-consuming routines. Chapter 4 contains the results of the performance comparisons. It also describes the test environment and measuring techniques used. Chapter 5 concludes the work.

3. H.263 VIDEO ENCODER

The optimised routines of the encoder were hand-tuned for optimal performance on a Pentium processor using MMX technology. General optimisation techniques, such as instruction scheduling for paired execution [5], correct alignment of data [5], and loop unrolling [4], were applied. Pentium-specific optimisation techniques, e.g., instruction scheduling, were applied carefully in both the MMX and scalar routines. Both the MMX and scalar routines are small enough to fit in the internal code cache (16 KB).

3.1 Overview

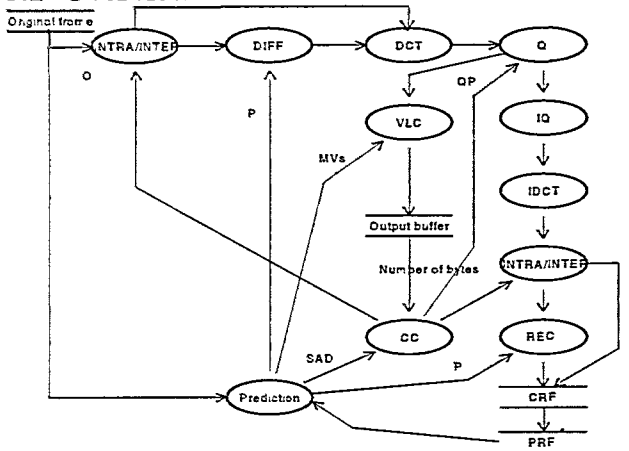

Figure 3-1 Data flow diagram of an H.263 video encoder [10].

The basic structure of an H.263 encoder is shown in Figure 3-1, which follows the Data Flow Diagram (DFD) notation of the Object Modeling Technique (OMT) format [14]. This paper concentrates on the most time-consuming parts of the encoder: motion estimation (prediction), DCT and quantisation, and IDCT and dequantisation routines, see [10] for further details of the encoder.

Motion estimation is a process, which is used to reduce temporal redundancy in a video stream. In a typical motion estimation process, the image is split into blocks of N x N pixels and a resembling prediction block in the previously reconstructed image (reference image) is searched for each block in the current image. This procedure is called block matching. The way the prediction block is found is determined by the used block matching algorithm, a computation-constrained layered search [3].

The prediction block is determined by comparing the block in the current image with each candidate block inside the search area in the reference image. The comparison of the two blocks is made by comparing corresponding pixels in the two blocks, using some error measure, e.g., the Sum of Absolute Differences (SAD) defined in Equation 3-1, where: N is the block width, F_0 and F_{-1} respectively are the current and the previously reconstructed frame, k and l define the position of the block in the current frame (multiples of N), and x and y define the position of the prediction block in the previously reconstructed frame.

$$SAD(x,y,k,l)=\sum_{i,j=0}^{N}\left|F_{-1}(i+x,j+y)-F_0(i+k,j+l)\right|$$

Equation 3-1. Definition of the Sum of Absolute Differences.

The calculation of the SAD is a dominant operation during the motion estimation. Another time-consuming operation is bilinear interpolation, which is needed because of half-pixel precision during the motion estimation. Half-pixel precision means that if one or both of the motion vector components indicate half-pixel position, the pixel values are found using bilinear interpolation as described in Figure 3-2. "/" indicates division by truncation.

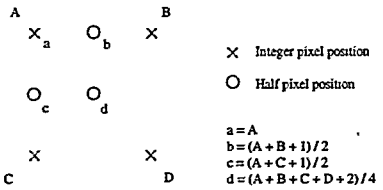

Figure 3-2. Half-pixel prediction by bilinear interpolation [8].

The motion estimation process also contains two additional routines, which were optimised. They are introduced in the next subsection.

The 8x8 DCT (IDCT) routine used in the encoder is described in [2]. At first, the routine calculates the horizontal transform, processing rows. Secondly, it calculates the vertical transform, processing columns. The original C versions of the DCT and IDCT routines are taken from the H.263 encoder written by Telenor (tmn v2.0).

3.2 Optimised MMX Code

The original C version of the 8x8 DCT routine calculates with 32-bit floating point numbers. The MMX version calculates with a poorer resolution: it uses fixed point numbers having a precision of three fractional bits. The MMX version stores two coefficients in one MMX register. It processes two rows at a time, with four passes.

Quantisation is defined in Equation 3-2 for INTRA blocks (except for the first coefficient, called the INTRA DC coefficient) and in Equation 3-3 for INTER blocks.

$$LEVEL = |COF| / (2 \times QUANT)$$

Equation 3-2. Quantisation for an INTRA block [9].

$$LEVEL = (|COF| - QUANT/2) / (2 \times QUANT)$$

Equation 3-3. Quantisation for an INTER block [9].

The INTRA DC coefficient is uniformly quantised with a quantisation step of 8. The quantisation parameter QUANT may take integer values from 1 to 31. COF stands for a transform coefficient to be quantised. LEVEL stands for the absolute value of the quantised version of the transform coefficient.

Both quantisation equations have (2xQUANT) as a divisor. This enables the calculation of a table containing values 1/(2xQUANT) for each possible QUANT value. This table makes possible the quantisation with multiplication. Because the calculation is done with limited precision, the quantisation using multiplication does not always yield exactly as accurate results as the quantisation using division. Quantisation processes four coefficients in one MMX register.

The abbreviation for these routines (and also for the corresponding scalar routines) to be used later on is DCT+Q, covering both the DCT and quantisation routines.

The original C version of the 8x8 IDCT calculates with 32-bit floating point numbers. Again, the MMX version calculates with a poorer resolution: it uses fixed point numbers having a precision of four to six fractional bits. The MMX version of the IDCT utilises parallelism basically in the same way as the MMX version of the DCT.

The dequantisation (for non-INTRA DC coefficients) is performed on four pixels in parallel, using the following formulae that give the relationship between coefficient levels (LEVEL), quantisation parameter (QUANT) and reconstructed coefficients (REC):

$$LEVEL = 0 \Rightarrow REC = 0$$

$$QUANT = odd \Rightarrow |REC| = QUANT \times (2 \times |LEVEL| + 1)$$

$$QUANT = even \Rightarrow |REC| = QUANT \times (2 \times |LEVEL| + 1) - 1$$

$$REC = sign(LEVEL) \times |REC|$$

Equation 3-4. Dequantisation [8].

The abbreviation for these routines is IQ+IDCT.

There are, altogether, eight SAD calculation functions, four for calculating the SAD for 16x16 blocks (abbreviations end with 16) and another four for 8x8 blocks (abbreviations end with 8). Two of these functions operate on the block having full-pixel coordinates (calcSADFF16/8). The other six functions operate on the block having half-pixel coordinates (calcSADHF16/8, calcSADFH16/8, and calcSADHH16/8).

The calculation of the SAD for the block having full-pixel coordinates utilises the parallelism and also the saturating arithmetic described in [6]. All pixel differences can be calculated in parallel, because they are all independent operations. Eight pixels are processed in parallel.

The calculation of the SAD for the block having half-pixel coordinates is a somewhat more complex operation. In this study, an optimised solution was developed. Consider a pixel, whose x-component is a half-pixel value and y-component is a full-pixel value. The half-pixel value for this pixel is obtained from Equation 3-5:

$$b = (A + B + 1) / 2$$

Equation 3-5. Half-pixel prediction, case b [8].

A and B are defined in Figure 3-2, and are presented as 8-bit unsigned numbers. Because calculation is done with integer numbers and the division is defined to be done by truncation, we cannot write Equation 3-5 as b = A/2 + B/2 + 1/2. It is possible to use Equation 3-5 for only four pixels in parallel, because A+B+1 can take more than 8 bits and thus need 16 bits of one MMX register. But if A and B could be divided separately, without first adding them and constant one, it would be possible to calculate eight pixels in parallel. In this case, Equation 3-6 turns out to be equivalent to Equation 3-5:

$$b = A/2 + B/2 + ((A \ AND \ 1) \ OR \ (B \ AND \ 1))$$

Equation 3-6. Half-pixel prediction, case b. Optimised version.

The two other cases of bilinear interpolation (cases c and d in Figure 3-2) can also be calculated using the same principle.

The function that makes the INTRA/INTER decision for an individual macroblock (CalcIntraDecision) is basically a SAD calculation function without bilinear interpolation routines. See [8] for the definition of a macroblock and coding modes (INTRA/INTER).

The function that calculates the difference macroblock (preCalcDiffMB) calculates the difference on a pixel-by-pixel basis. Because the pixel difference calculation is an independent operation, it can be calculated in parallel. The pixel difference can have values ranging from -255 to 255, which take 16 bits of one MMX register. Thus, this operation can be performed on only four pixels in parallel.

The MMX versions of the DCT, IDCT and quantisation routines calculate with a poorer resolution than the original C versions. However, the differences in the encoding results (e.g., subjective and objective video quality) between the MMX and the C versions are almost negligible. On the average, the difference in objective quality, in terms of PSNR (Peak Signal-to-Noise Ratio), is only 0.02 dB [10]. In practice, this means that no loss in subjective quality can be observed.

3.3 Optimised Scalar Code

Scalar versions of the DCT and IDCT use floating-point numbers. The reason for using floating-point instructions instead of integer instructions is the nonpipelined, 11-cycle integer multiply operation. The floating-point multiply takes only three cycles to execute and executes in a pipelined unit. Only one row or column is processed simultaneously. All coefficients in that row or column are, however, kept in the FPU registers during the calculation, thus the precision at that stage is very good, 80 bits. The data is stored in a temporary buffer between the two passes (horizontal and vertical) by using 32-bit floating point numbers.

Quantisation must be performed in integer numbers. Since the integer divide instruction is very slow, multiplication quantisation routine is implemented. The multiplication quantisation works similarly to that in MMX code. Note that the integer multiply instruction is much slower than that for MMX. Both quantisation and dequantisation process one value at a time.

During dequantisation, the (2x|LEVEL|+1) part in Equation 3-4 is acquired from a look-up table.

Scalar versions of the SAD calculation functions calculate the absolute difference using the same technique as the MMX version, but the calculation can only be done pixel by pixel [6]. However, the scalar versions of the bilinear interpolation routines can operate on four pixels in parallel and they also use the same technique as the MMX version.

Scalar versions of functions CalcIntraDecision and preCalcDiffMB operate on one pixel at a time.

The scalar versions of the DCT, IDCT and quantisation routines calculate with a better resolution than the MMX versions of the corresponding routines. Thus, the differences in the encoding results between the scalar and the C versions are even smaller than those between the MMX and the C versions.

4. PERFORMANCE COMPARISONS

4.1 Test Environment

The test environment for the performance testing was the following: Fujitsu ICL ErgoPro S450 PC, having 32 megabytes of random access memory and based on a 133 MHz Pentium MMX processor. The H.263 encoder used is a Win32 console application that was run in Windows 95. During the measurements there was a minimum computational load caused by other programs. In addition, the priority of the encoder process was set to the maximum value. This was done by calling Windows 95 functions SetPriorityClass and SetThreadPriority with parameters REALTIME_PRIORITY_CLASS and THREAD_PRIORITY_TIME_CRITICAL, respectively.

Four original uncompressed QCIF-sized (the luminance resolution: 176x144) sequences, Akiyo, Claire, Coastguard and Foreman, were used. These sequences were selected from a set of standard test sequences that were used during the development of video coding standards ITU-T Recommendation H.263 [8] and MPEG4 [7]. Frames 0 to 299 of all sequences were encoded with a target frame rate of 10 fps. Two optional coding modes of the H.263 recommendation, the Unrestricted Motion Vector mode (option D) and the Advanced Prediction mode (option F), were used [8].

In order to reduce the influence of disk caching on the execution time, the same sequence was encoded three times in a row. Because the purpose was to measure the performance of the encoding process, not any I/O operations, only the last execution was taken into account. The proportional time usage of I/O operations was very small in the last execution, because the disk cache was filled during the two former executions.

Two different performance comparison tests were executed. During the first test, all four test sequences were encoded with bit rates from 8 to 48 kbps using both the MMX and scalar versions. Due to the different nature of test sequences, a unique set of target bit rates was selected for each sequence. This test resulted in overall speedup achieved using MMX technology. During the second test, the Foreman sequence was encoded with a target bit rate of 24 kbps. The execution times of the optimised routines were measured. This test resulted in the speedup of each individual routine.

In the first test, the total execution time of the encoder was measured using Windows 95 GetTickCount function, which returns the time in milliseconds. In the second test, the execution time of each optimised routine was measured using Pentium's RDTSC (Read from Time Stamp Counter) instruction, which returns clock cycles.

4.2 Overall Speedup

Table 4-1 shows the encoding speed (in frames/s) of the MMX version for all sequences, when both the options D & F are used. Table 4-2 contains the corresponding information about the scalar version.

On the average, the MMX implementation is approximately 67% faster than the scalar implementation.

Bitrate	Akiyo	Coastguard	Claire	Foreman	Mean
8 kbps	17.27		17.29		17.28
10 kbps	17.21		16.96		17.09
12 kbps	17.01		16.84	14.12	15.99
14 kbps	16.94		16.63	14.41	15.99
16 kbps	16.63	15.14	16.44	14.56	15.69
24 kbps	16.23	14.39	15.99	14.12	15.18
28 kbps	15.81	14.51	15.83	13.85	15.00
36 kbps	15.70	13.89	15.46	13.43	14.62
48 kbps	15.35	13.39	14.91	12.94	14.15
Mean	16.46	14.26	16.26	13.92	15.23

Table 4-1. Encoding speed for test sequences (MMX).

Bitrate	Akiyo	Coastguard	Claire	Foreman	Mean
8 kbps	10.12		10.02		10.07
10 kbps	10.01		9.86		9.94
12 kbps	9.90		9.83	8.43	9.39
14 kbps	9.82		9.74	8.53	9.36
16 kbps	9.78	9.06	9.67	8.68	9.30
24 kbps	9.64	8.93	9.53	8.57	9.17
28 kbps	9.62	8.80	9.42	8.47	9.08
36 kbps	9.45	8.57	9.20	8.28	8.88
48 kbps	9.29	8.35	9.08	8.07	8.70
Mean	9.74	8.74	9.59	8.43	9.13

Table 4-2. Encoding speed for test sequences (scalar).

4.3 Speedup of Each Optimised Routine

Table 4-3 shows the minimum execution time (in cycles), average execution time (in cycles), execution frequency, and average speedup (in %) for each optimised routine, when encoding the Foreman stream. Two subsequent rows contain information on each routine, the upper reporting the MMX version, the lower the scalar version.

Routine	Version	Min. time	Average time	Times executed	Average speedup
DCT+Q	MMX	1035	1226	63954	58.60
DCT+Q	scalar	2498	2961	64062	
IQ+IDCT	MMX	1071	1198	63954	57.83
IQ+IDCT	scalar	2661	2841	64062	
calcSADFF16	MMX	275	396	152655	65.71
calcSADFF16	scalar	1075	1155	149759	
calcSADHF16	MMX	663	770	21306	66.90
calcSADHF16	scalar	2067	2326	21335	
calcSADFH16	MMX	566	757	21296	70.10
calcSADFH16	scalar	2076	2532	21330	
calcSADHH16	MMX	1078	1239	42568	63.97
calcSADHH16	scalar	3148	3439	42616	
calcSADFF8	MMX	107	131	92076	62.78
calcSADFF8	scalar	331	352	92652	
calcSADHF8	MMX	215	239	90184	61.26
calcSADHF8	scalar	571	617	90288	
calcSADFH8	MMX	189	231	93492	66.13
calcSADFH8	scalar	580	682	93064	
calcSADHH8	MMX	317	347	106740	61.40
calcSADHH8	scalar	835	899	106800	
CalcIntraDecision	MMX	318	515	10791	63.73
CalcIntraDecision	scalar	1130	1420	10791	
preCalcDiffMB	MMX	1478	2808	10659	5.39
preCalcDiffMB	scalar	1816	2968	10677	

Table 4-3 Min. execution time, average execution time, execution frequency and average speedup for each routine.

The overall average speedup of all routines was calculated in two different ways: by using arithmetic mean (58.65%) and weighted arithmetic mean (60.68%). The latter uses the execution frequencies as weighting factors.

The overall encoding speeds for the MMX and scalar versions for these encoder runs are slightly lower than the corresponding speeds presented in Table 4-1 and Table 4-2 (13.79 and 8.49 frames/s, respectively), because of the timing instructions around each routine. The overall speedup based on these encoding speeds is 62.43%. This is quite close to the average speedup of all routines (60.68%). However, the share of the total execution time for these routines is 63.21% for the scalar version and 40.37% for the MMX version.

In general, the speedup of each routine seems to be relatively close to the average speedup of all routines. There is only one exception; the preCalcDiffMB routine (speedup: 5.39%). This routine does only one subtract operation, reads both of its input operands from the memory and writes the results into the memory in a loop. Although the MMX version does the subtract operation on 4 pixels in parallel (the scalar version processes 1 pixel at a time), most of the time is spent on memory reads and writes.

Only fraction of the execution time of the preCalcDiffMB routine can utilise parallelism. That fraction can be calculated by using Amdahl's Law (defined in Equation 2-1). The overall speedup of the routine ($Speedup_{overall}$ in Equation 2-1) is 5.39%. The speedup in the enhanced mode ($Speedup_{enhanced}$ in Equation 2-1) is 4. In this section, it is assumed that both the MMX and the scalar routines use both the pipelines similarly. Substituting into Equation 2-1:

$$1.0539 = \frac{1}{\frac{Fraction_{parallel}}{4} + (1 - Fraction_{parallel})}$$

Simplifying this equation yields: $Fraction_{parallel} = 0.06819$

Thus only 6.82% of the execution time of the preCalcDiffMB routine can utilise the parallelism. This is the main reason for the very low speedup. Other routines have different speedups in their enhanced modes ($Speedup_{enhanced}$ in Equation 2-1), such as 2 and 8 (explained in the following paragraphs). Other routines have better overall speedups than the preCalcDiffMB routine. This indicates that the fractions utilising the parallelism are greater than those of the preCalcDiffMB routine. However, for all these routines, these fractions are far from 100%.

The MMX versions of the DCT+Q and IQ+IDCT routines process 2 pixel rows (or columns) simultaneously during the DCT (IDCT) algorithm, and 4 pixels in parallel during quantisation (dequantisation). The corresponding figures for the scalar versions are 1 pixel row and 1 pixel. Because the DCT (IDCT) consumes more time than quantisation (dequantisation), the speedup in the enhanced mode is closer to 2 than 4.

The MMX version of the calcSADFF16 routine processes 8 pixels in parallel, while the scalar version processes only 1 pixel at a time (the speedup in the enhanced mode is 8). Still, the actual speedup is 65.71%. We can calculate the fraction that utilises the parallelism of the execution time of this routine by using Amdahl's Law as before. The result, 45.02%, explains the relatively low overall speedup of the routine compared to the speedup in the enhanced mode.

Other SAD calculation functions contain bilinear interpolation routines that consume more time than the calculation of the SAD. The MMX versions of the bilinear

interpolation routines process 8 pixels in parallel, while the scalar versions process 4 pixels in parallel (the speedup in the enhanced mode is closer to 2 than 8).

5. CONCLUSIONS

An H.263 encoder contains several time-consuming routines suitable for MMX optimisation, among them routines that perform independent operations on pixel data.

In this study, twelve time-consuming routines were written in both MMX and scalar code. This resulted an overall speedup of over 65% on the H.263 encoder, when comparing optimised MMX code and optimised scalar code (assembly code that does not contain MMX instructions).

It is difficult to compare the results of different performance comparison tests because of the different test applications, test environments and measuring techniques. An earlier study by the author [10] presents a speedup of almost 350% on the H.263 encoder, when comparing optimised MMX code and non-optimised C code (only compiler optimisations applied) using the same test environment and measuring techniques as in this study. A speedup of 80% on an MPEG1 video decoder, and 370% on an image filtering application are presented in [13]. Intel's goal for the Pentium processor with MMX technology was to provide an overall performance boost of 50% to 100% on multimedia applications [13].

When calculating the speedups of individual routines instead of measuring the actual speedups, the speedups tend to be higher, e.g., between 200% and 400% [12]. The reasons for these higher speedups are several simplifying assumptions about cache misses, for example.

Although MMX instructions can utilise the parallelism, the execution latencies of instructions accessing memory are similar to the latencies of non-MMX instructions. The penalties of cache misses, in particular, are similar for both the MMX and scalar code. As a typical multimedia application, an H.263 encoder suffers severely from the cache misses because it handles large amounts of data during the encoding process. The time-consuming routines contain frequent memory operations, and no complex operations occur between them. Additionally, the MMX code must arrange the data for its parallel operations.

The performance gains achieved using MMX technology are remarkable, although they depend on the application; the application should perform independent operations on small data elements in its time-consuming routines. To achieve additional performance gains, parallelism between instructions can be exploited by utilising Pentium's superscalar execution, for example. Applications that do not contain as frequent memory operations as the H.263 video encoder may achieve even higher speedups than those reported in this study.

6. ACKNOWLEDGMENTS

I thank Ilkka Haikala (my supervisor), Marko Luomi and Janne Juhola for useful discussions and comments. Special thanks to Jussi Lahdenniemi for his help with assembly coding.

7. REFERENCES

[1] Amdahl, G. A., Validity of the single-processor approach to achieving large scale computing capabilities, AFIPS Conf. Proc., vol. 30, Thompson, Washington, D.C., 1967, pp. 483-485.

[2] Chen, W.-H., Smith, C. H. and Fralick S. C., A Fast Computational Algorithm for the Discrete Cosine Transform, IEEE Transactions on Communications, vol. COM-25, 1977, pp. 1004–1009.

[3] Côté, G., Gallant, M. and Kossentini, F., Efficient Motion Vector Estimation and Coding for H.263-Based Very Low Bit Rate Video Compression, http://www.ece.ubc.ca/spmg/

[4] Hennessy, J., Patterson, D., Computer Architecture: A Quantitative Approach, San Mateo, CA Morgan Kaufmann Publ., 1990, pp. 315-318.

[5] Intel Architecture MMX™ Technology Developer's Manual (Order No. 243006-001), March 1996.

[6] Using MMX™ Instructions to Compute the AbsoluteDifference in Motion Estimation, AP-530, http://developer.intel.com/drg/mmx/AppNotes/

[7] International Organization for Standardization, ISO/IEC JTC1/SC29/WG11N1902 14496-2 Committee Draft (MPEG-4), November 1997.

[8] Draft ITU-T Recommendation H.263, 2 May, 1996.

[9] ITU-T Study Group 15, Video Codec Test Model Near-Term, Version 7 (TMN7), 25 February 1997.

[10] Lappalainen, V., Implementation of H.263 Video Encoder Using Intel MMX Technology, Master of Science Thesis, Tampere University of Technology, 1997.

[11] Levy, M., Multimedia Instructions Boost Host-Based Processing, EDN Asia, September 1996, pp. 20-36.

[12] Peleg, A. and Weiser, U., MMX Technology Extension to the Intel Architecture, IEEE Micro, August 1997, pp. 42-50.

[13] Peleg, A., Wilkie, S. and Weiser, U., Intel MMX for Multimedia PCs, Communications of the ACM Vol.40 No.1, January 1997, pp. 25-38.

[14] Rumbaugh, J. et al., Object-Oriented Modeling and Design, Prentice-Hall, 1991.