Performance Optimization of an MPEG-2 to MPEG-4 Video Transcoder

Hari Kalva, Anthony Vetro, and Huifang Sun
Mitsubishi Electric Research Labs, Murray Hill, NJ

ABSTRACT

The MPEG-2 compressed digital video content is being used in a number of products including the DVDs, camcorders, digital TV, and HDTV. The ability to access this widely available MPEG-2 content on low-power end-user devices such as PDAs and mobile phones depends on effective techniques for transcoding the MPEG-2 content to a more appropriate, low bitrate, video format such as MPEG-4. In this paper we present the software and algorithmic optimizations performed in developing a real time MPEG-2 to MPEG-4 video transcoder. A brief overview of the transcoding architectures is also provided.

The transcoder was targeted and optimized for Windows PCs with the Intel Pentium-4 processors. The optimizations performed exploit the SIMD parallelism offered by the Intel's Pentium-class processors with MMX support. The transcoder consists of two distinct components: the MPEG-2 video decoder and the MPEG-4 video transcoder. The MPEG-2 video decoder is based on the MPEG-2 Software Simulation Group's reference implementation while MPEG-4 transcoder is developed from scratch with portions taken from the MOMUSYS implementation of the MPEG-4 video encoder. The optimizations include: 1) generic block-processing optimizations that affected both the MPEG-2 decoder and the MPEG-4 transcoder and 2) optimizations specific to the MPEG-2 video decoder and the MPEG-4 video transcoder. The optimizations resulted in significant improvements both in MPEG-2 decoding as well as the MPEG-4 transcoding. With optimizations, the total time spent by the transcoder was reduced by over 82% with MPEG-2 decoding reduced by over 56% and MPEG-4 transcoding reduced by over 86%.

Keywords: MPEG-2, MPEG-4, Transcoder, Optimization, Intel CPU, MMX, Intel C++ Compiler

1. INTRODUCTION

With MPEG-2 video compression being used in a number of video-centric applications, more and more digital video content is becoming available in the MPEG-2 format. The widespread use of MPEG-2 video has also resulted in decreased costs that has further spread the use of MPEG-2 video. While MPEG-2 video is fit for a number of high quality digital video applications, it is not suitable for low bitrate applications. There are a number of applications such as video surveillance, and mobile multimedia services that require the same high quality MPEG-2 video to be delivered to a less capable device such as a PDA or cell phone over a lower bandwidth connection. MPEG-4 video compression algorithm is designed for low bitrate video and has features that enable transmission over less reliable medium such as wireless networks. Transcoding MPEG-2 video to MPEG-4 format is computationally intensive. A set of transcoding architectures have been developed that take advantage of the similarities in MPEG-2 and MPEG-4 video encoding to speedup the transcoding process [2]. These transcoding architectures avoid the need for full re-encoding of MPEG-4 video. The complexity-quality analysis of these architectures was reported in [1].

The desktop PCs are becoming more and more powerful with each generation of CPUs. Starting with its PentiumPro CPU, Intel has begun supporting an instruction set to speedup multimedia and video processing. With the advances in CPU speed and architecture, it is now possible to decode and playback HDTV resolution MPEG-2 video in real time with software players. While it is possible to transcode MPEG-2 video to MPEG-4 offline, there are a number of applications that require real-time transcoding of MPEG-2 video. Furthermore, offline transcoding of MPEG-2 video to MPEG-4 format at different bitrates and quality as demanded by all the different end users is not possible. We have developed a software implementation of an MPEG-2 to MPEG-4 video transcoder to run on Intel's Pentium-4 processors. The goals of the project are to transcode MPEG-2 video in real time and maximize the number of MPEG-2 video streams that can be transcoded simultaneously. The software and instruction set support available for Pentium-4 processors makes it an ideal platform to develop transcoding applications. The speedup is primarily achieved by using MMX instructions that operate on a set of data in parallel.

The block-based compression and decompression of video in MPEG-2 and MPEG-4 lends itself nicely for parallel processing. A parallel algorithm and implementation on a Single Instruction Multiple Data (SIMD) parallel architecture of the JPEG image compression, a block-based image compression algorithm, was reported in [3]. Intel's Pentium processors support a form of SIMD parallelism with instructions that operate on as may as 16 bytes in parallel. An efficient compiler is also necessary to take advantage of the architectural features offered by the processors. We used Intel's C++ compiler to optimize the transcoder application further. We report the performance gains due to the compiler optimizations alone. Using MMX instructions to optimize the performance of an H.263 encoder was presented in [4]. Even though the H.263 encoder has some commonalities with the MPEG-2 to MPEG-4 transcoder, the transcoder presents a different set of optimization problems.

The rest of this paper is organized as follows: a brief introduction to MMX technology is given in Section 2, followed by an overview of the transcoding architectures in Section 3. In Section 4 we discus the MMX optimizations used in the transcoder. The results are discussed in section 5.

2. OVERVIEW OF INTEL MMX TECHNOLOGY

SIMD stands for *single instruction multiple data* – a form of parallel processing where a single instruction stream operates on multiple data streams. Parallel computers with SIMD architectures typically use a large number of simple processors to solve complex problems. The SIMD support started to appear in microprocessors in the early 90s in microprocessors such as HP's MAX2 and Sun's VS extensions [5]. The most common processors today with SIMD instruction support are the Intel processors with MMX support and AMD processors with 3DNow! support. In this section we present an overview of the Intel MMX technology. A broad introduction to the MMX technology can be found in [6].

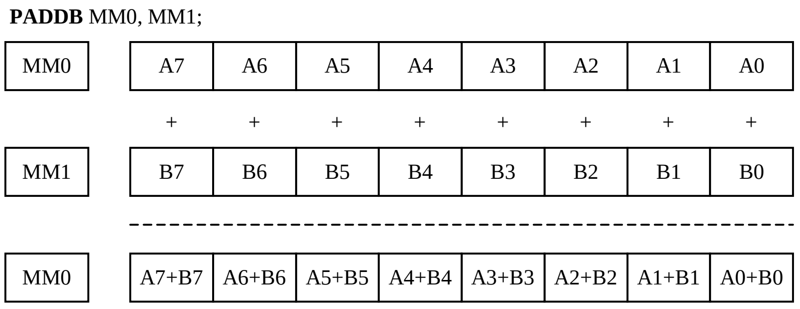

Figure 1. Illustration of SIMD Addition Using Intel MMX Instructions

The MMX technology in Intel processors introduces SIMD parallelism, new instructions, new data types, and eight 64-bit wide registers called MMX registers, MM0 to MM7. The extensions to MMX, called streaming SIMD extensions (SSE) introduced eight 128-bit wide registers called XMM registers, XMM0 to XMM7, and instructions that work with those registers. The SSE extensions were introduced in the Pentium-3 processor. Another extension to the instruction set was made with the introduction of SSE2 extensions in Pentium-4 and Xeon processors. In this paper, we use the term MMX to refer to MMX, SSE, and SSE2 extensions collectively and make explicit distinctions where necessary. Figure 1 illustrates the principle of SIMD parallelism. Registers MM0 and MM1 are 64-bit wide registers loaded with 8 bytes of data each. The instruction PADDB takes two MMX registers as operands and adds the packed 8-bit integers in the source operand (MM1) to the 8-bit integers in the destination operand (MM0) and stores the result in the destination operand. The addition of eight pairs of 8-bit integers that normally require eight separate ADD instructions can be performed with a single instruction. The PADDB instruction can also take 128-bit operands (XMM registers) and perform the addition of 16 pairs with a single instruction.

2.1 MMX Data Types

The MMX technology introduced four new data types: packed bytes, packed words, packed double-words, and quad-word. Each of these packed data types contains two or more basic data types packed into a 64-bit quantity. Packed byte thus contains 8 bytes, packed word contains 4 words, and a packed double word contains two double words. A quad-word is a single 64 bit quantity. The SSE extensions introduced a 128-bit packed single-precision floating-point data type. The SSE2 extensions introduced five more data types for use with 128-bit XMM registers: packed double-precision floating-point data type, packed byte integers, packed word integers, packed double-word integers, and quad-word integers. These data types for the 64-bit MMX and 128-bit XMM registers offer good possibilities to optimize compute intensive applications such as transcoding.

2.2 MMX Instructions

The MMX instruction set provides variants of the arithmetic and logical instructions to operate on packed data types and also instructions specific to packed data types. The instruction set was expanded with SSE and SSE2 extensions to operate on new data types and extend new functionality to MMX data types. In the interest of space, we limit the discussion to the instruction that were used in the optimizations. Instructions that are used in optimizing in specific portions of the code are discussed in Section 4.

3. OVERVIEW OF THE TRANSCODING ARCHITECTURES

We consider three transcoding architectures for MPEG-2 to MPEG-4 transcoding. All the three architectures perform spatial and temporal resolution reduction on the input MPEG-2 bitstream. The MPEG-4 output can be configured to be one-half or one-fourth of the spatial resolution of the input MPEG-2 video. The temporal resolution of the output can be configured to 10, 4, or 2 frames per second; the actual output frame rate depends on the GOP structure of the MPEG-2 input. The architectures under consideration are shown in Figures 2-4. Figure 2 illustrates the Cascaded architecture,

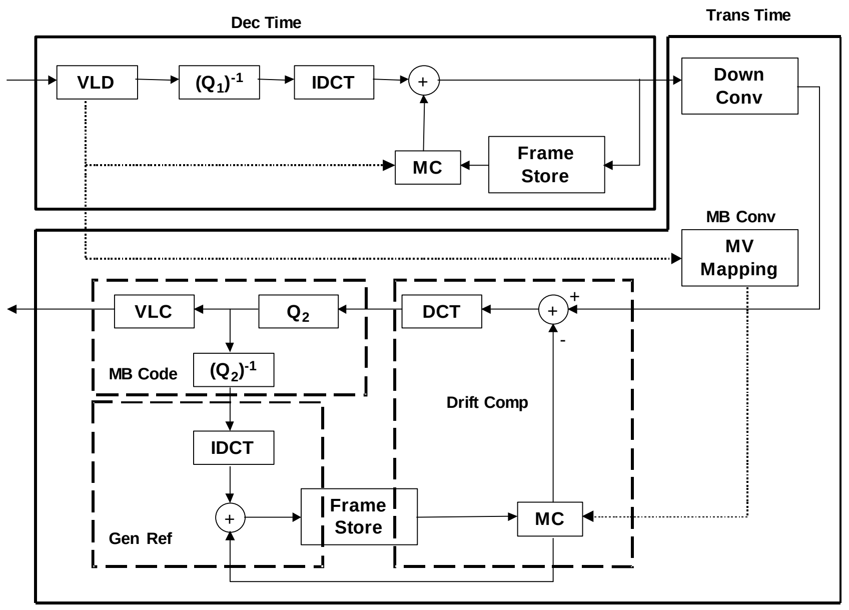

Figure 2. Reference architecture for reduced spatial-resolution transcoding

which is simply a cascaded approach that decodes, down-samples and re-encodes the video. Figure 3 shows the Intra Refresh architecture, which compensates for various errors by converting select macroblocks to intra-coded blocks. Figure 4 shows the Partial Encoder architecture, which is similar to the Cascaded architecture, but simplifies the re-encoding process by not compensating for re-quantization errors. The background regarding the development of these architectures can be found in [2], but a brief description of the Intra Refresh and Partial Encode architectures shown in Figures 3 and 4 is included below for completeness.

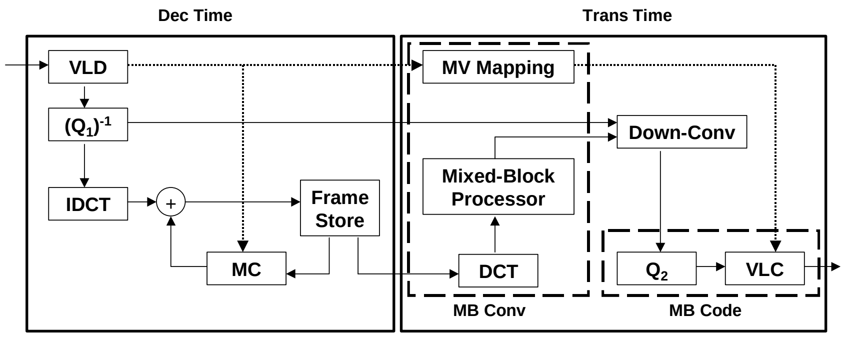

Figure 3 Intra Refresh architecture for reduced spatial-resolution transcoding

In reduced resolution transcoding, drift error is caused by many factors, such as requantization, motion vector truncation and down-sampling. Such errors can only propagate through inter-coded blocks. By converting some percentage of inter-coded blocks to intra-coded blocks, drift propagation can be controlled. In the past, the concept of intra-refresh has successfully been applied to error-resilience coding schemes [10], and we have found that the same principle is also very useful for reducing the drift in a transcoder. The Intra Refresh architecture shown in Figure 3 is based on this concept.

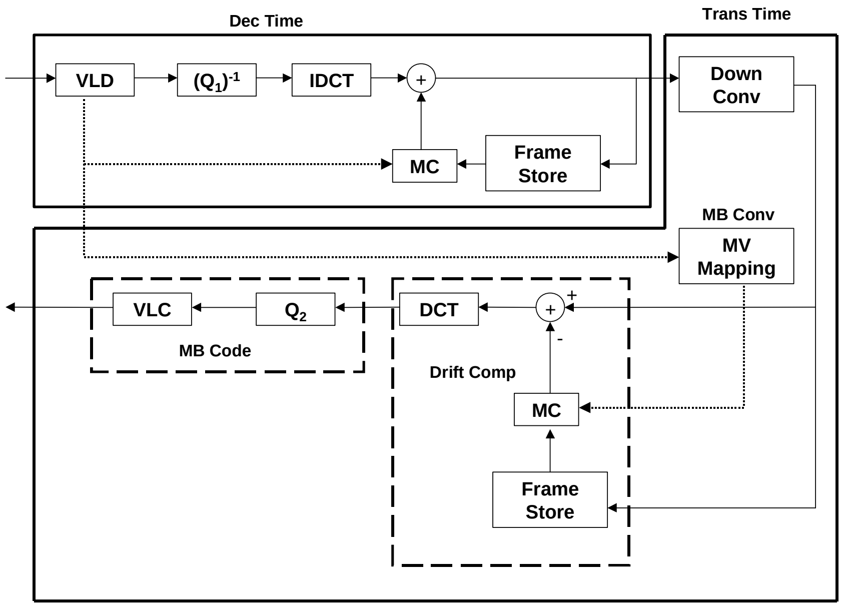

Figure 4 Partial Encode architecture for reduced spatial-resolution transcoding

In the Intra Refresh scheme, output macroblocks are subject to a DCT-domain down-conversion, requantization and variable-length coding. Output macro-blocks are either derived directly from the input bitstream, i.e., after variable-length decoding and inverse quantization, or retrieved from the frame store and subject to a DCT operation. Output blocks that originate from the frame store are independent of other data, hence coded as intra blocks; there is no picture drift associated with these blocks.

The decision to code an intra-block from the frame store depends on the macroblock coding modes and picture statistics. In a first case based on the coding mode, an output macroblock corresponds to four input macroblocks for size conversion by a factor of two in each direction. Since all sub-blocks must be coded with the same mode, the transcoder must avoid having *mixed-blocks*, i.e., inter-coded and intra-coded sub-blocks in the same output macroblock. This is detected by the mixed-block processor, which will trigger the output macroblock to be intra-coded. In a second case based on picture statistics, the motion vector and residual data are used to detect blocks that are likely to contribute to larger drift error. For this case, picture quality can be maintained by employing an intra-coded block in its place. Of course, the increase in the number of intra-blocks must be compensated for by the rate control. Please refer to [2] for details on the operation of the rate control function.

As an alternative to the Reference architecture, the Partial Encode architecture is considered in Figure 4. This architecture aims to eliminate the drift error due to down-sampling and motion vector scaling; drift error caused by requantization is neglected. It operates under the assumption that the error due to requantization in a reduced resolution transcoder is much less than the error due to down-sampling and motion vector scaling.

In the Reference architecture, the reconstructed reference frame used for re-encoding consists of two parts, the low-resolution motion compensated prediction and the reconstructed low-resolution residual. In contrast to this Reference architecture, the Partial Encode architecture essentially removes the feedback components (inverse quantization and inverse DCT) that contribute the residual component to the reconstructed reference frame.

4. TRANSCODER DESIGN AND OPTIMIZATION

The MPEG-2 to MPEG-4 video transcoder performs spatial and temporal resolution reduction of the MPEG-2 video input and produces an output bitstream compliant to the MPEG-4 video bitstream syntax. Figure 5 shows the high-level system diagram. The transcoder core accepts one frame of MPEG-2 video data at a time and transcodes it to MPEG-4 format. The transcoder drops all the input 'B' frames resulting in an output stream with reduced temporal resolution. The application sets the transcoder configuration parameters such as the transcoding architecture, the output bitrate, and the output resolution. The transcoder core has a well-defined interface that accepts single MPEG-2 video frames as input and outputs at most one MPEG-4 video frame. This allows applications to support different MPEG-2 sources such as MPEG-2 transport streams or program streams.

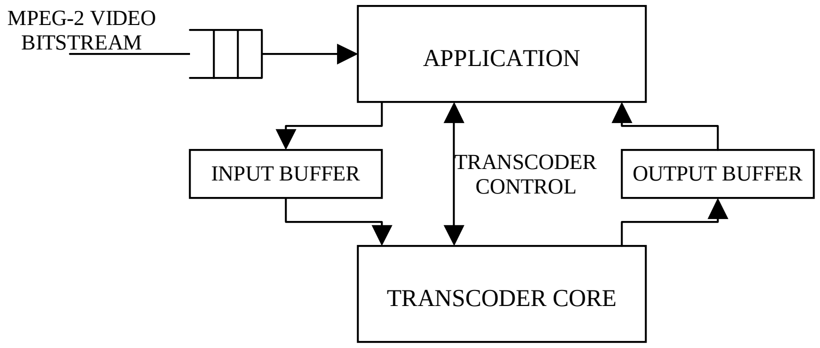

Figure 5. MPEG-2 to MPEG-4 Video Transcoder

The design goals were: 1) realtime transcoding, 2) easy integration in applications, and 3) live MPEG-2 transcoding. The original (unoptimized) software was based on the public domain MPEG-2 decoder from the MPEG Software Simulation Group and an internally developed MPEG-4 transcoder. As can be seen from the Figures 2-4, the MPEG-2 decoding and the MPEG-4 transcoding have a few common components such as IDCT. In addition to that, the commonalities include several block processing routines for operations such as clipping, copying, zeroing, adding, and type converting 8x8 blocks. Using a common set of functions for these operations in both the MPEG-2 decoder and the MPEG-4 transcoder reduces code duplication and allows optimizations across the components. The optimizations used to speedup the transcoder include: 1) MMX optimizations to speedup data parallel portions of the code, 2) general software optimization techniques such as loop unrolling, strength reduction, reducing jumps and conditionals, and 3) compiler optimizations using Intel C++ compiler. In general, MMX optimization possibilities exist wherever there are blocks of data operated on in loops. The following subsections briefly discuss optimization of some of the processes in the transcoder.

4.1 Optimizing Common Block Processing Operations

The following is a list of the MMX optimized common block processing operations used in the MPEG-2 decoding and the MPEG-4 transcoding process:
ZeroBlock – function to set the elements of a block of data to 0
CopyBlock – copy a block of data from source to destination
AddBlock – add a source block to a destination block and save the result in the destination
SubBlock – subtract a source block from a destination block and save the results in the destination
SumBlock – compute the sum of elements of the block and return the sum
ClipBlockXtoY – clip the elements of a block to the range [X, Y]
CopyBlockShortToInt – type convert a block of 32-bit integers in the source to 16-bit short integers and copy to the destination
These set of functions use SIMD instructions and where necessary variants of a function are used to optimally process blocks of different data types or blocks of different size.

4.2 Optimizing IDCT and FDCT

The forward and inverse discrete cosine transforms (DCT) are the most compute-intensive portions of the transcoding process. While FDCT is used in MPEG-4 transcoding, IDCT is used in MPEG-2 decoding and also in reference picture generation in the Cascaded architecture of MPEG-4 transcoding. The SSE extensions allow computation of DCT with IEEE precession using integer instructions. We adopted the fast and precise DCT implementation discussed in [8].

4.3 Optimizing Motion Compensation

Motion compensation is performed in the decoding process where the predicted block is computed based on the decoded motion vectors and then added to the current block to form a reconstructed block. The block prediction involves averaging pixels values and the SSE instruction set includes an instruction for SIMD averaging of 8-bit and 16-bit integers. The PAVGB instruction computes the average of the unsigned integers in the source operand and the destination operand and saves the result in the destination operand. To compute the average, the corresponding 8-bit integers are added, then a 1 is added to the sum, and the result is shifted to the right by one-bit position. The instruction can compute the average of either 8 or 16 bytes depending on whether MMX or XMM registers are used.

The second part of motion compensation is adding the prediction to the decoded coefficients and clipping the result to the range [0, 255]. The prediction computed is stored as unsigned character and the decoded coefficients are stored as signed short integers. To perform the SIMD addition, the 8-bit integers are converted to 16-bit integers and the add and clip operations are performed on the 16-bit integers. The PUNPCKHBW instruction convert bytes into words by copying and interleaving the high order data elements of the source and destination operands. The lower order elements are unchanged. Similarly, PUNPCKLBW instruction converts lower order bytes to words. The PUNPCKLBW instruction is illustrated in Figure 6. This SIMD type conversion is followed by a PADDW instruction that adds packed word integers and a clip operation using PMINSW and PMAXSW. The PMINSW instruction compares the signed short integers in the source and destination operands and returns the minimum value to the destination. Similarly, PMAXSW computes the maximum. A PMINSW operation with the upper bound of the clip range and a PMAXSW operation with the lower bound of the clip range clips the values to the range resulting in a reconstructed block.

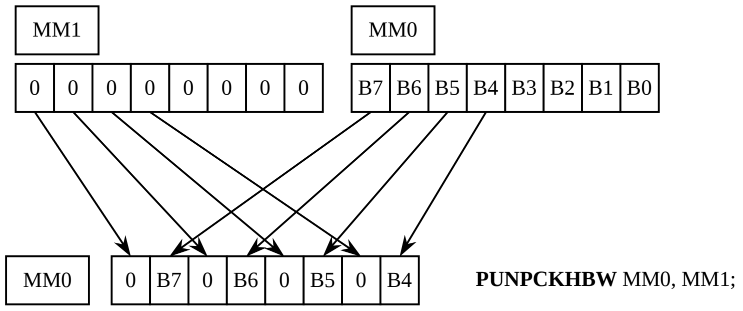

Figure 6. Illustration of SIMD type conversion from byte to word

4.4 Optimizing Mismatch Control

To minimize the error accumulation due to IDCT mismatch at the MPEG-2 encoder and decoder, mismatch control is performed before IDCT. This involves clipping the decoded coefficients to the range [-2048, 2047], summing the block, and if the sum is even, mismatch control is applied to the last coefficient of the block. If the last coefficient is odd, it is decremented by 1, and if even incremented by 1. This is a compute intensive process and is applied to every decoded block. This process is optimized by using PMINSW and PMAXSW to clip coefficients to the range and PADDW and PUNPCK* instructions to compute the sum of the coefficients.

4.5 Optimizing DCT Domain Spatial Resolution Reduction

The Intra Refresh architecture performs spatial resolution reduction in the DCT domain. Complexity reduction in DCT domain spatial reduction is discussed in [1]. The DCT domain down sampling takes significant computation but has some room for optimization. The symmetry of the down conversion filters and the presence of zeros in the filter was exploited to significantly reduce the number of multiplications necessary. The filters were converted to integer filters by multiplying with a constant and then shifting the results appropriately thus converting floating-point multiplications into integer multiplications. MMX instructions were then used to parallelize the simplified down conversion loops.

4.6 Optimizing Quantization

Quantization is the second most compute-intensive portion of the transcoding after DCT. Computation of quantized coefficients for the inter blocks is shown below:

```
level = (ABS(coeff[i])-QP/2)  / (2*QP);
qcoeff[i] =  MIN(2047,MAX(-2048,SIGN(coeff[i]) * level));
```

This computation is performed for every pixel of a video frame. This computation is first simplified as:

```
slevel = coeff[i]*(1.0/(2*QP))- SIGN(coeff[i])*(0.25);
qcoeff[i] =  MIN(2047,MAX(-2048, slevel));
```

In the computation of the level, the integer division is replaced by floating point multiplication, as it is more efficient than division. The coefficients which are stored as short integers are first converted to 32-bit integers using unpack instructions and then converted to 32 bit floating point values using CVTDQ2PS instruction that converts four integers in an XMM registers to four single precession floating point values. A SIMD multiplication of single precession floating point values (coeff[i]*(1.0/(2*QP))) is supported by the MULPS instruction. The final value of the level is computed by adding or subtracting 0.25 depending on the sign of the coefficients. Such conditional additions are efficiently computed by first computing a sign mask using a compare instruction, CMPPS, and then using ANDPS and ANDNPS to conditionally add the value. While ANDPS performs a bit wise logical AND of the four single precision floating point values in the source operand and the destination operand, ANDNPS negates the bits of the destination before the bitwise AND. The following example illustrates the use of these instructions:

Suppose that register XMM3 is loaded with the coefficients multiplied by 1/(2*QP).

```
MOVAPS XMM0, [one_y_4_f]; // float one_y_4_f [] = {0.25, 0.25, 0.25, 0.25}
```

Create a sign mask of the coefficients:

```
PXOR XMM4, XMM4; // set XMM4 to 0
CMPPS XMM3, XMM4; // is coefficient > 0, is so the corresponding bits in XMM3 are set to 1 else 0
MOVAPS XMM4, XMM3; // copy mast to XMM4
```

Conditionally add +0.25 or –0.25:

```
ANDPS XMM3,  [neg_one_y_4_f]; // float neg_one_y_4_f [] = {-0.25, -0.25, -0.25, -0.25}
ANDNPS XMM4, [one_y_4_f]; // float one_y_4_f [] = {0.25, 0.25, 0.25, 0.25}
ADDPS XMM3, XMM4; // conditional value if now in XMM3
```

5. RESULTS

The results of the transcoder performance optimization are discussed in this section. The evaluation platform is a Dell workstation with Pentium-4/1.8 GHz processor, 512MB memory, and running Windows 2000. The transcoder has been built and tested using the Microsoft Visual Studio 6.0 with Microsoft C++ compiler and Intel C++ compiler 7.0. For the purpose of this evaluation, a 30 second MPEG-2 video clip with a resolution of 720x480, encoded at 5 Mbps, and 30 FPS is used. The transcoder is configured to produce an MPEG-4 bitstream with a resolution of 352x240, encoded at 384Kpbs and 10 fps. The *Partial Encode* transcoding architecture is used for evaluation. The time is measured using the Windows APIs for QuertHighPerformanceFrequency() and QueryHighPerformanceCounter(). All the times are given in seconds. Table 1 summarizes the performance results.

Table 1: Transcoder Performance Results

Case	Total Time	Decoding	Transcoding	Down Conv.	MB Coding	Drift Comp
1	39.07	7.89	31.18	9.07	4.92	16.23
2	21.4	7.25	14.15	4.58	4.83	3.69
3	30.81	7.9	22.89	0.59	4.25	17.89
4	12.0	5.31	6.66	0.43	3.69	2.41
5	6.8	3.21	3.53	0.38	2.21	0.83

The optimizations used in the four cases listed in the table are:

Case1: unoptimized, Microsoft Visual C++ Compiler
Case 2: unoptimized Intel C++ compiler
Case 3: software optimizations (without written MMX code), Microsoft C++ compiler
Case 4: software optimizations (without written MMX code), Intel C++ compiler
Case 5: software and MMX optimizations and Intel Compiler.

Table 1 shows the total time, time for MPEG-2 decoding, time for MPEG-4 transcoding, and times for the important components of the transcoding time: down conversion, MB coding, and drift compensation. The following sub sections briefly discuss each of these cases.

5.1 Unoptimized Code With Microsoft C++ Compiler

Case 1 shows the times using unoptimized code and the standard C++ compiler that comes with Microsoft Visualstudio 6.0. The total transcoding time is significantly more than the realtime of 30 seconds, the length of the input MPEG-2 clip. The MPEG-2 decoding potion of the code is well optimized and has little room for general software optimizations. The transcoding portions on the other hand have room for significant improvements. All the code was written in standard C. The transcoding time is dominated by down conversion and drift compensation. The down conversion uses a complex filter for horizontal and vertical down conversion of the decoded MPEG-2 signal in the spatial domain. The drift compensation is dominated by residual prediction and DCT computation.

5.2 Unoptimized Code With Intel C++ Compiler

Case 2 shows the times using the unoptimized code and the Intel C++ compiler 7.0. The code is just recompiled using the Intel C++ compiler and optimizations enabled are the default optimizations to maximize speed (optimization option o2). Global optimizations, inline expansion, and code generation for linker optimizations are also enabled. The total transcoding time went down by 45%, a significant improvement. The MPEG-2 decoding is not significantly affected. Most of the gains are from the transcoding portions of the code. Intel compiler generates parallel code when possible resulting in significant improvements when data parallel computations are involved.

5.3 General Software Optimizations With Microsoft C++ Compiler

Case 3 shows the times using the code with general software optimizations and the Microsoft C++ compiler 7.0. The general optimizations performed include loop unrolling, replacing multiplications with shifts, using smaller data types where possible (e.g., replace int by short), using common block processing code across the MPEG-2 decoding and MPEG-4 transcoding portions of the transcoder, and eliminating dead code. The code was also wrapped in C++ classes for ease of use and extensibility. These changes did not result in significant gains but prepared code for better MMX optimizations and parallelization. The reduction in down conversion time is mainly because of using a simple averaging filter. It is also possible that there is some overhead because of the C++ classes used.

Another important optimization is optimizing memory access. Memory access is significantly impacted by memory alignment. Access to data that is not aligned to a 64-byte boundary requires two memory accesses and several µops to be executed instead of one [9]. To avoid memory access overhead, all memory was allocated on 16-byte boundary. Intel runtime library provides a memory allocation routine, mm_malloc, to allocated aligned block of memory. Specialized data movement instructions (e.g., MOVDQA) are provided to load and store XMM registers and require data alignment on 16-byte boundary. Using unaligned memory with these instructions would result in a general protection exception.

5.4 General Software Optimizations With Intel C++ Compiler

Case 4 shows the times using the code with general software optimizations and the Intel C++ compiler 7.0. The additional Intel C++ compiler options enabled in this evaluation are: optimizations targeting Pentium-4 processors, inter procedural optimizations, inline expansion, and multifile optimizations. As evident from the times observed, the software optimizations such as type conversion have significant impact when using the Intel Compiler. The general optimizations performed are not Intel compiler specific but the compiler is able to generate a better code with these optimizations. The total trancoding time went down by over 44% compared with unoptimized code using a Intel C++ compiler. The MPEG-2 decoding portions went down by over 26% and the transcoding portions, without considering the down conversion, went down by 35%.

5.5 MMX Optimizations With Intel C++ Compiler

Finally, case 5 shows the times with additional MMX optimizations. These optimizations resulted in significant improvements in the transcoding time. Compared with case 4, the total transcoder time went down by 43% with the MPEG-2 decoding time going down by 39% and the transcoding time going down by 48%. Significant gains of 65% were observed in drift compensation mainly due to optimized DCT computation. The gains in the MB Coding process were dominated by the optimized quantization.

CONCLUSION

Using a combination of optimization techniques we were able to achieve significant performance improvements in the MPEG-2 to MPEG-4 video transcoding. The total transcoding time was reduced by over 82% making the realtime transcoding possibility. The optimized transcoder can transcode four MPEG-2 video streams simultaneously. These results highlight the importance of software and compiler support to maximize the benefits offered by the advanced processor architectures. With advanced CPUs and appropriate optimizations, compute-intensive video processing tasks that once required dedicated hardware can now be accomplished in software.

REFERENCES

1. A. Vetro, T. Hata, N. Kuwahara, H. Kalva and S. Sekiguchi, “Complexity-quality evaluation of transcoding architecture for reduced spatial resolution,” *IEEE Transactions on Consumer Electronics*,” Aug. 2002.
2. P. Yin, A. Vetro, H. Sun and B. Liu, “Drift compensation architectures for reduced resolution transcoding,” Proc. SPIE Conf. on Visual Communications Image Processing, San Jose, CA, Jan. 2001.
3. Hari Kalva et. al., "Parallel JPEG Image Compression Algorithm for DEC-Maspar," Technical Report No. TR-CSE-94-5, Dept. of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL.
4. V. Lappalainen, “Performance Analysis of Intel MMX Technology for an H.263 Video Encoder,” Proceedings of ACM Multimedia Conference, 1998, pp. 309-314.
5. http://www.simdtech.org
6. Intel Corp., “MMX Technology Technical Overview,” http://developer.intel.com.
7. Intel Corp., “IA-32 Intel Architecture Software Developer’s Manual Volume 1: Basic Architecture,” Order Number 245470, http://developer.intel.com.
8. Intel Corp., “A Fast Precise Implementation of 8x8 Discrete Cosine Transform Using the Streaming SIMD Extensions and MMX instructions,” AP-922, Order Number 742474-001.
9. Intel Corp., “Intel Pentium 4 and Intel Xeon Processor Optimization Manual,” Order number 248966-05.
10. K. Stuhlmuller, N. Farber, M. Link and B. Girod, “Analysis of Video Transmission over Lossy Channels," *J. Select Areas of Commun*ications, June 2000.