EURASIP Journal on Applied Signal Processing 2001:2, 100–109
© 2001 Hindawi Publishing Corporation

Software Optimization of Video Codecs on Pentium Processor with MMX Technology

Pohsiang Hsu

Microsoft Corporation, One Microsoft Way, 25/2481, Redmond, WA 98052-6399, USA
Email: pohhsu@microsoft.com

K. J. Ray Liu

Department of Electrical and Computer Engineering, University of Maryland, College Park, USA
Email: kjrliu@eng.umd.edu

Received 14 March 2001 and in revised form 3 May 2001

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) 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. The described codec is adopted for video mail and video phone softwares used in IBM ThinkPad.

Keywords and phrases: video coding, MMX, software optimization.

1. INTRODUCTION

Recent advances in the personal computer (PC) 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 examples include the popular software implementations of JPEG standard used to exchange images over the Internet, MPEG1 and MPEG2 decoders for music videos and movies stored in CD-ROM and DVD-ROM, respectively, and the H.263 standard for video conferencing and video telephony. These software tools facilitate the introduction of desktop video, video based interactive multimedia services, and desktop video conferencing to the general publics with a PC. Some advantages of implementing the video codec (COder/DECoder) 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.

Due to the fact that a great deal of parallelism exists in many DSP algorithms including video encoding and decoding, many microprocessors have been designed to support dedicated hardware to exploit the parallelism. These special hardware extensions give microprocessors the ability to process multiple data with the same operation efficiently through the addition of an SIMD (single instruction multiple data) instruction set. This feature is often referred to as subword parallelism, pack-arithmetic, or multimedia extension [1]. Indeed, many widely used general-purpose microprocessor platforms are adopting various forms of this concept. Some examples are the MMX instructions [2, 3] from Intel, VIS instructions from Sun, and PA-RISC multimedia extensions from HP.

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. Sometimes, we may have the case where algorithm A has greater number of operations than algorithm B while the implementation of algorithm A is faster than the implementation of algorithm B on a hardware platform. This is because each microprocessor has its 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. Specifically, the selection of a component should not always be the one with the lowest number of operations but rather the one that takes the least number of cycles to execute on the given hardware platform.

In this paper, we consider the problem of video codec optimization on the Intel Pentium with MMX technology processor, which powers a vast majority of the computers in the world. MMX technology is an extension to the Intel architecture and it provides fifty seven powerful SIMD instructions aimed to aid the exploitation of parallelism inherent in many multimedia and digital signal processing applications. By taking advantage of the strengths of the hardware, we present a fast software implementation of an H.263 video encoder. 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 speedup. Optimization techniques such as removal 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 speedup through software pipelining.

This paper is organized as follows. In Section 2, we briefly describe the Intel Pentium with MMX technology processor and point out some features that are relevant to software optimization. In Section 3, we describe the design principle of the MMX technology. In Section 4, we describe the type of instructions that the MMX technology provided. In Section 5, we analyze the computational requirements of the H.263 video encoder. In Section 6, we describe a fast implementation of the H.263 video encoder. Lastly, we conclude in Section 7.

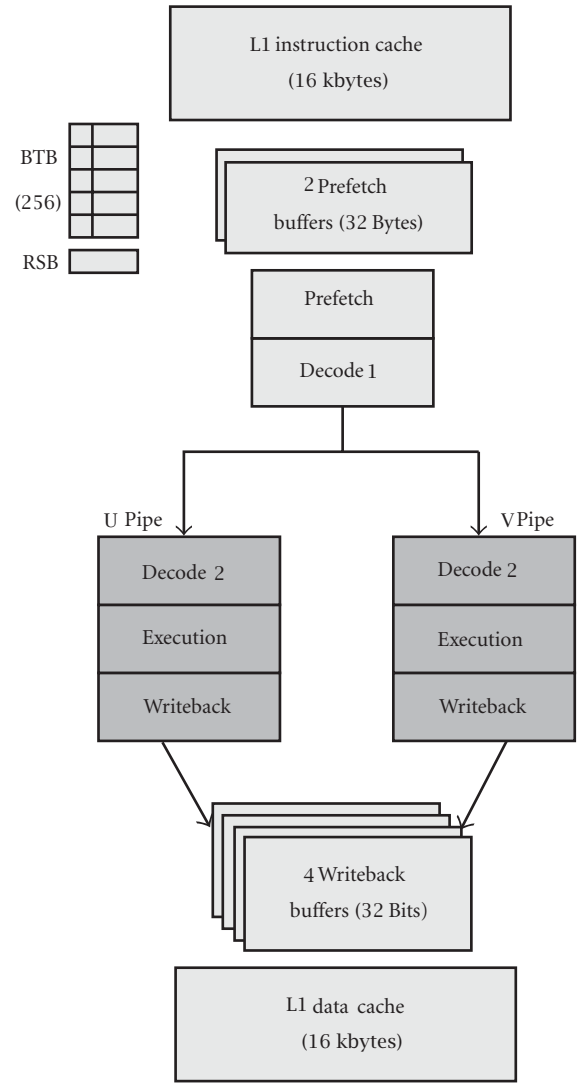

Figure 1: Architecture view of the Pentium with MMX technology processor.

2. INTEL PENTIUM WITH MMX TECHNOLOGY PROCESSOR

The Intel Pentium with MMX technology processor is an advanced superscalar processor. An overview of the processor's architecture is shown in Figure 1. The key components of the processor are a five stage pipelined architecture with dual execution pipelines (U and V pipe), separate instruction and data L1 caches, four shared write buffers, instructions prefetching unit, a branch target buffer (BTB), and a return to stack buffer (RSB).

One salient feature of the processor is the ability to execute up to two integer instructions every clock cycle using the U and V pipes. However, the high execution rate can only be sustained if certain rules and restrictions are followed in scheduling the instructions. These rules and restrictions are of a direct consequence of the underlying architecture of the processor. Therefore, an understanding of the main components of the processor is paramount in software optimization. In this section, we describe the main components of the processor and discuss its relevance to high performance coding style.

The Pentium with MMX processor has two 16 Kilobytes L1 caches. The Level 1 (L1) cache is a small piece of memory located on chip that is very fast, typical ten times faster than main memory. One cache (instruction cache) is reserved for fetched instructions and the other cache (data cache) is reserved for data accesses. The two caches are independent with its own internal 64-bit bus so that the processor can load instruction and data in the same clock cycle. Each cache is organized into 32-byte chunks called cache lines. The cache line is the minimum unit for data transfer between the external bus (e.g., memory) and the cache. On a read or write hit to the L1 cache, the request can be done in just one clock cycle. However, the processor bursts an entire cache line into the L1 cache when a read miss occurs and writes the data directly to the main memory.

The cache's purpose is to exploit the temporal and spatial locality found in typical code [4]. Temporal locality is based on the idea that if an item is referenced, it is highly probable that it will be referenced again soon. Spatial locality refers to the idea that if an item is referenced, it is highly likely that nearby items will be referenced. By requesting a cache line at a time, neighboring items can be accessed in a fast manor.

Since data transfer from the L1 cache to the processor is much faster than from main memory, a key to construct good code will be to organize it in such a way that data are reused as often as possible when it is in the L1 cache. First, it is important to make sure that the size of the inner loops is below 16 KB so that it can fit inside the L1 cache or the L1 cache will thrash continuously. Second, the code should be organized so that operations on the same set of data are grouped together. This ensures reuse of the data while it is still in the data cache. Lastly, the technique of allocation of data ahead of time can be used to minimize the wait time.

The Pentium processor has two integer execution pipelines called the U and the V pipes as shown in the center of Figure 1. In the ideal case, two instructions can be executed at the same time, one in each pipe, per processor cycle. Each pipeline is divided into five execution stages that allows overlapped execution of different instructions at one time. The five stages are called the PF (prefetch) stage, the D1 (decode 1) stage, the D2 stage (decode 2), the E stage (execution), and the WB (writeback) stage. In the PF stage, the instructions are prefetched, parsed, and pushed into an instruction FIFO, which is located before the D1 stage. In the D1 stage, two instructions from the instruction FIFO are examined to determine whether they are pairable under current situation. Either one or two instructions are pulled from the instructions FIFO depending on whether the two instructions are pairable or not. In the D2 stage, the addresses of the memory operands are calculated. The instruction is executed in the E stage and the results are written back in the WB stage. To take advantage of the dual pipelined execution of the Pentium processor, the assembly code must be carefully scheduled to follow a set of pairing rules. The pairing rules describe situations where the pipeline cannot be maintained at maximum capacity due to data dependency, register contention, and other restrictions imposed by the Pentium processor. Therefore, it is important to schedule your assembly code to follow the pairing rules so that the maximum throughput can be achieved. One useful optimization technique for transforming an inner loop that does not pair well into an improved loop with better pairing is software pipelining [5, 6]. The basic idea is to partially unroll the loop by making several copies and interleave them to achieve better packing in the execution pipelines.

3. 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. These new instructions allow an application to exploit certain types of parallelism that exists in many applications. It was first implemented on the Pentium processor and has been added to the Pentium II processor as well. The MMX technology was designed with the following goals in mind. First, it is designed to significantly improve the performance of multimedia, communications, and other compute intensive applications. Second, the addition of the MMX extension must provide full backward compatibility, which means that applications written for older Intel processors must still be able to run. Third, the architecture should be able to be scaled to keep pace with future Intel processors. Lastly, the new instruction set provided by the MMX technology will be composed of general-purpose instructions only.

The general-purpose MMX instruction set was designed by analyzing a broad range of software applications in the field of multimedia and communications such as computer graphics (2D and 3D), image processing, music synthesis, speech compression, image compression, video compression, and video conferencing. 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. First, it is found that operations were typically performed on small native data types such as 8-bit pixels for image/video applications and 16-bit audio samples for speech and music applications. Second, the memory access patterns were found to be regular, recurring, and usually data independent. Lastly, the computations on the data are typically localized and recurring. From 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.

For example, a common operation in image processing and video coding is the pixel-wise addition of two equally sized images. In this operation, the pixels located on corresponding spatial locations from the two images are added and clipped to form the resulting image. The basic underlying computation of this process is the addition of two integer arrays where each element is of a fixed size. We consider the case of adding two integer arrays A and B together where each array has four short integers. Normally, the native instructions of the microprocessor, such as the Pentium processor, is designed under the single instruction single data (SISD) architecture. This means that the microprocessor can only operate on one set of data at a time. Therefore, the array elements have to be added individually to form the result in a sequential manner as shown in Figure 2. On the other hand, SIMD instructions are designed to operate on multiple data elements at a time. The main difference between SISD instructions and SIMD instructions is that SISD instructions operate on individual elements at a time while SIMD instructions operate on an array of elements at a time. As shown in Figure 3, the four additions are done at the same time in parallel. Therefore, we can see the benefit of having such an SIMD instruction.

The eight new 64-bit general-purpose registers are logi-

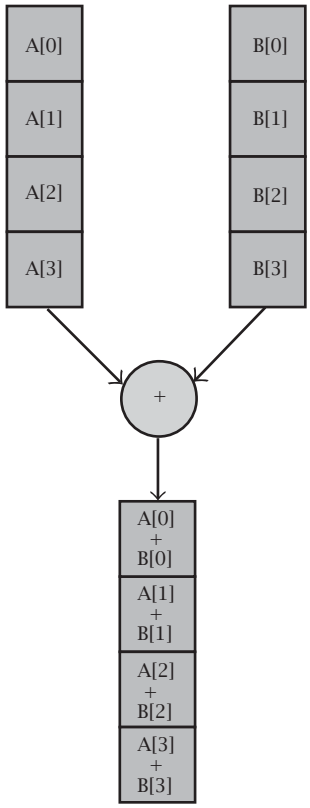

Figure 2: Scalar instruction scalar data (SISD).

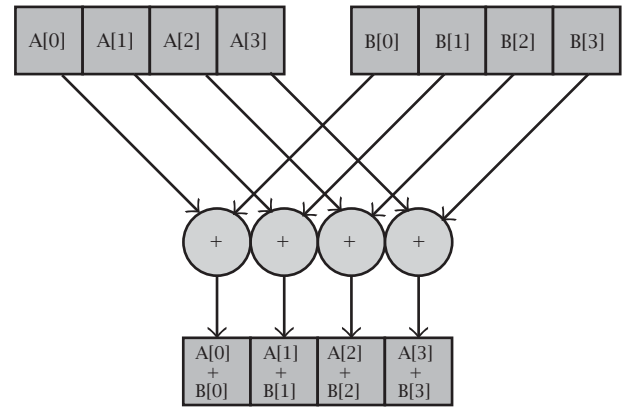

Figure 3: Single instruction multiple data (SIMD).

cally defined by the MMX technology only. In actuality, these eight registers are mapped onto the eight 80-bit floating-point registers on the Pentium processor. The reason for this implementation is to achieve full compatibility with existing operating systems and applications. Current operating systems save and restore the contents of the integer and floating-point registers between context switching. Therefore, if a new set of registers were defined for MMX then the existing operating systems must be modified in order to save the information on the new registers. By alias the MMX registers on top of the floating-point registers, it is ensured that the contents of the MMX registers are saved by existing operating systems during context switching. Since the MMX registers and the floating-point registers are physically the same, the registers must be reset when switching from floating-point instructions to MMX instructions and vice versa. This brings us to the point that the code should be partitioned in such a way that there are no intermixing of MMX instructions and floating-point instructions on the instruction level to avoid frequent reset of the registers. However, it is acceptable to mix MMX instructions and floating-point instructions on the procedural level.

4. MMX TECHNOLOGY INSTRUCTIONS

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 as shown in Figure 4. All four new data types are 64-bit wide and fit nicely inside an MMX register. 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 contains eight bytes, packed word contains four words (a word is a 16-bit quantity), and packed doubleword contains two doublewords. The data inside an MMX register is interpreted as one of these four new types depending on the executed instruction. The MMX instruction sets include fifty seven new instructions to perform various operations on these four new data types. New instructions introduced by the MMX technology includes packed arithmetic

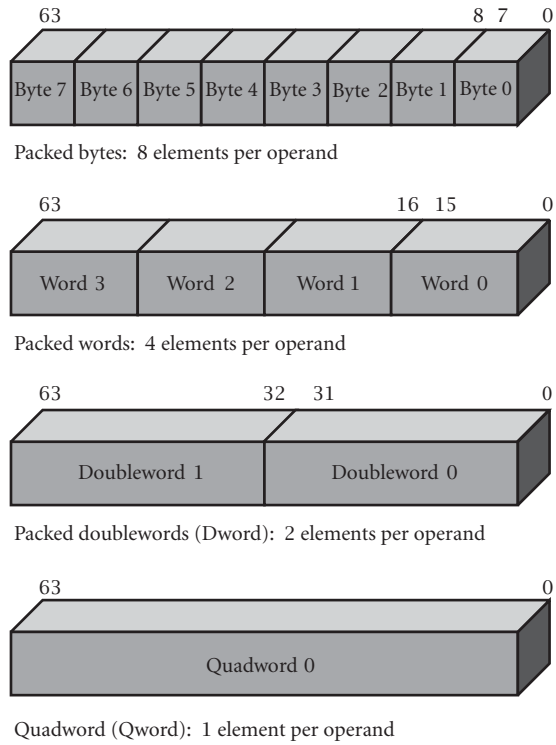

Figure 4: Input data formats for MMX technology.

Table 1: Summary of MMX instruction set.

Instruction Type	Instruction	Description
Arithmetic	P[ADD,SUB][B,W,D]	Add/subtract with wraparound
	P[ADD,SUB]S[B,W]	Add/subtract signed with saturation
	P[ADD,SUB]US[B,W]	Add/subtract unsigned with saturation
	PMUL[H,L]W	Multiply four words and store high/low 32-bit result in register
	PMADDWd	Packed multiply and add
Comparison	PCMPEQ[B,W,D]	Compare if equal
	PCMPGT[B,W,D]	Compare if greater than
Conversion	PACKSSWb	Convert signed word/dword to signed
	PACKSSDw	byte/word using signed saturation
	PACKUSWb	Convert signed word to signed byte using signed saturation
	PUNPCK[H,L]Bw	Interleave the high/low order 32-bit data
	PUNPCK[H,L]Wd	elements of the source and destination
	PUNPCK[H,L]Dq	operands across data type boundary
Logical	PAND, PANDN	Bitwise logical AND, AND NOT
	POR, PXOR	Bitwise logical OR, XOR
Shift	PS[R,L]L[W,D,Q]	Shift right/left logical without carry across data type boundary
	PSRA[W,D]	arithmetic shift right
Data transfer	MOV[D,Q]	Transfers 32/64 bits between MMX register and integer register or memory
EMMS	EMMS	Empty MMX technology state and clears FP tag word

instructions, saturating arithmetic instructions, data manipulation instructions, and logical instructions. The set of new instructions introduced by MMX is summarized in Table 1.

The packed arithmetic allows the same arithmetic operations to be applied to each individual data element of a packed data type in parallel. For example, adding two packed bytes together using PADDB will perform eight additions, one for each byte, in parallel. Similarly, four and two arithmetic operations are performed when working with packed words and packed doubleword, respectively. 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 efficiently to perform insertion, transposition, and other data manipulation operations.

5. GENERAL PROCESSING REQUIREMENTS OF VIDEO CODECS

An analysis of the operations found in many video standards reveals the implementation of many internal modules of a video codec can benefit from using SIMD instructions such as those provided by the MMX technology. From analysis, the following characteristics are typically found in video codecs. First, the input data and coefficients have usually eight to sixteen bits of precision, which is ideal for fixed-point packed data operations. Second, floating-point operations are typically not required. Third, the multiply-accumulate operation, where the multiplication is often performed with constants, is very common. Lastly, many operations require clipping to a predefined range as a final step, which can be efficiently implemented with saturating arithmetic. These observations imply that many operations in a video codec are good candidates for optimization using MMX instructions.

Standards based video codec achieves compression by exploiting the temporal and spatial redundancies inherent in video signals. The main functional blocks of these video cod-

Table 2: Basic arithmetic operations in standard video codecs.

Function	Arithmetic operations
DCT/IDCT	$ax + b$, $\sum c_i x_i$
Quant./Dequant.	x_i/c_i, $x_i c_i$
Motion Est./Comp.	$\sum \lvert x_i - y_i \rvert$, $\sum (x_i - y_i)^2$, $\min(a, b)$, $x_i \pm x_j$
Color transformation	$\sum c_i x_i$, $(x_i + x_j)/2$, $(1/4)\sum_{i=1}^{4} x_i$
Huffman coding	shifts, comparisons

ing standards are the discrete cosine transform (DCT), inverse discrete cosine transform (IDCT), quantization, dequantization, Huffman coding, motion estimation/compensation. In addition, color transformation and various pre-processing and post-processing filtering are an essential part to any actual video communication system. These major functional blocks occupy a significant portion of the computational load. From an examination of these functional blocks, certain basic arithmetic operations are found to form a major portion of each functional block. In Table 2, we list some of the major arithmetic operations found in typical standards based video codecs. In most cases, these operations are independently applied to a number of data elements or pixels and thus can be efficiently done in parallel by SIMD instructions. It is found that certain operations in the DCT/IDCT, quantization/dequantization, motion estimation, motion compensation, and color transformation blocks are very suitable for MMX implementation.

6. H.263 ENCODER IMPLEMENTATION DETAIL

The H.263 video coding standard was established by the International Telecommunication Union (ITU) mainly for providing real-time low bit-rate visual communication over the telephone lines. In addition, it has been used to provide compressed video contents by many Internet sites. For real-time applications such as video telephony using the PCs, the amount of available processing speed becomes an important issue. In order to provide video telephony, each PC must be capable of handling video and audio capture, video and audio encoding, multiplexing, transmission, demultiplexing, video and audio decoding, video display, and audio playback simultaneously. Among these tasks, video encoding is typically the most computationally intensive and occupies a major portion of the computational load in such a system. In this section, we describe a fast implementation of an H.263 video encoder under the Pentium with MMX processor platform.

6.1. Motion estimation

The major computational blocks in an H.263 video encoder are motion estimation, motion compensation, DCT/IDCT, quantization/dequantization, 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

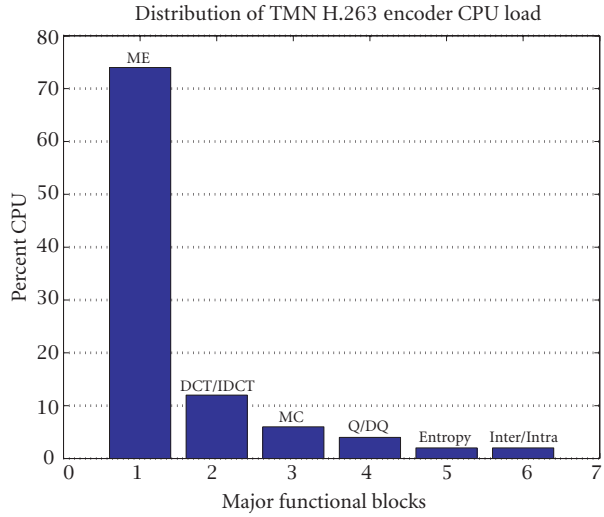

Figure 5: Distribution of CPU load for TMN H.263 video encoder. The abbreviations ME, MC, Q/DQ, stands for motion estimation, motion compensation, and quantization/dequantization, respectively.

from a typical H.263 encoder, we encoded a video sequence using the ITU TMN (test model near-term) 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 Figure 5, 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. The basic idea of block matching is to partition the current image into nonoverlapping blocks and then find the best prediction for each block in the current frame from the previous frame. The search for the best prediction is typically constrained to a search window where a block of the same size is extracted from the previous frame and compared with the original block to obtain the best match as shown in Figure 6. 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 a search is prohibitively high as we can see from Figure 5 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

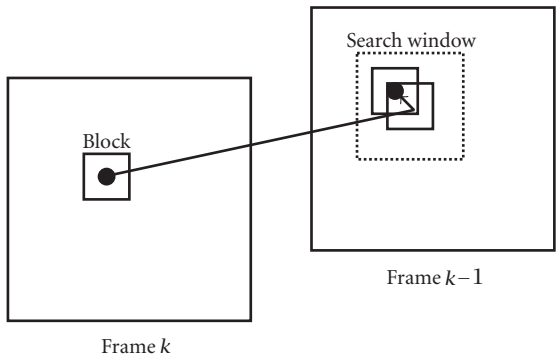

Figure 6: Block matching.

criteria that requires fewer computations, [7], than the sum of absolute difference (SAD) or the mean square error (MSE) are used. Also, sub-sampled approach such as using SAD from a smaller set of pixels as the matching criterion, [8], has been proposed. In fast search, the SAD or MSE criteria is typically still used but the average number of points searched is smaller than the total number of points in the entire search space. The fast search approaches include the 2D log search [9], conjugate direction and one-at-a time search [10], three step search [11], gradient descent search [12], and predictive search [13]. A common theme among these methods is the exploitation of the unimodal error surface assumption which is generally not true. This assumption states that the matching difference is monotonically increasing as a particular vector moves further away from the desired global minimum. However, it seems like a reasonable assumption in a small neighborhood around the global minimum. Therefore, a search procedure using the monotonicity of error surface assumption is in danger of being trapped around a local minima.

In our video encoder, we employed a fast search block matching that is based on the three step search [11]. 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 needed in the search process of future starting points due to overlap. Therefore, the computed matching scores are stored so that they can be accessed 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:

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

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 differences together to form the absolute difference. If x equals to y, then the computation produces the correct result. If x does not equal 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. On the Pentium processor, we have obtained a speedup of about six times over a pure C implementation.

6.2. Motion compensation

The purpose of motion compensation process for the decoder is to generate the prediction image block by block using the previous image and the set of motion vectors. The motion vectors are allowed to take on either integer or half pixel values. For blocks with integer motion vectors, the prediction block is generated by copying the appropriate block indexed by its motion vector from the previous image. On the other hand, the prediction block of blocks with half pixel motion vectors needs to be generated by bilinearly interpolating the appropriate region in the previous image.

On the encoder side, the motion compensation process is closely tied to the motion estimation process. 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 that 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. In these two cases, the predicted pixel Y is obtained from the two pixels X_1 and X_2 by

$$Y = \frac{X_1 + X_2 + 1}{2}. \tag{2}$$

For the third case, we need to perform averaging across the adjacent rows and columns to find the predicted value. In this case, given four pixels X_1, X_2, X_3, and X_4, the predicted pixel Y can be obtained by

$$Y = \frac{X_1 + X_2 + X_3 + X_4 + 2}{4}. \tag{3}$$

For our encoder, we take advantage of the MMX technology to perform the motion compensation process. We 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 cannot 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. We consider the case of averaging two integers X_1 and X_2 to form Y as shown in (2). Suppose we simply perform the following operations to form Y_1:

$$Y_1 = X_1 \gg 1 + X_2 \gg 1, \tag{4}$$

where $\gg$ indicates a right shift. Comparing Y and Y_1 reveals that the following relationship holds:

$$Y = X_1 \gg 1 + X_2 \gg 1 + Z, \tag{5}$$

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 cannot 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.

6.3. DCT and quantization

The H.263 encoder uses DCT to reduce the spatial redundancy of the video sequence. The input to the DCT is either the image or the prediction residual depending on whether the frame is intra-coded or inter-coded. The input is subdivided into eight by eight blocks and transformed using a 2D DCT followed by quantization. For a frame that required motion estimation, its previous quantized frame needs to be reconstructed at the encoder. The reconstruction process simply reverses of the encoding process where the quantized input signal is first dequantized followed by the application of the IDCT. The DCT is popular in image compression because it achieves good energy compaction and it has many fast algorithms available.

Let $s(y,x)$ be the 2D input sample values and $S(v,u)$ be the 2D DCT coefficients, then the mathematical definitions of the eight by eight 2D DCT and its inverse IDCT for the H.263 video standard are defined as follows:

$$\begin{aligned} S(v,u) &= \frac{C(v)}{2}\frac{C(u)}{2}\sum_{y=0}^{7}\sum_{x=0}^{7} s(y,x) \\ &\quad \times \cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right], \\ s(y,x) &= \sum_{v=0}^{7}\frac{C(v)}{2}\sum_{u=0}^{7}\frac{C(u)}{2}S(v,u) \\ &\quad \times \cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right], \end{aligned} \tag{6}$$

where $x, y, u, v = 0, \ldots, 7$ and the normalization factor C is defined as

$$C(x) = \begin{cases} \dfrac{1}{\sqrt{2}}, & x = 0, \\ 1, & x > 0. \end{cases} \tag{7}$$

An important property of the 2D DCT/IDCT is its separability. This implies that the transform can be computed by applying 1D DCT/IDCT on each row and then applying 1D DCT/IDCT on the columns of the results or vice versa. A quick examination of the above equations reveals that direct computation of the 2D DCT/IDCT using 1D DCT/IDCT requires 1024 multiplications and 896 additions, which is a prohibitively large number of operations. Fortunately, many

fast algorithms have been proposed for efficient computation of the 2D DCT/IDCT in the literature [14, 15]. There are three main approaches in fast 2D DCT/IDCT computations namely 1D based, 2D based, and statistical based. The 1D based approaches compute the 2D DCT/IDCT using fast 1D DCT/IDCT.

We note that given any fast DCT algorithm, a fast IDCT with the same computational complexity can be easily constructed. A fast IDCT can be obtained by simply reversing the flow graph of a fast DCT. This is because the DCT/IDCT is an orthonormal transform, which implies that the inverse of the transform is simply the transpose of the transform and the transpose can be obtained by transposing the flow graph [15].

For our encoder, the direct fast 2D DCT developed by Feig [16] and the scaled 1D DCT developed by Arai et al. [17] were considered as candidates. The Feig 2D DCT is the most efficient algorithm proposed in the literature in terms of number of operations. It is a true 2D 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 1D DCT proposed in the literature is by Arai et al. This method requires 13 multiplications and 29 additions. However, eight of the thirteen multiplications can be absorbed into the quantization stage and thus the 1D DCT can be computed with 5 multiplications and 29 additions. Therefore, the 2D DCT can be computed by applying this fast 1D 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 2D scaled DCT. Therefore, the computational complexity of the separable approach is higher than the direct 2D approach. However, in terms of implementation, the 1D DCT approach was better suited for MMX instructions than the direct 2D approach due to the computational flow. The separable 2D 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.

7. EXPERIMENT

We compared the speed of our optimized implementation of an 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 than the TMN encoder. The distribution of the CPU load for our optimized H.263 encoder is shown in Figure 7. The "surprising" outcome is that with MMX, Q/DQ + scaling becomes the largest component. This is somewhat unexpected from other computing platforms.

Table 3: 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.

Encoder	Sequence	File Size	PSNR	Time 1	Time 2
TMN	Miss America	11.3 KB	37.1 dB	46.6 s	14.2 s
Encoder	Carphone	33.8 KB	33.2 dB	42.3 s	13.8 s
Optimized	Miss America	17.3 KB	37.1 dB	4.3 s	1.5 s
Encoder	Carphone	40.8 KB	33.1 dB	4.5 s	1.6 s

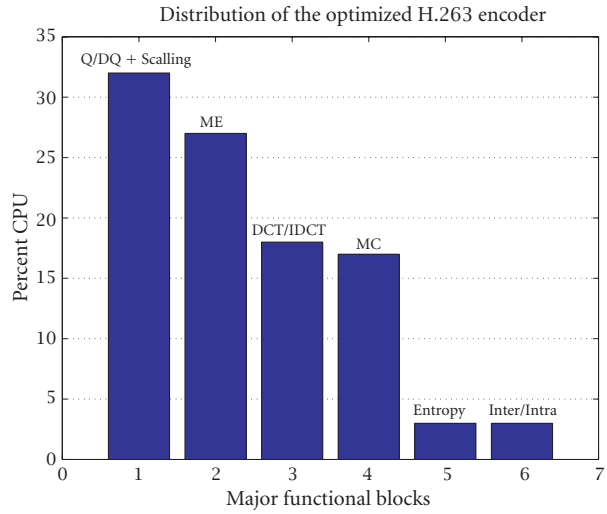

Figure 7: Distribution of CPU load for our optimized H.263 video encoder. The abbreviations ME, MC, Q/DQ + scaling, stands for motion estimation, motion compensation, and quantization/dequantization plus scaling for IDCT/DCT, respectively.

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 chose a fast motion estimation algorithm that traded compression efficiency with computation complexity. In conclusion, we note that for real-time applications, the compression efficiency must be balanced with the encoding speed for a given bandwidth to achieve maximum throughput (i.e., received frame rate).

8. CONCLUSION

In this paper, we considered the problem of software optimization of video codecs on the Pentium with MMX platform. We first reviewed the architecture of the Pentium processor by pointing out features that are relevant to software implementation. Then, we described the concept of SIMD

architecture and reviewed the MMX technology. Finally, we described an actual implementation of a fast H.263 video encoder utilizing the MMX technology. The "surprising" outcome is that with MMX, Q/DQ + scaling becomes the largest component, which is somewhat unexpected from other computing platforms. 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. This is different from the traditional approach of constructing a fast video encoder by selecting the fastest algorithm in terms of number of arithmetic operations for each component. The reason behind this is that each hardware platform has its own strengths and weaknesses, which is translated to the fact that certain operations can be done more efficiently. Thus, the fastest algorithm in term of number of operations might not result in the fastest implementation on a specific hardware platform. We compared the speedup with the H.263 standard TMN video encoder and found that the speedup is about tenfold at the expense of compression efficiency.

REFERENCES

[1] R. Lee, "Accelerating multimedia with enhanced microprocessors," *IEEE Micro*, vol. 15, no. 2, pp. 22–32, 1995.

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

[3] R. Coelho and M. Hawash, *DirectX, RDX, RSX, and MMX Techonolgy: A Jumpstart Guide to High Performance APIs*, Addison-Wesley, 1997.

[4] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1990.

[5] M. Lam, "Software pipelining: an effective technique for vliw machines," in *Proceedings of the SIGPLAN' 88 Conference on Programming Language Design and Implementation*, July 1988, pp. 318–328.

[6] V. Allan, R. Jones, R. Lee, and S. Allan, "Software pipelining," *ACM Computing Surveys*, vol. 27, no. 3, pp. 367–432, 1995.

[7] X. Lee and Y. Zhang, "A fast hierarchical motion-compensation scheme for video coding using block feature matching," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 6, no. 6, pp. 627–634, 1996.

[8] B. Liu and A. Zaccarin, "New fast algorithms for the estimation of block motion vectors," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 3, no. 2, pp. 148–157, 1993.

[9] J. Jain and A. Jain, "Displacement measurement and its application in interframe image coding," *IEEE Trans. on Communications*, vol. 29, pp. 1799–1808, 1981.

[10] R. Srinivasan and K. Rao, "Predictive coding based on efficient motion estimation," *IEEE Trans. on Communications*, vol. 33, pp. 888–895, 1985.

[11] R. Li, B. Zeng, and M. L. Liou, "A new three-step search algorithm for block motion estimation," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 4, pp. 438–442, August 1994.

[12] L. Liu and E. Feig, "A block-based gradient descent search algorithm for block motion estimation in video coding," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 6, pp. 419–422, August 1996.

[13] J. Chalidabhongse and C. Kuo, "Fast motion vector estimation using multiresolution-spatio-temporal correlations," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 7, no. 3, pp. 477–488, 1997.

[14] K. Rao and R. Yip, *Discrete Cosine Transform: Algorithms, Advantages, and Applications*, Academic Press, California, 1990.

[15] W. Pennebaker and J. Mitchell, *JPEG, Still Image Data Compression Standard*, Van Nostrand Reinhold, New York, 1994.

[16] E. Feig and E. Linzer, "Discrete cosine transform algorithms for image data compression," in *Proceedings Electronic Imaging'90 East, Boston*, November 1990, pp. 84–87.

[17] Y. Arai, T. Agui, and M. Nakajima, "A fast dct-sq scheme for images," *Trans. IEICE*, vol. 71, pp. 1095–1097, November 1988.

Pohsiang Hsu received the B.S. degrees in electrical engineering and mathematics, M.S. and Ph.D. degrees in electrical engineering all from the University of Maryland at College Park in 1993, 1995, 1999, respectively. He was a research assistant in the DSP lab at Maryland from 1995 to 1997. Then, He joined a start up company called Odyssey Technologies from 1997 to 1999. Currently, He is with Microsoft Corporation. His research interests include video compression and image processing.

K. J. Ray Liu received the B.S. degree from the National Taiwan University, and the Ph.D. degree from UCLA, both in electrical engineering. He is a Professor of Electrical and Computer Engineering Department and Institute for Systems Research of University of Maryland, College Park. He was a Visiting Professor at Stanford University, CA; IT University, Copenhagen, Denmark; and Hong Kong Polytechnic University. His research interests span broad aspects of signal processing algorithms and architectures; image/video compression, coding, and processing; wireless communications; security; and medical and biomedical technology. He has published over 200 papers, of which over 60 are in refereed journals. Dr. Liu received numerous awards including the 1994 National Science Foundation Young Investigator, the IEEE Signal Processing Society's 1993 Senior Award (Best Paper Award), IEEE Vehicular Technology Conference Best Paper Award, Amsterdam, 1999, and the George Corcoran Award in 1994 for outstanding contributions to electrical engineering education and the Outstanding Systems Engineering Faculty Award in 1996 in recognition of outstanding contributions in interdisciplinary research, both from the University of Maryland. Dr. Liu is Editor-in-Chief of EURASIP Journal on Applied Signal Processing, and has been an Associate Editor of IEEE Transactions on Signal Processing, a Guest Editor of special issues on Multimedia Signal Processing of Proceedings of the IEEE, a Guest Editor of special issue on Signal Processing for Wireless Communications of IEEE Journal of Selected Areas in Communications, a Guest Editor of special issue on Multimedia Communications over Networks of IEEE Signal Processing Magazine, a Guest Editor of special issue on Multimedia over IP of IEEE Trans. on Multimedia, and an editor of Journal of VLSI Signal Processing Systems. He currently serves as the Chair of Multimedia Signal Processing Technical Committee of IEEE Signal Processing Society and the series editor of Marcel Dekker series on signal processing and communications.