ESTIMATING MULTIMEDIA INSTRUCTION PERFORMANCE BASED ON WORKLOAD CHARACTERIZATION AND MEASUREMENT

By

ADIL ADI GHEEWALA

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2002

Dedicated to my parents, Adi and Jeroo, and my brother Cyrus

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to the chairman of my supervisory committee, Dr. Jih-Kwon Peir, for his constant encouragement, support, invaluable advice and guidance during this research. I would like to express my deep gratitude to Dr. Jonathan C.L. Liu for his inspiration and support. I wish to thank Dr. Michael P. Frank for willingly agreeing to serve on my committee and for the guidance he gave me during my study in this department. I also wish to thank Dr. Yen-Kuang Chen of Intel Labs. for his experienced advice towards my thesis.

I would like to recognize Dr. Manuel E. Bermudez for sharing his knowledge, his continual encouragement and guidance during my study at the University.

I would also like to thank Ju Wang and Debasis Syam for their involvement and help in my research.

In addition I would like to give a special thanks to my parents and brother for their endless support and confidence in me, and the goals I could achieve. Without their guidance, I would have never made it this far.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

ESTIMATING MULTIMEDIA INSTRUCTION PERFORMANCE BASED ON WORKLOAD CHARACTERIZATION AND MEASUREMENT

By

Adil Adi Gheewala

December 2002

Chair: Jih-Kwon Peir
Major Department: Computer and Information Science and Engineering

Single instruction multiple data (SIMD) techniques can be used to improve the performance of media applications like video, audio and graphics. MMX technology extends the Intel architecture to provide the benefits of SIMD for media applications.

We describe a method to estimate performance improvement of a new set of media instructions on emerging applications based on workload characterization and measurement for a future system. Application programs are characterized into sequential segments and vectorizable segments. The vectorizable portion of the code can be converted to its equivalent vectorized segment by using the new SIMD instructions. Additional data movement instructions are required to fully utilize the capabilities of these new SIMD instructions. Benchmarking and measurement techniques on existing systems are used to estimate the execution time of each segment. The speedup and the data movement instructions needed for the efficient use of the new media instructions can be estimated based on these measurement results. Using the architectural details of the

systems we can estimate the speedup of the new instructions. Processor architects and designers can use this information to evaluate different design tradeoffs for the new set of media instructions for next generation systems.

CHAPTER 1
INTRODUCTION

The popularity and importance of digital signal and multimedia processing are on the rise in many microprocessor applications. The volume of data processed by computers today is increasing exponentially, placing incredible demands on the microprocessor. New communications, games and entertainment applications are requiring increasing levels of performance [1]. Virtually, all commercial microprocessors, from ARM to Pentium, have some type of media-oriented enhancements. MMX technology includes new data types and 57 new instructions to accelerate calculations common in audio, 2D and 3D graphics, video, speech synthesis and recognition, and data communications algorithms [1]. These instructions exploit the data parallelism at sub-word granularity that is often available in digital signal processing of multimedia applications.

1.1 Factors Affecting Performance Improvement

The performance improvement due to the use of media-enhanced instructions, such as the MMX technology, is based on three main factors: the percentage of data parallelism in applications that can be exploited by the MMX instructions; the level of parallelism each MMX instruction can exploit, which is determined mainly by the data granularity and the operations that are parallelised; and the relationship between the data structures for the memory used and the MMX registers in order to utilize the MMX instruction capabilities. The third factor is due to the data movement between the MMX

registers and the memory. These data movements are required for efficiently using the MMX instructions and can reduce the overall performance improvement.

1.2 New Media Instructions

One essential issue in defining the media-extension instructions is their ability to exploit parallelism in emerging multimedia applications to achieve certain performance targets. Performance evaluation of a new set of media instructions on applications is critical to access architectural tradeoffs for the new media instructions. The traditional cycle accurate simulation is a time-consuming process that requires detailed processor models to handle both regular and new single instruction multiple data (SIMD) media instructions. In addition, proper methods are needed to generate executable binary codes for the new media-extension instructions to drive the simulator.

We describe a method of estimating performance improvement of new media instructions based on workload characterization and measurements. The proposed method allows processor architects and media experts to quickly discover the speedup of some emerging applications for a future system with a few additional media instructions to the existing instruction set architecture. The uniqueness of this approach is that we only need the real existing hardware, no cycle-accurate simulator.

The basic approach involves several steps. First, a multimedia application program is selected and we identify the sections, “hot spots,” that take a sizeable amount of the total execution time. We then develop an equivalent code for the entire application with new SIMD media instructions that we would like to incorporate in the future system. Second, the application is characterized into three execution segments, the sequential code, the data manipulation instructions required for the correct use of the new media instructions, and the segment of the code that can be replaced (or vectorized) by the new

SIMD instructions. We measure the execution time of each segment on an existing system, and assume that the fraction of time for these segments will not change significantly on the future system. The execution time of the vectorizable segment can be extrapolated according to the architectural speedup of the new SIMD instructions. We can estimate the speedup of the vectorizable segment by taking the weighted average of speedups for each new SIMD instruction. The speedup of each new media instruction can be estimated by comparing the cycle count of the new SIMD instruction on the future system with the cycle count of its equivalent C-code on the existing system. Finally, the total execution time of the application with the additional SIMD instructions can be calculated by summing the execution time of the three segments.

The proposed method is experimented on an Intel Pentium III system that has the MMX technology. We estimated the execution time of an inverse discrete cosine transformation (IDCT) [2-5] program with "new" media instructions. For this case study we assumed pmaddwd, paddd, psubd and psrad to be the new MMX instructions that we want in the future instruction set architecture. The estimated measurements are within 5% of the actual measured execution time.

1.3 Organization Of The Thesis

We have organized our work by first providing, in chapter 2, a brief review of the basic principles of the system and application used. This chapter explains Pentium III pipeline design, how instruction can be paired in the 2 pipes, cycle time and latency of instructions. We also explain the features of the MMX technology and guidelines to vectorize an application. We then explain the application, IDCT [2,3], used to present our methodology.

In chapter 3, we discuss the methodology used to characterize and measure timings for different segments of the application, application program used, the compiler and subroutines used to measure the timing of each segment and the method of estimating of the speedup of the code that is vectorized. Additionally, we describe some limitations and assumptions of our methodology.

In chapter 4 we explain, with results of our measurements, a case study of an application program. We select IDCT as the media application since it is one of the most compute-intensive programs in JPEG, MPEG and many other real life media applications.

Finally, we provide our conclusion and observations in chapter 5.

CHAPTER 2
REVIEW OF ARCHITECTURE OF PENTIUM III AND APPLICATION

This chapter will give an overview of MMX technology followed by the Pentium III architecture and then the application we used to describe the methodology, inverse discrete cosine transformation (IDCT).

2.1 MMX Technology

In 1996, the Intel Corporation introduced MMX technology into Intel Pentium processors. MMX technology is an extension to the Intel architecture (IA) instruction set. The technology uses a single instruction multiple data (SIMD) technique to speed up multimedia software by processing data elements in parallel. The MMX instruction set adds 57 new instructions and a 64-bit quadword data type. There are eight 64-bit MMX technology registers, each of which can be directly addressed using the register names MM0 to MM7. MMX technology exploits the parallelism inherent in many multimedia algorithms. Many of these algorithms exhibit the property of repeated computation on a large data set.

Media applications have the following common characteristics [6]:

- Small, native data types (for example, 8-bit pixels)
- Regular memory access patterns
- Localized, recurring operations on the data
- Compute-intensive

MMX technology defines new register formats for data representation. The key feature of multimedia applications is that the typical data size of operands is small. Most

of the data operands' sizes are either a byte or a word (16 bits). Also, multimedia processing typically involves performing the same computation on a large number of adjacent data elements. These two properties lend themselves to the use of SIMD computation.

MMX technology features include the following [6]:

- New data types built by packing independent small data elements together into one register.
- An enhanced instruction set that operates on all independent data elements in a register, using a parallel SIMD fashion.
- New 64-bit MMX registers that are mapped on the IA floating-point registers.
- Full IA compatibility.

2.1.1 New Data Types, 64-Bit MMX Registers And Backward Compatibility

New data types. MMX technology introduces four new data types: three packed data types and a new 64-bit entity. Each element within the packed data types is an independent fixed-point integer [6].

The four data types are defined below in Table 2-1 and Figure 2-1.

Table 2-1. Data Types In MMX

Data Type	Description
Packed byte	8 bytes packed into 64 bits
Packed word	4 words packed into 64 bits
Packed doubleword	2 doublewords packed into 64 bits
Packed quadword	64 bits

64-bit MMX registers. MMX technology provides eight new 64-bit general-purpose registers that are mapped on the floating-point registers. Each can be directly addressed within the assembly by designating the register names MM0-MM7 in MMX instructions [6].

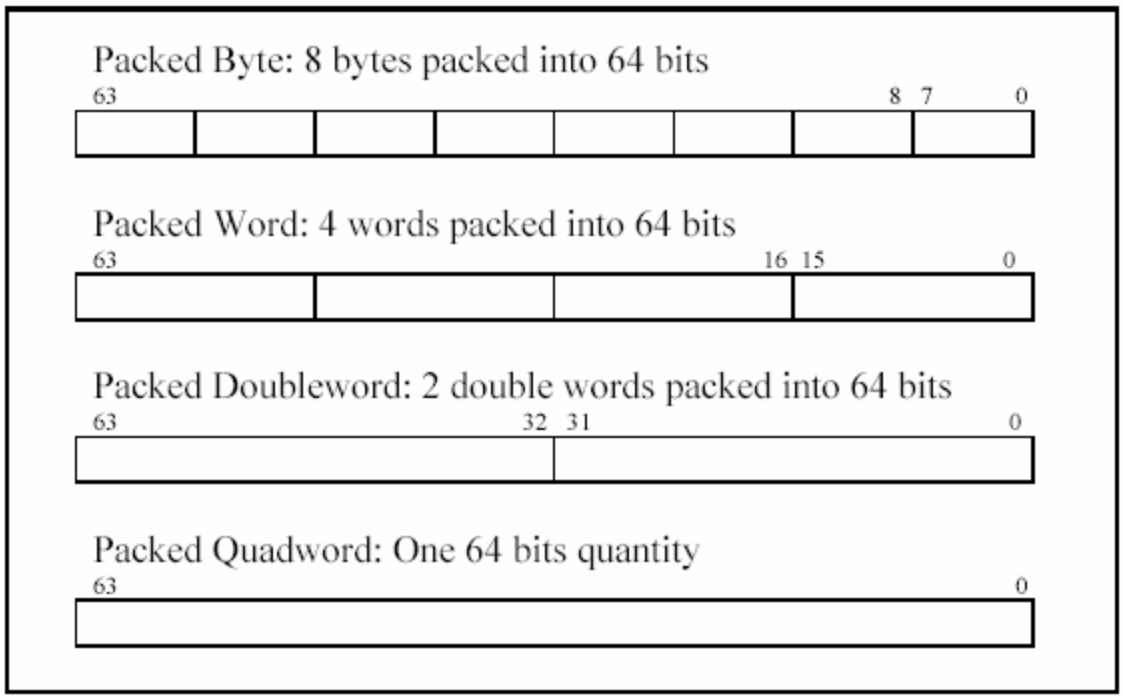

Figure 2-1. Data Types In MMX.

Backward compatibility. One of the important requirements for MMX technology was to enable use of MMX instructions in applications without requiring any changes in the IA system software. MMX technology, while delivering performance boost to media applications, is fully compatible with the existing application and operating system base [6].

2.1.2 Enhanced Instruction Set

MMX technology defines a rich set of instructions that perform parallel operations on multiple data elements packed into 64 bits (8x8-bit, 4x16-bit, or 2x32-bit fixed-point integer data elements) [6]. Overall, 57 new MMX instructions were added to the Intel Architecture instruction set. Selected MMX instructions can operate on signed or unsigned data using saturation arithmetic.

Since MMX instructions can operate on multiple operands in parallel, the fundamental principle of MMX technology optimization is to vectorize the operation, Figure 2-2.

Following are the points to remember for MMX technology optimization [3]:

- Arrange multiple operands to be execute in parallel.
- Use the smallest possible data type to enable more parallelism with the use of a longer vector.
- Avoid the use of conditionals.

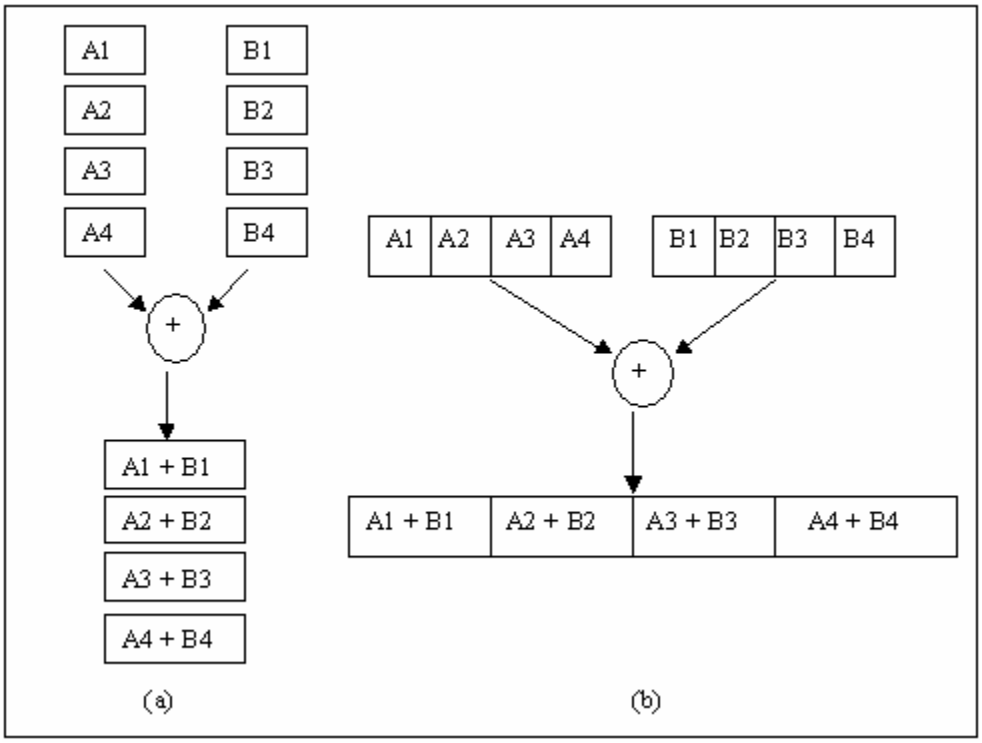

Figure 2-2. Scalar Vs. SIMD add Operation Example (a) Conventional scalar operation, to add two vectors we have to add each pair of components sequentially. (b) Using SIMD instructions we can add the two vectors using one instruction.

Some arithmetic instructions are shown in Table 2-2 and data movement instructions in Table 2-3. See Appendix A for examples of these instructions.

Table 2-2. Some Arithmetic MMX Instructions

Opcode	Options	Cycle Count	Description
PADD [B/W/D] PSUB [B/W/D]	Wrap-around and saturate	1	Packed eight bytes (b), four 16-bit words (w), or two 32-bit doublewords (d) are added or subtracted in parallel.
PMADDWD	Word to doubleword conversion	Latency:3 Throughput: 1	Packed four signed 16-bit words are multiplied and adjacent pairs of 32 results are added together, in parallel. Result is a doubleword.

If an instruction supports multiple data types, byte (b), word (w), doubleword (d), or quadword (q), then they are listed in brackets. Source [6,7]

Table 2-3. Some Data Movement MMX Instructions

Opcode	Options	Cycle Count	Description
PUNPCKL[BW/WD/DQ] PUNPCKH[BW/WD/DQ]		1	Packed eight bytes (b), four 16-bit words (w), or two 32-bit doublewords (d) are merged with interleaving
MOV[D/Q]		1 (if data in cache)	Moves 32 or 64 bits to and from memory to MMX registers, or between MMX registers. 32-bits can be moved between MMX and integer registers.

If an instruction supports multiple data types, byte (b), word (w), doubleword (d), or quadword (q), then they are listed in brackets. Source [6,7]

2.2 Pentium III Architecture

We briefly discuss here the Pentium III super scalar architecture, cycle count and throughput of some instructions and rules for pairing these instructions as we use this information in chapter 4 to calculate the cycles an MMX code can take. Using the calculated cycle count of the MMX code we can measure the speedup of the vectorizable C-code by comparing their cycle counts.

In 1999, the Intel Corporation introduced a new generation of the IA-32 microprocessors called the Pentium III processor. This processor introduced 70 new

instructions, which include MMX technology enhancements, SIMD floating-point instructions, and cacheability instructions [3].

The Pentium III processors are aggressive micro architectural implementations of the 32-bit Intel architecture (IA). They are designed with a dynamic execution architecture that provides the following features [8]:

- Out-of-order speculative execution to expose parallelism.
- Super scalar issue to exploit parallelism.
- Hardware register renaming to avoid register name space limitations.
- Pipelined execution to enable high clock speeds.
- Branch prediction to avoid pipeline delays.

Pipeline. The Pentium III processors' pipelines contain the following three parts [8], as shown in Figure 2-3:

- The in-order issue front end.
- The out-of-order core.
- The in-order retirement unit.

The Pentium III processor has two pipelines for executing instructions, called the U-pipe and the V-pipe. When certain conditions are met, it is possible to execute two instructions simultaneously, one in the U-pipe and the other in the V-pipe. It is therefore advantageous to know how and when instructions can be paired. Some instructions like MOV, LEA, ADD, SUB, etc are pairable in either pipe. Instructions like SAR, SHL, SAL are pairable in the U-pipe only [9].

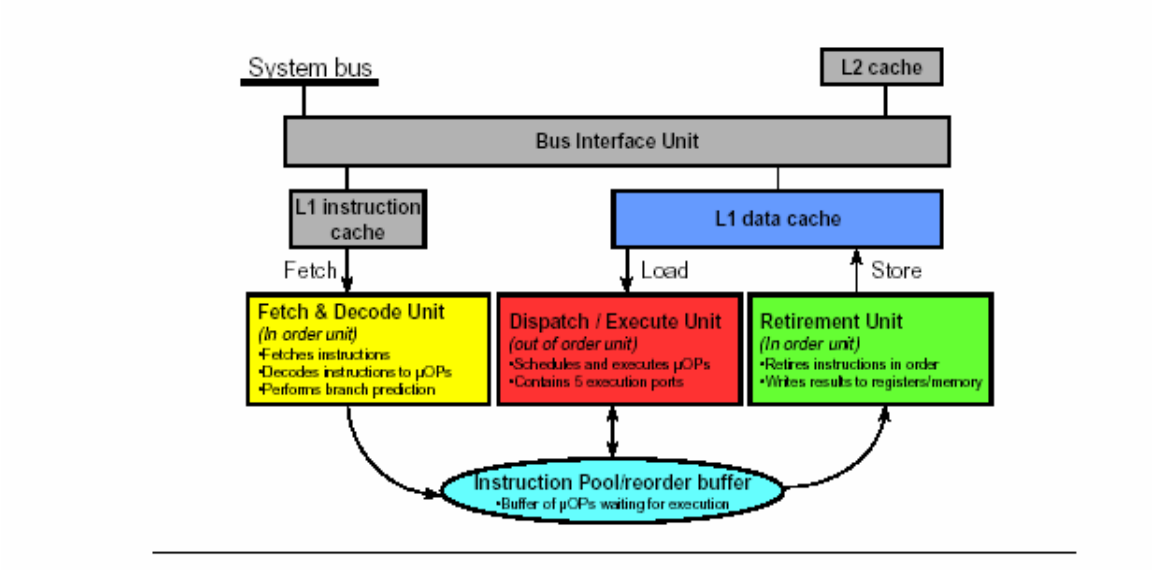

Figure 2-3. PIII Architecture [8]

Two consecutive instructions will pair when the following conditions are met [9]:

- The first instruction is pairable in the U-pipe and the second instruction is pairable in the V-pipe.
- The second instruction does not read or write a register, which the first instruction writes to.

The following are special pairing rules for MMX instructions [9]:

- MMX shift, pack or unpack instructions can execute in either pipe but cannot pair with other MMX shift, pack or unpack instructions.
- MMX multiply instructions can execute in either pipe but cannot pair with other MMX multiply instructions.
- MMX instructions that access memory or integer registers must execute in the U pipeline.

The clock cycles and pairability of instructions that we will need can be summarized in Table 2-4.

Table 2-4. Cycle Cycles And Pairability Of Instructions

Instruction	Operands	Latency	Pairability
MOV	r/m, r/m/i		pairable in either pipe
MOV	m , accum		pairable in either pipe
LEA	r, m		pairable in either pipe
ADD SUB AND OR XOR	r, r/i		pairable in either pipe
ADD SUB AND OR XOR	r , m		pairable in either pipe
ADD SUB AND OR XOR	m , r/i		pairable in either pipe
SHR SHL SAR SAL	r , i		pairable in U-pipe
IMUL	r, r	4	not Pairable

*r = register, m = memory, i = immediate. Delay = the delay the instruction generates in a dependency chain. Source [2,3] For a complete list of pairing rules see reference [8,9]

A list of MMX instruction timings is not needed because they all take one clock cycle, except the MMX multiply instructions which take 3. MMX multiply instructions can be overlapped and pipelined to yield a throughput of one multiplication per clock cycle.

The Table 2-5 shows the delay and throughput of some MMX instructions on Pentium III

Table 2-5. Delay And Throughput Of Some MMX Instructions

Instruction	Operands	Delay	Throughput
PMUL PMADD	r64,r64	3	1/1
PMUL PMADD	r64,m64	3	1/1
PADD PSUB PCMP	r64,r64		1/1
PADD PSUB PCMP	r64,m64		1/1
MOVD MOVQ	r,r		2/1
MOVD MOVQ	r64,m32/64		1/1
MOVD MOVQ	m32/64,r64		1/1
PSRA PSRL PSLL	r64,r64/i		1/1
PACK PUNPCK	r64,r64		1/1
PACK PUNPCK	r64,m64		1/1
PAND PANDN POR PXOR	r64,r64		2/1

*r = register, m = memory, i = immediate Source [3,9]

2.3 IDCT

8x8 DCTs and 8x8 IDCTs are mathematical formulae extensively used in image, video compression/decompression, DVD encoding/decoding and many other signal processing applications. The optimization for DCT/IDCT has also been implemented for MMX technology.

An MPEG 2 decoder structure [4] and a JPEG block diagram [5] are show in Figure 2-4 and Figure 2-5, to highlight the importance of IDCT in signal processing.

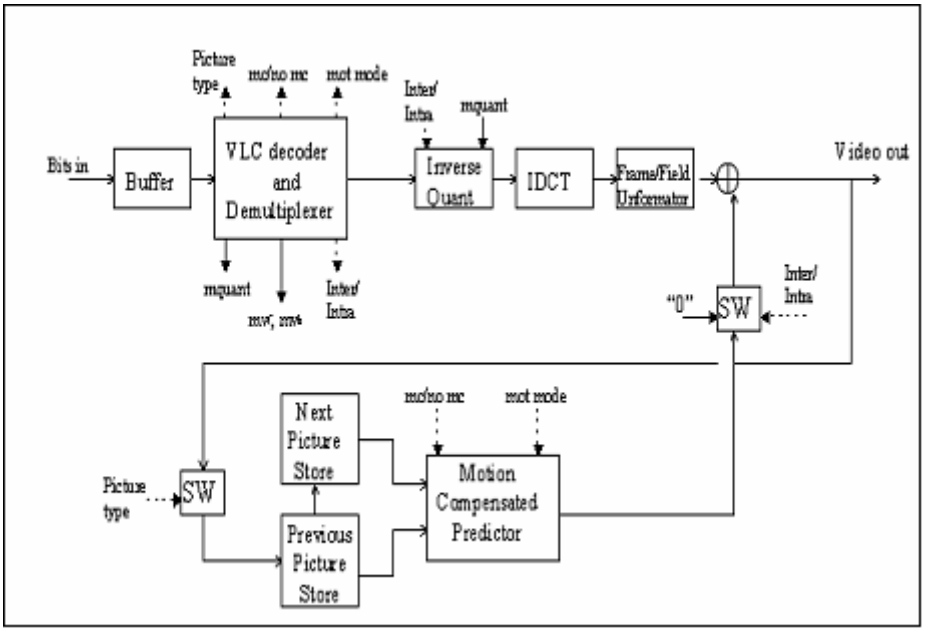

Figure 2-4. MPEG2 Decoder Structure [4]

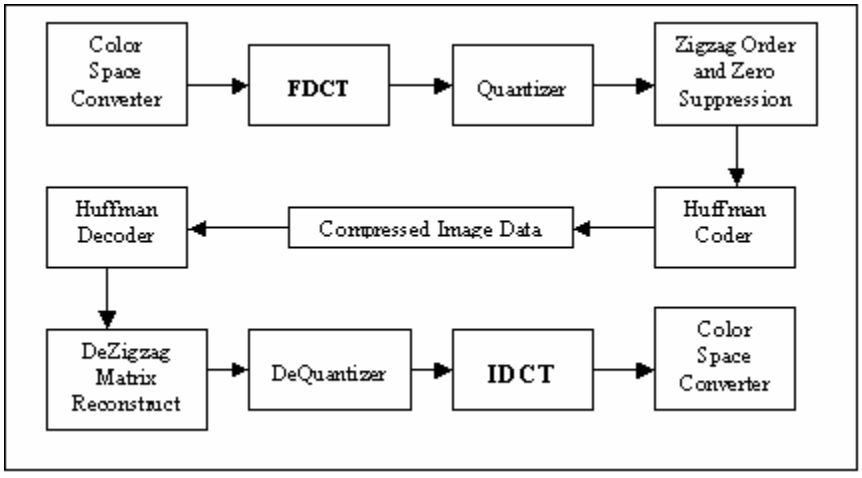

Figure 2-5. JPEG Block Diagram [5]

The discrete-cosine transformation matrix is factored out in many fast DCT implementations into butterfly and shuffle matrices, which can be computed with fast integer addition. For example, Chen's algorithm is one of the most popular algorithms of this kind. [3].

The general formula for 2D IDCT is [10]

$$f_{xy} = \frac{2}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} C_u C_v F_{uv} \cos\left(\frac{(2x+1)u\pi}{2N}\right).\cos\left(\frac{(2y+1)v\pi}{2N}\right).$$

Replacing N with 8 since we are computing 8x8 2D IDCT we get

$$f(x,y) = \frac{1}{4}\sum_{x=0}^{7}\sum_{y=0}^{7} C_u C_v F(u,v) \cos\left(\frac{(2x+1)u\pi}{16}\right).\cos\left(\frac{(2y+1)v\pi}{16}\right).$$

x,y = spatial coordinates in the pixel domain (0,1,2,…7)

u,v = coordinates in the transform domain (0,1,2,…7)

$$C_u = \frac{1}{\sqrt{2}} \; for \; \mathrm{u} = 0, \text{otherwise } 1$$

$$C_v = \frac{1}{\sqrt{2}} \; for \; \mathrm{v} = 0, \text{otherwise } 1$$

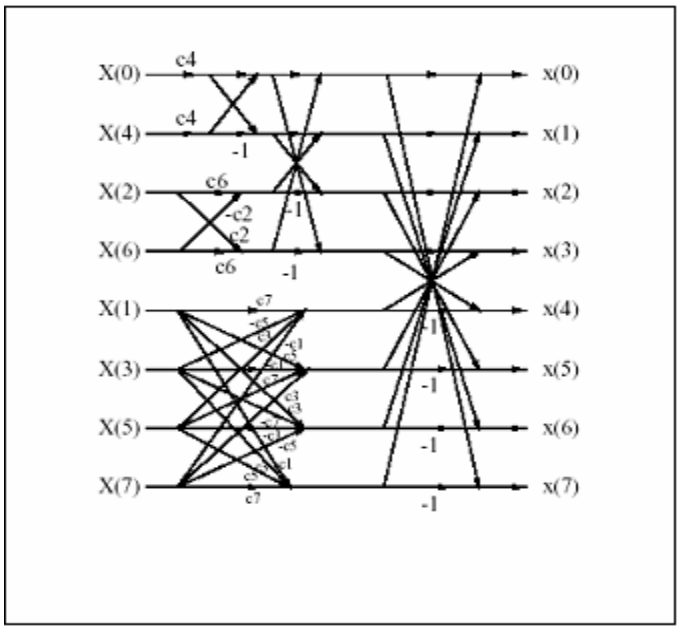

Figure 2-6. Even/Odd Decomposition Algorithm For IDCT [2]

The 2-D 8x8 IDCT can be implemented with the row-column decomposition method of 1-D 8-point IDCT. The 8-point IDCT is carried out using the even-odd

decomposition algorithm. The resulting algorithm can be represented by the Butterfly chart for IDCT Algorithm [2] as shown in Figure 2-6. This algorithm is compute-intensive on small size data. The repetition of operations like multiplication and addition make it a good choice for the use of SIMD instructions.

CHAPTER 3
METHODOLOGY

A program can be broken up into sequential segments and vectorizable segments. The execution time of each segment is measured on an existing system, assuming that the new set of media instructions do not exist. We use the Intel MMX technology as the basis of our experiment. To simplify our discussion we assume that these new instructions do not themselves include any data move instructions. We refer to the arithmetic, logical, comparison and shift instructions as computational instructions, while the move, pack and unpack as the data move instructions.

3.1 Using SIMD Instructions

The vectorizable segments can be converted to the new set of media instructions, and the data might need to be shuffled around to facilitate the use of these instructions. This will require data move instructions, which position the input data for the correct operation of the new set of media instructions. They are also required to move data to and from memory and MMX registers. These data move instructions are the programmer's responsibility and can be characterized and measured for performance evaluation. SIMD instructions operate on media registers whose number is limited, i.e. 8, each with a limited width, 64-bits. This limitation reduces the efficiency of the moving data between memory and media registers. This limitation is further exacerbated by the semantic gap between the normal layout of data in memory and the way the data needs to be arranged in the media registers.

3.1.1 Data Movement Instructions In Pure C Code

When we program in a high-level language like C, we do not explicitly load the input data into internal registers before use. High-level languages free the programmer from taking care of the data movement instructions needed to carry out an operation. The programmer simply writes a program that operates on data stored in data structures supported by the language.

The compiler generates the data movement instructions, required for the correct operation of the instruction, when it produces the assembly level code. This is in contrast to programming using MMX instructions. Writing a code in MMX is equivalent to programming in assembly level, giving us access to load internal registers as we see fit. The assignments made by the programmer may hence not be optimal.

We assume that the data move instructions generated by the compiler are part of the computational instruction in C, where as the data movement instructions in MMX are characterized as the separately.

Figure 3-1 gives an example of a C code and its gcc compiled version. The table and row array elements need to be loaded into internal registers before performing the multiplications and then the partial results are added. The result generated is then stored back into the variable's address.

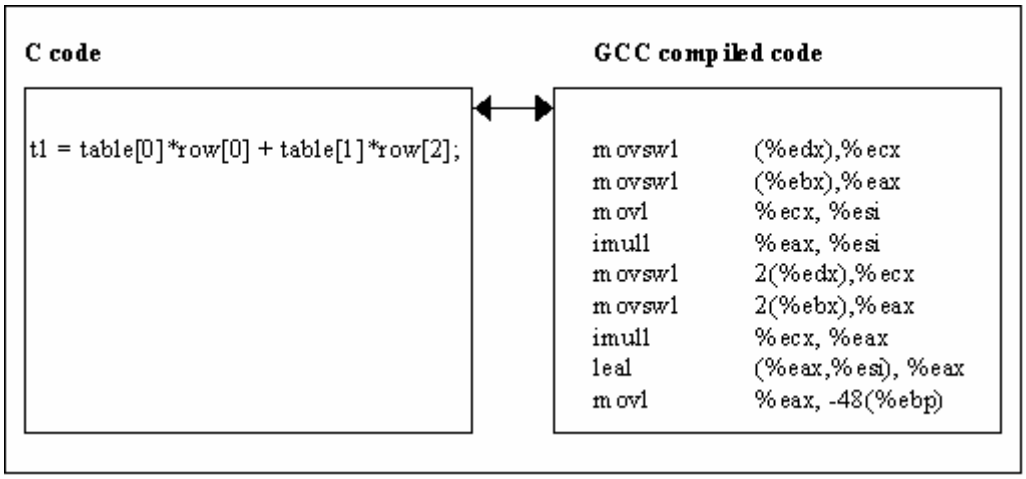

Figure 3-1. Relationship Of The High-Level And Its Corresponding Assembly Level Code

3.1.2 Memory Hierarchy Changes

There is an important assumption about the memory hierarchy performance that we would like to point out. The L1 cache is limited and cache misses can be very expensive. We assume that all the data needed by our application is cached.

The memory hierarchy in the future system may improve due to increase in cache size or any other memory improvement techniques and hence the time taken by the data movement instructions, which we measure on our current system, might reduce in the future system. We assume that the fraction of time the data moves take will not significantly change in the future system. This is also true for the sequential segment.

This reduction may be compensated by the increase in application program size. Also an increase in data size may further reduce the additional performance gain due to improvement in memory hierarchy.

3.1.3 Matrix-Vector Multiplication

The above concept is explained by Figure 3-2, which illustrates a simple 4x4 matrix multiplied with a 4x1 vector using MMX instructions. We assume that 16-bit

input data is used. The Packed-Multiply-Add (pmaddwd) performs four 16-bit multiplications and two 16-bit additions to produce two 32-bit partial results. In conjunction with another pmaddwd and a Packed-Add (paddd), two elements in the resulting vector can be obtained as shown in part (a) of the figure. The other 2 elements of the resulting vector can be similarly obtained by using 2 pmaddwd and 1 paddd.

Each row of the matrix needs to be split in 2. The first 2 halves of 2 rows of the matrix need to be grouped in a 64-bit media register. The similar process is done on the lower halves of the 2 rows. The first two elements of the vector are duplicated and so are the lower 2 in two separate media registers. This is done to take advantage of the pmaddwd and paddd instructions. This peculiar data arrangement makes MMX programming difficult.

In addition, data arrangement incurs performance reduction. Not only are data move instructions required to set up the source for pmaddwd and paddd but also required to place the results back into memory. The time taken by the data moves needs to be considered in estimating the speedup for adding the new SIMD media instructions.

The speedup of an application can be estimated by the following modified Amdahl's law

$$Speedup = \frac{1}{(1-f) + (f/n) + O}$$

where f is the fraction of the code that is vectorizable, n is the ideal speedup of f and O is the fraction of time for the data movement instructions required for using the media instructions.

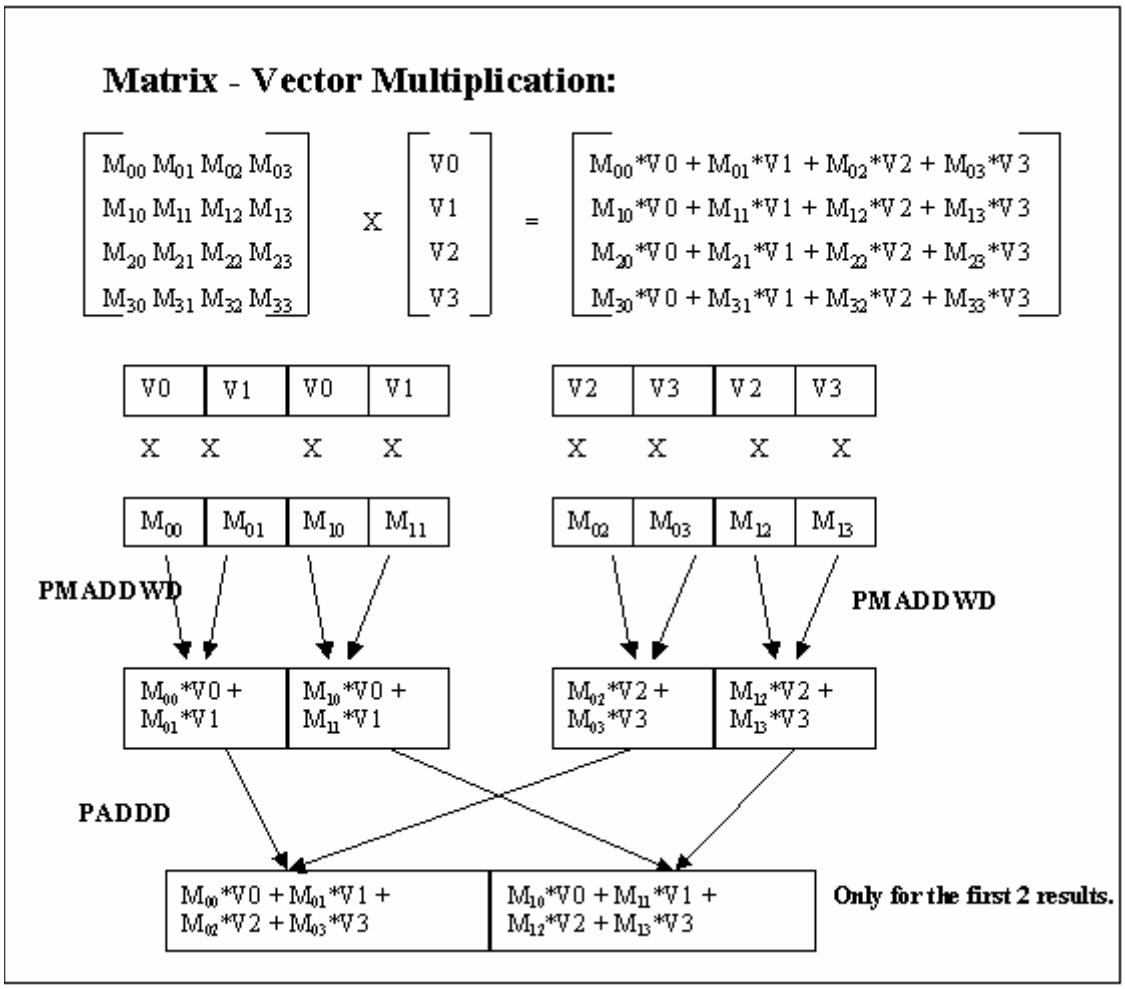

Figure 3-2. Using The Current MMX Instructions

In contrast, a more natural data arrangement can be accomplished as shown in Figure 3-3. The entire source vector and each row of the matrix are moved into separate MMX registers for pmaddwd to operate. To allow this type of arrangement a new paddd must be invented that adds the upper and lower 32-bits of each of the source registers. However depending on the subsequent computations, further data rearrangement may still be needed.

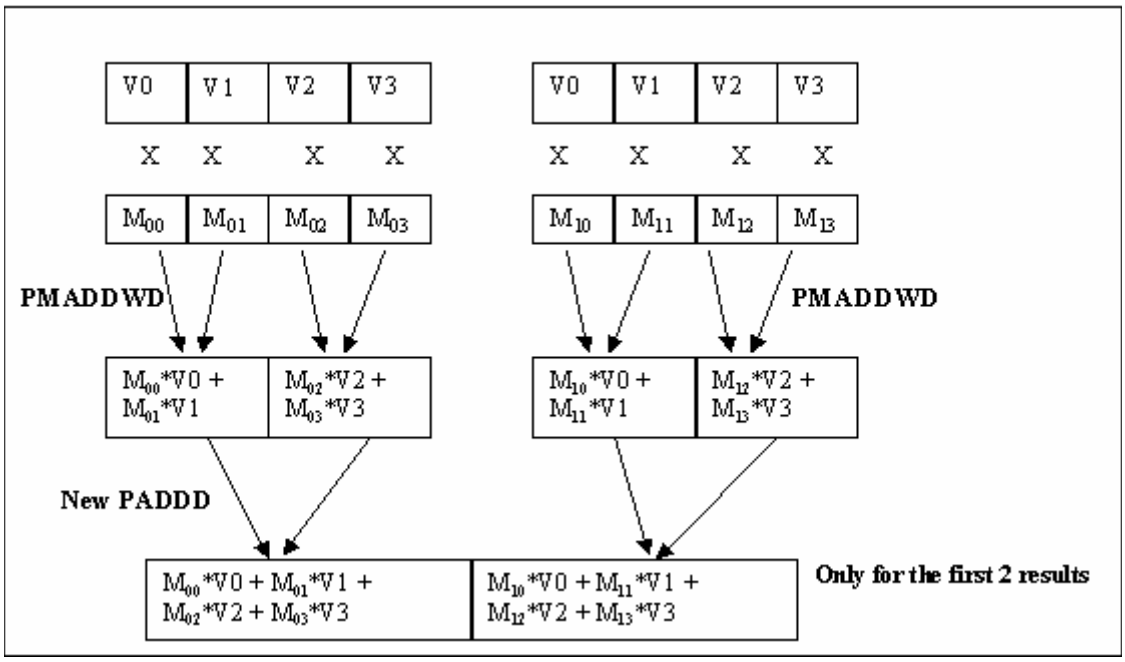

Figure 3-3. A More Natural Way To Do The Multiplication

3.2 Methodology In Detail

In the first step, we develop the application program (assuming written in C, also referred to as the C-code), its equivalent vectorized code (referred to as the MMX-code), and its equivalent MMX code (referred to as the pseudo MMX-code). The MMX-code includes SIMD computational instructions along with the necessary data movement instructions. The pseudo MMX-code includes SIMD instructions for data movement along with the equivalent plain C instructions for computational instructions. We refer to this step as a vectorization step because of its similarity with developing vector code for vector machines.

There are two important considerations in this step. First, the MMX-code should be as optimized as possible to exploit the SIMD capability. Second, the MMX-code and Pseudo-MMX code should be able to run on the current system with correct outputs.

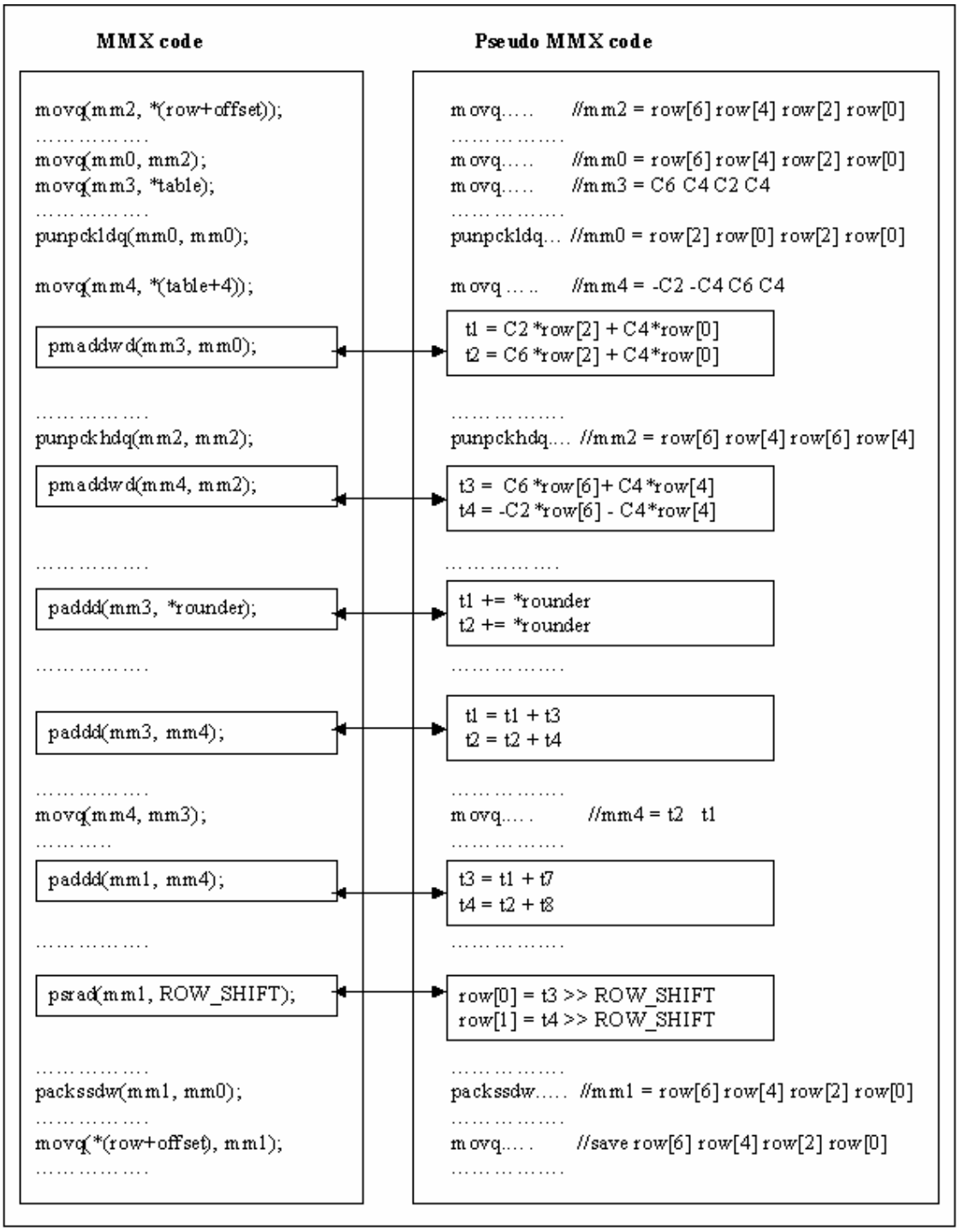

Figure 3-4. Portion Of IDCT Code And Its Pseudo Vectorized Equivalent

Based on the C-code, MMX-code and Pseudo MMX-code we can estimate the time associated with the data moves. We can also estimate the total execution time with the new computational SIMD instructions. In the Pseudo MMX-code the equivalent C-code

must be perfectly mapped to the corresponding MMX-code. Since a true new MMX instruction is not available on the existing processor, the Pseudo MMX-code can be used for timing measurement. Figure 3-4 shows a portion of the MMX-code from idct_row that has been vectorized and its equivalent Pseudo MMX-code. The equivalent C-code is used as the basis for calculating the time taken by the data moves. Since the equivalent C-code is directly translated from the MMX-code, the C-code may not be optimal.

C-code. We first run the equivalent C-code on a host system that has MMX technology, measuring several time components including the total execution time, time required for the sequential code and the time required for the vectorizable code.

MMX data movement. Next, we need to estimate the time of the MMX data movement instructions needed for the new computational instructions. The execution times of the Pseudo MMX-code that consists of all the data moves and the equivalent C-code are measured on the host system. The difference of the execution times of the equivalent C-code and the Pseudo MMX-code can provide the estimated time for the data moves. To make sure that all the data movement instructions are actually executed we use the lowest level of compiler optimization and verify the execution of each instruction by using the gprof [11] tool.

Vectorizable C-code. Similarly, the computational instructions in the MMX-code can be removed without replacing them by the equivalent C-code. This Crippled-code will not generate correct results with all the removed computational instructions. The time measurement is performed on this Cripple-code using gprof to verify that all the remaining instructions are executed. The measured execution time represents the sequential code plus the time for the data movement associated with the MMX-code.

Therefore, the difference of the execution times of the Crippled-code and the Pseudo MMX-code can provide the estimated execution time for the vectorizable portion of the C-code. This portion of the total execution time is the target for improvement with the new SIMD computational instructions.

Estimating the architectural speedup for the SIMD instructions. We need to estimate the speedup of the vectorizable segment due to the use of the new SIMD instructions. We have measured the time the vectorizable segment takes on the host system. If the speedup of the new media instructions is estimated, its execution time can be easily calculated.

The relationship between the MMX code and the C-code is shown in Figure 3-4. Each instruction in the C code can be mapped to the assembly level instructions as shown in Figure 3-5 and 3-6. The complete row method can bee seen in the Appendix B. Instructions, in the C-code, that get converted to a new SIMD instruction, are grouped together. Using the system's architectural information and the assembly level code, the number of cycles an instruction takes can be calculated, as shown in Figure 3-5 and 3-6. Pairing rules and system architecture explained in chapter 2 are used to calculate these cycle counts. These results are within 10% accuracy when verifies by using the read time stamp counter (RDTSC) [12] tool explained in later sections. Dividing the cycles so calculated for each instruction by the cycle time of the new SIMD instruction, provided by the architect, gives us the speedup for that new media instruction. Speedup of each new SIMD instruction can be calculated by using this information. A weighted speedup can be calculated by summing the products of the instruction count and its individual speedup and diving the sum by the total number of instructions.

By dividing the execution time of the vectorizable code by the weighted speedup we can estimate the time for the computational instructions.

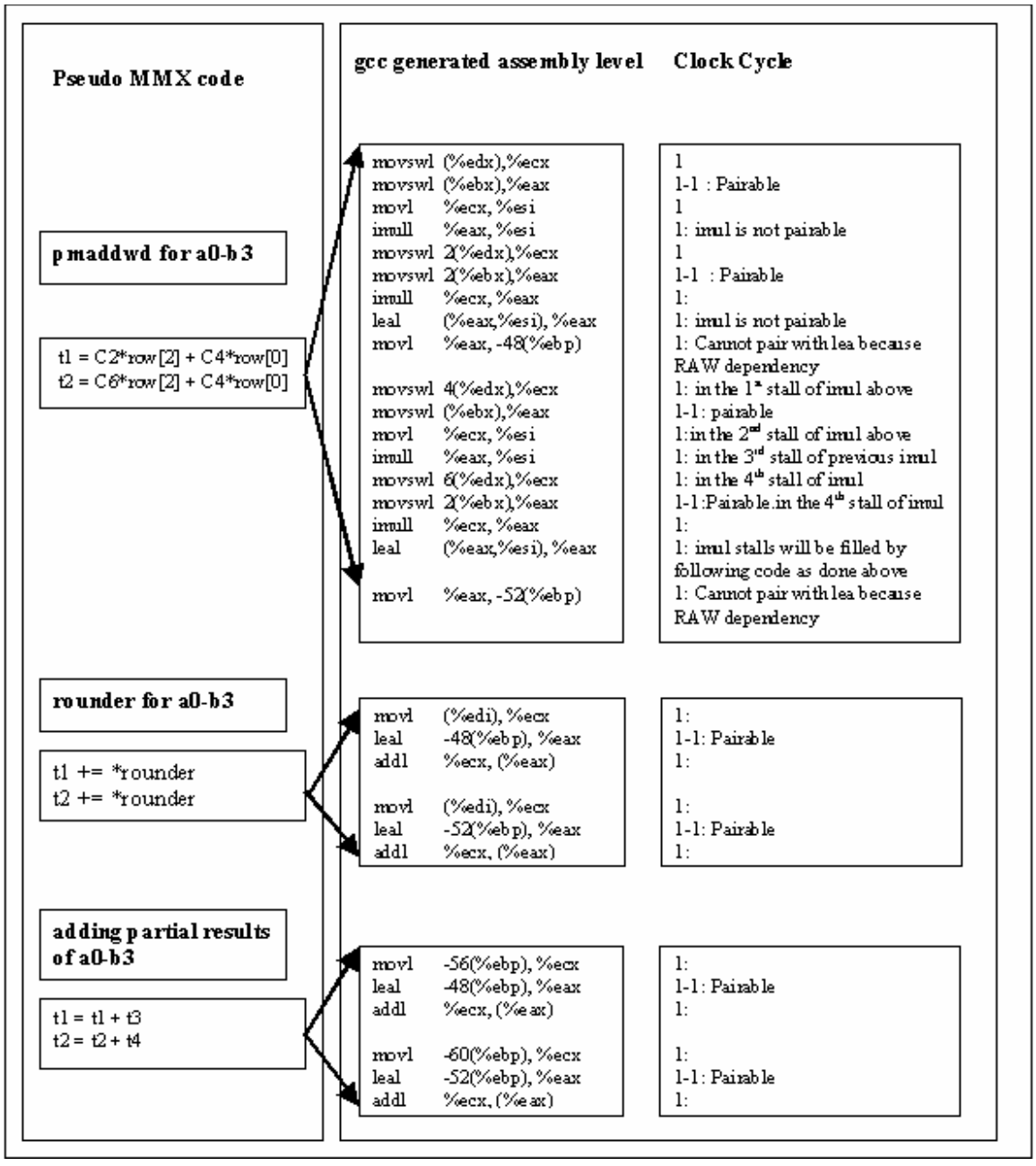

Figure 3-5. Relationship Of C Code And Assembly Level Code To Calculate a0-b3

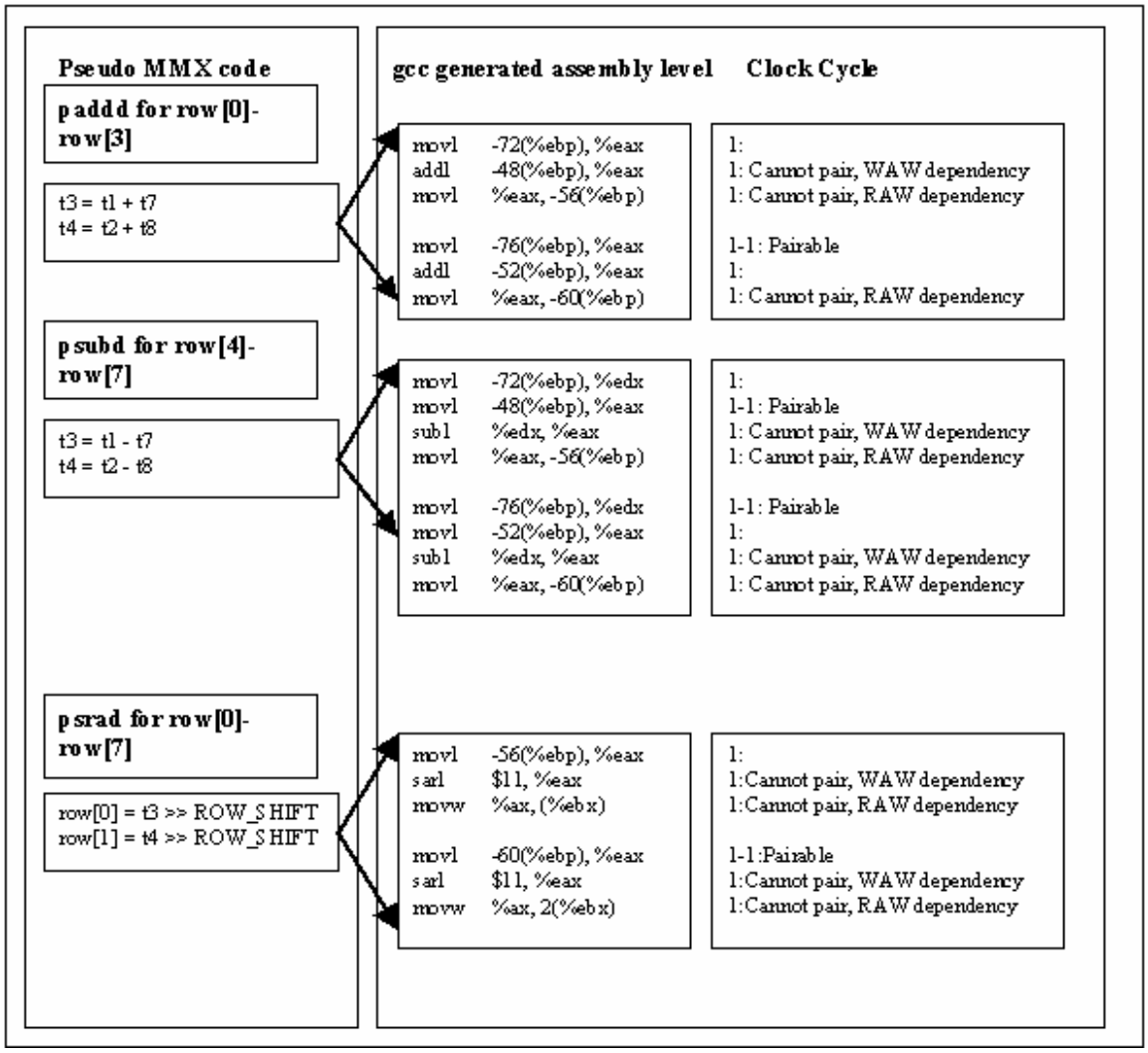

Figure 3-6. Relationship Of C Code And Assembly Level Code To Calculate row[0]-row[7]

One important aspect that we would like to point out here is the implicit data moves embedded in the assembly level code. These moves are generated by the compiler and are characterized in the architectural speedup of that instruction, which causes the cycle count and hence the speedup of an instructions to increase. For example pmaddwd performances 4 multiplication and 2 addition operations and one would think that the speedup due to this new MMX instruction would be 6. But we need to consider the data moves which are hidden under the equivalent C-code for pmaddwd too. This causes the

cycle count and hence the speedup to be more than 6. The estimation for the architectural speedup of some instructions are summarized in Table 4-2.

Estimating the overall execution time. Using the time components measured on the existing system we can now estimate the total execution time with the new set of computational instructions on a future system. The estimated execution time is equal to the summation of the time taken by the sequential code, the time taken by data moves and the estimated time taken by the new computational instructions.

Verifying results. To verify the proposed method we consider four existing MMX instructions, PMADDWD, PADDD, PSUBD and PSRAD, as new SIMD computational instructions. The estimated total execution time is verified against the measured time on the system using the MMX-code. (Note: this step cannot be done for true new instructions.) One difficulty in obtaining a perfect performance projection is due to the inaccuracy of the estimated speedup of the new computational instructions. Due to the processor pipeline designs, out-of-order execution, etc, it is very difficult to accurately provide a single speedup number. One alternative approach is to draw a speedup curve based on a range of architecture speedup of the new computational instruction. Such a speedup curve, along with the estimated data moves can help architects make proper design tradeoffs.

3.3 Application And Tools Used

The system that we use to prove this methodology is a Dell PowerEdge 2500 with Pentium III 1GHz processor, 512MB RAM, 16KB L1 instrcution and 16KB L1 Data Cache, running Red Hat Linux release 7.2 [Enigma], O.S Release 2.4.7-10.

The application that we use to describe our methodology is the code of the IDCT algorithm of the mpeg2dec and libmpeg2 software. mpeg2dec is an mpeg-1 and mpeg-2

video decoder. mpeg2dec and libmpeg2 are released under the GPL license. libmpeg2 is the heart of the mpeg decoder. The main goals in libmpeg2 development are conformance, speed, portability and reuseability. Most of the code is written in C, making it fast and easily portable. The application is freely available for download at http://libmpeg2.sourceforge.net/

3.3.1 Compiler Used And Its Options

We use the GCC[13] compiler to compile the C files. "GCC" is a common shorthand term for the GNU Compiler Collection. This is both the most general name for the compiler, and the name used when the emphasis is on compiling C programs. When we invoke GCC, it normally does preprocessing, compilation, assembly and linking. The "overall options" allow us to stop this process at an intermediate stage. For example, the c option says not to run the linker. Then the output consists of object files output by the assembler.

The compilation process consists of up to four stages: preprocessing, compilation proper, assembly and linking, always in that order. The first three stages apply to an individual source file, and end by producing an object file; linking combines all the object files into an executable file. The gcc program accepts options and file names as operands.

Options can be used to optimize the compilation (-O), to specify directories to search for header files, for libraries and for parts of the compiler (-I), compile or assemble the source files, but not link them and to output an object file for each source file (-c), to stop after the stage of compilation proper and not do the assemble stage with the output in the form of an assembler code file for each non-assembler input file specified (-S). There are some debugging options like –g to produce debugging

information in the operating system's native format we use, -pg to generate extra code to write profile information suitable for the analysis program prof.

3.3.2 gprof

We use the gprof [11] utility to produce an execution profile of a program. The effect of called routines is incorporated in the profile of each caller. The profile data is taken from the call graph profile file that is created by programs compiled with the -xpg option of cc(1), or by the -pg option with other compilers. Profiling allows us to learn where the program spent its time and which functions called which other functions while it was executing. This information can show us which pieces of the program are slower than we expected, and might be candidates for rewriting to make the program execute faster. It can also tell us which functions are being called more or less often than we expected. The profiler uses information collected during the actual execution of the program. The "-l" option enables line-by-line profiling, which causes histogram hits to be charged to individual source code lines, instead of functions. The `-A' option causes gprof to print annotated source code. We use this tool to verify that all the instructions in our segments execute the number of times we anticipated them to.

The command "gprof -l -x -A a.out gmon.out" produces the source code annotated with the execution count of each instruction to its left as

```
…….
8000000 ->    movq_m2r(*(row+offset), mm2);          // mm2 = x3 x2 x1 x0
8000000 ->    movq_m2r(*(row+offset+4), mm5);        // mm5 = x7 x6 x5 x4
8000000 ->    movq_r2r(mm2, mm0);                    // mm0 = x3 x2 x1 x0
…….
```

3.3.3 Clock ()

C has a clock () function that does not take any arguments and returns a value of type clock_t, which is defined on /usr/include/sys/types.h and /usr/include/time.h as int. This function returns CPU time that elapsed since the execution of the program commenced and can be used to measure the time for code segments. The returned time is not in seconds. It is in clock cycles. There is a constant CLOCKS_PER_SEC defined on /usr/include/time.h, which tells us how many clock cycles there are per second. So, in order to find out how many CPU seconds we have used so far we have to divide the result obtained by calling CLOCKS_PER_SEC.

The macro CLOCKS_PER_SEC returns a value that is the number of clocks per second measured by the clock function. POSIX [14] requires that this value be one million independent of the actual resolution.

In typical usage, we call the clock function at the beginning and end of the interval we want to time, subtract the values, and then divide by CLOCKS_PER_SEC (the number of clock ticks per second) to get processor time, like this

```
#include <time.h>
clock_t start, end;
double cpu_time_used;
start = clock();
... /* Do the work. */
end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

3.3.4 RDTSC

The cycle count for code segments can also be measured by another technique using RDTSC [12]

Starting with the Pentium processor, Intel processors allow the programmer to access a time-stamp counter. The time-stamp counter keeps an accurate count of every cycle that occurs on the processor. To access this counter, programmers can use the RDTSC (read time-stamp counter) instruction. This instruction loads the high-order 32 bits of the register into EDX, and the low-order 32 bits into EAX

The code for the RDTSC is

```
void access_counter(unsigned *hi, unsigned *lo)
{
asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
: "=r" (*hi), "=r" (*lo)
:
: "%edx", "%eax");
}
```

RDTSC is used to verify the cycle counts for code segments, which is used to estimate the architectural speedup.

CHAPTER 4
CASE STUDY

In this chapter we describe our case study to prove our methodology described in the previous chapter. We use the architectural details mentioned in Chapter 2 to calculate the speedup of the vectorizable segment of the idct_row method. Using this information and the measurements made we estimate within 8% accuracy the execution time of the MMX-code.

4.1 The Application Program – IDCT

We chose IDCT as our case study because it is widely used in signal and multimedia processing. The repetitive compute-intensive operations on 16-bit data make it a good choice to be vectorized. The operations performed in the butterfly model for IDCT consist of a set of multiplications and additions that can be vectorized to pmaddwd and paddd in MMX.

For example

```
b0 = table[8]*row[1]  + table[9]*row[3] + table[12]*row[5] + table[13]*row[7];
b1 = table[10]*row[1] + table[11]*row[3] + table[14]*row[5] + table[15]*row[7];
```

can be vectorized to

```
pmaddwd_r2r (mm5, mm1);
pmaddwd_r2r (mm6, mm7);
paddd_r2r (mm7, mm1);
```

Since most of these operations are performed in the idct_row we selected the row method to prove our methodology.

4.2 Measurements And Performance

The original IDCT file, idct_mmx.c consists of 2 key methods idct_row and idct_coln. We identify key operations in the idct_row method that can be vectorized by new instructions. The source data of idct_row are 16-bit quantity. Since MMX registers are 64 bits wide and mmx instructions can operate on 2 mmx registers, each with four 16-bit data, some operations can be grouped together and a SIMD instruction can be used to operate on them simultaneously. This will speedup the idct_row method and hence the overall application.

We focus on vectorizing idct_row using the "new" pmaddwd, paddd, psubd, psrad instructions. We assume that the idct_coln method has already been vectorized using existing MMX technology and is considered as the sequential code segment. The performance timing results from the measurements and projections are summarized in Table 4-1. During the experiement, we found a separate timing component that belongs to the vectorized idct_row, but cannot benefit from the SIMD computational instructions. Program constructs, such as procedure calls, loop controls, etc belongs to this category, which is referred to as "Others" in the table. The "Others" portion behaves very similar to the sequential code and its execution time is generally unchanged in the MMX-code.

The C-code takes 1.56 seconds to execute. The sequential portion, idct_col, takes 0.13 seconds and the target vectorizable code plus "Others" takes 1.43 seconds to execute. The Pseudo MMX-code, that has the new pmaddwd, paddd, psubd and psrad instructions with the equivalent C-code, takes 1.69 seconds. Therefore the data moves are estimated at 0.13 seconds. The Cripple MMX-code that removes the computational instructions from the MMX-code takes 0.38 seconds to execute. Therefore the

vectorizable code in idct_row takes 1.31 seconds. Hence we are aiming to vectorize code that takes about 84% of the total execution time.

Table 4-1. Performance Measurement And Projection–A Case Study

Program	Timing Component	Time (sec)
1. C-code	Sequential (idct_col) + Vectorizable segment (idct_row) + Others (call to idct_row)	1.56
	Sequential (idct_col)	0.13
	Vectorizable segment (idct_row) + Others (call to idct_row)	1.43
2. Pseudo MMX code (Computational replaced by C-equivalent)	Sequential (idct_col) + Vectorizable segment (idct_row) + Moves + Others (call to idct_row)	1.69
	Moves = (2)-(1) = 1.69-1.56	0.13
3. Crippled MMX code (computational removed)	Sequential (idct_col) + Moves + Others (call to idct_row)	0.38
	Vectorizable segment (2) –(3) = 1.69-0.38	1.31
	Others = (3) – Moves – Sequential = 0.38-0.13-0.13	0.12
4. Estimated Execution time	(0.13 +0.12+0.13) + 1.31/8.09	0.5419
5. Overall Speedup	1.56/0.5419	2.878
6. MMX code (for verification)	Sequential + moves + Others + computational	0.505

All measurements are for 1 Million execution of the idct function.

In the Cripple MMX-code we remove the data move instructions to measure the execution time as 0.252 seconds. This time is for the idct_col and the calls to idct_row. We further remove all the calls to the idct_row method to get 0.132 seconds. Hence the call to the idct_row method takes 0.12 seconds.

We then calculate the speedup of each new SIMD instructions as described in section 3.2, Figure 3-5, Figure 3-6 and the results are summarized in Table 4-2.

Table 4-2. Estimating The Speedup For New SIMD Instructions

Code Segment	Operations	C code Cycles	MMX code Cycles	Speedup	Occurrence Count of instruction
pmaddwd for a0-b3	4 multiplications and 2 additions	14	1	14	8
paddwd for rounder	2 additions	4	1	4	2
paddwd for a0-b3	2 additions	4	1	4	4
paddwd /psubwd for row[]	2 additions 2 subtractions	5 6	1 1	5.5	2 2
prad for row[]		5	1	5	4

Each pmaddwd equivalent C-code takes 14 cycles, when its assembly level code is analyzed using the architectural information of the current system. The new MMX instruction will take 1 cycle. This information, the cycle count for the new media instructions, needs to be provided by the system architect. Hence the estimated speedup of pmaddwd is 14. Similarly the speedup of each new SIMD instruction can be estimated. We then calculate a weighted speedup by averaging the sum of the products of the individual speedup and the number of occurrences of the instruction.

$$WeightedSpeedup = \frac{(14*8 + 4*2 + 4*4 + 5.5*4 + 5*4)}{22}$$

$$= 8.09$$

We calculate a weighted speedup using the individual speedup of instructions. The weighted speedup for our new SIMD instructions is 8.09.

Finally, the estimated time using the new computational instructions can be calculated as follows

$$Estimated\ Execution\ Time = Time(sequential\ code) + Time(data\ moves) + Time(Others) + \frac{Time(vectorizable\ code)}{Speedup\ of\ the\ vectorizable\ code}$$

$$= 0.13 + 0.13 + 0.12 + \frac{1.31}{8.09}$$
$$= 0.5419$$

This result is within 8 % of the actual result. Where speedup of the vectorizable code, is the architectural speedup of the computational instructions. The overall calculated speedup is

$$Speedup = \left(\frac{1.56}{0.5419} \right)$$
$$= 2.878$$

Using all the estimations done so far an architect can do sensitivity study as shown in Figures 4-1, 4-2 and 4-3.

Figure 4-1 plots the overall execution time and speedup given various improvements of the computational instructions. As observed from the figure, with 2 times the performance improvement for the new computational instructions, the overall speedup is close to 1.5, while the speedup is just over 3 given 10 times improvement of the computational instructions. The data moves alone take about 8% of the total execution time without any improvement on the vectorizable code. The percentage increases to 25.4 % when the new computational instructions get 10 times of performance improvement.

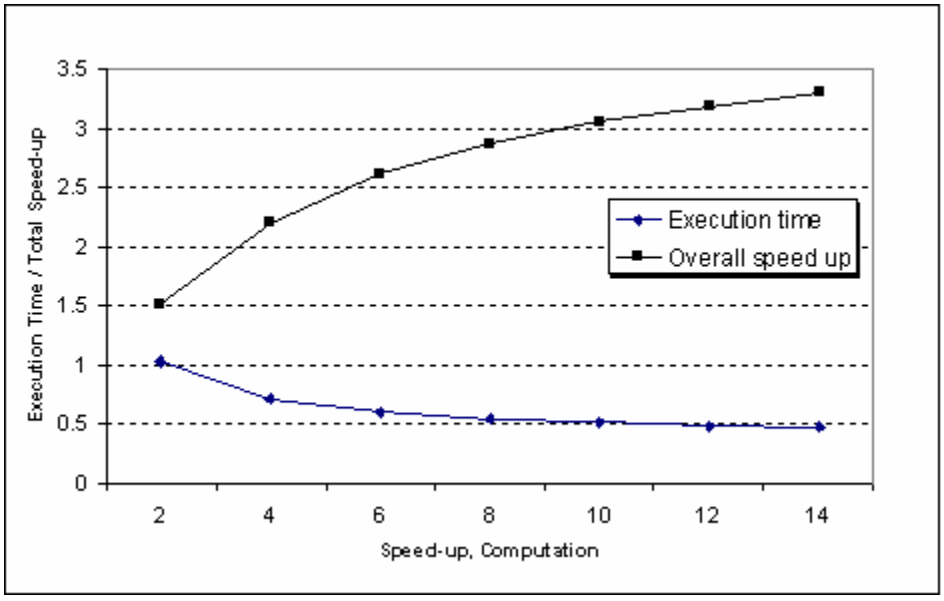

Figure 4-1. Execution Time and Speedup With Respect To Performance Of New Computational Instructions

Figure 4-2 plots the overall execution time and speedup given various improvements for the data move instructions.

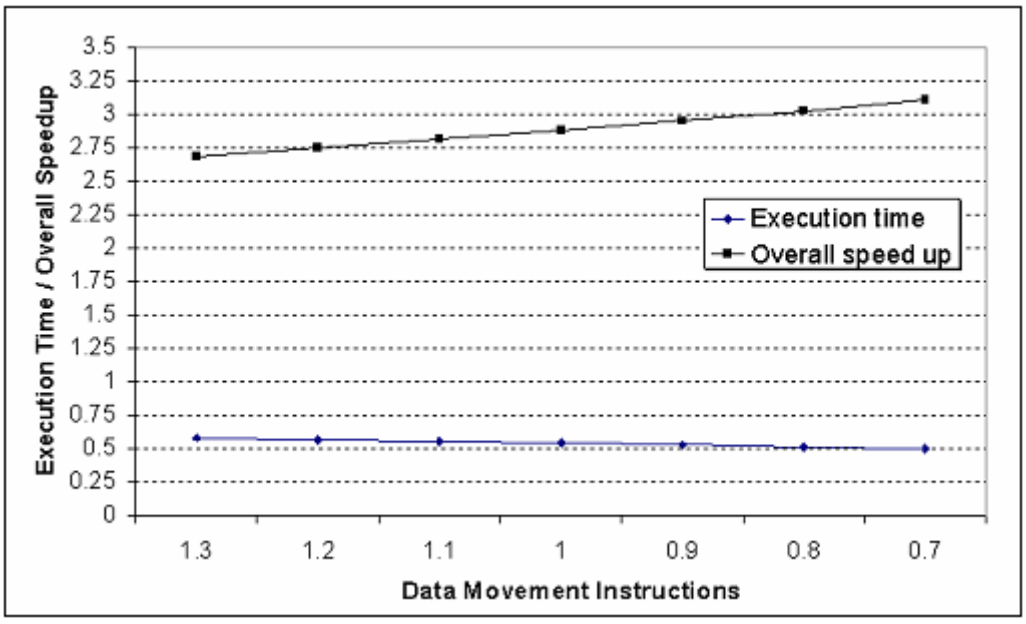

Figure 4-2. Execution Time And Speedup With Respect To Performance Of New Data Movement Instructions

Using such a chart architects can analyze the variation in overall speedup with deterioration and improvement in the execution time of data movement instructions.

The above two figures can be combined into the Figure 4-3, which shows the variations in overall speedup with the change in data move instructions and the architectural speedup.

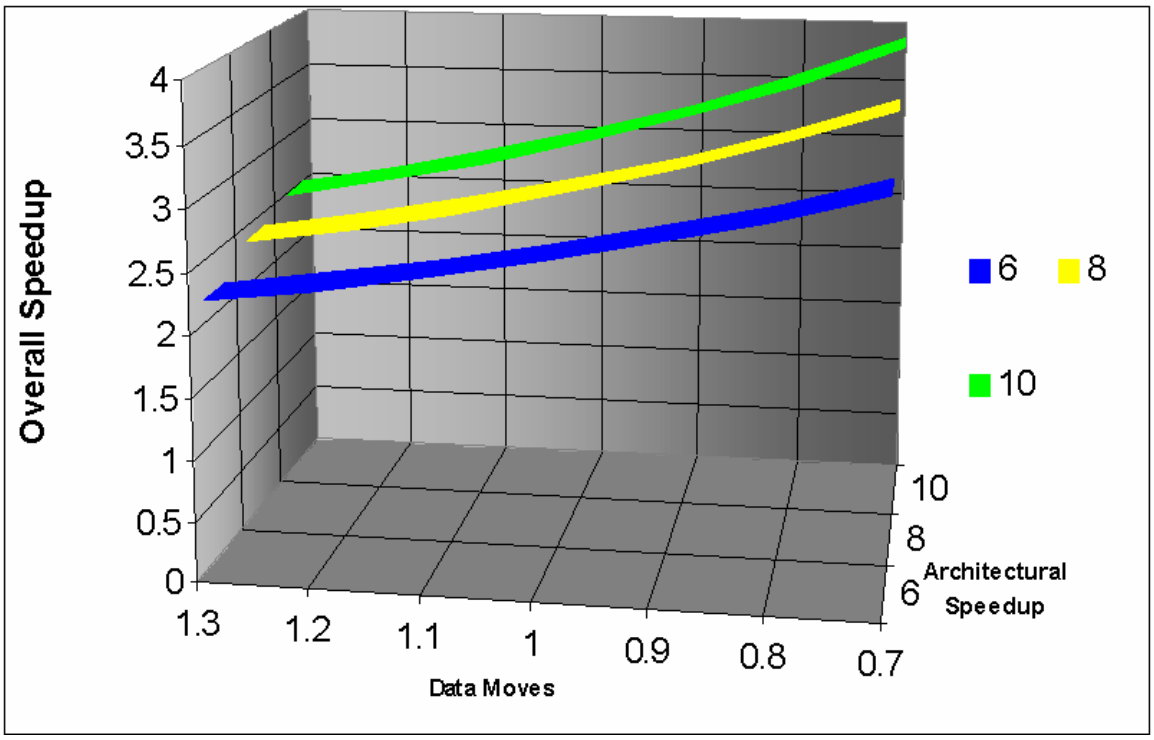

Figure 4-3. Overall Speedup With Respect To Performance Of New Data Movement Instructions And Architectural Speedup

CHAPTER 5
SUMMARY AND FUTURE WORK

Digital signal and multimedia processing is widely used in real life toady and many microprocessors support such applications. Emerging applications will require the existing instruction set to be enhanced. Estimating design tradeoff for new SIMD instructions will help architects select instructions most suitable for the application.

5.1 Summary

We presented a performance estimation method for new media instructions for a future system. Instead of using a cycle-accurate simulator, our method is based on workload characterization, benchmarking and measurement on a current system. Given a range of performance improvement of the new media instructions, the proposed method can provide a range of speedups of using the new media instructions.

We break up the application into different segments and pick the one that can be vectorized by the new computation instructions. After measuring the timings of different segments on the current system, we estimate the clock cycles the new instructions take by using the architectural details of the future system. Using this count and the measured cycle count of the vectorizable code we calculate the speedup that the computation instructions offers. We calculate the total execution time of the code with the new media instruction using the timings and measurement done. This execution time is an estimate and can vary depending on the gap in the memory hierarchy and architecture of the current and the future system. We estimated the execution time of a program using our methodology and the results were within 8% of the actual measured times. This

difference is small as we used the current system as the future system, and hence eliminated the difference in time, which could have occurred due to the architectural and memory hierarchy differences.

Our methodology can be used to quickly and roughly estimate the performance improvement due to the addition of new SIMD instructions by characterizing and measuring time for different segments on an existing system. The computer architect can do sensitivity study to analyze the effects of different architectural speedups and times taken by data move instructions.

5.2 Future Work

One can take this methodology to the next level by running larger size applications and characterizing their effects. Estimating the change in performance due to new data moves can also be a very interesting topic.

APPENDIX A
OPERATION OF MMX INSTRUCTIONS

Operations of a few MMX instructions on a Pentium III are shown in the Figure A1

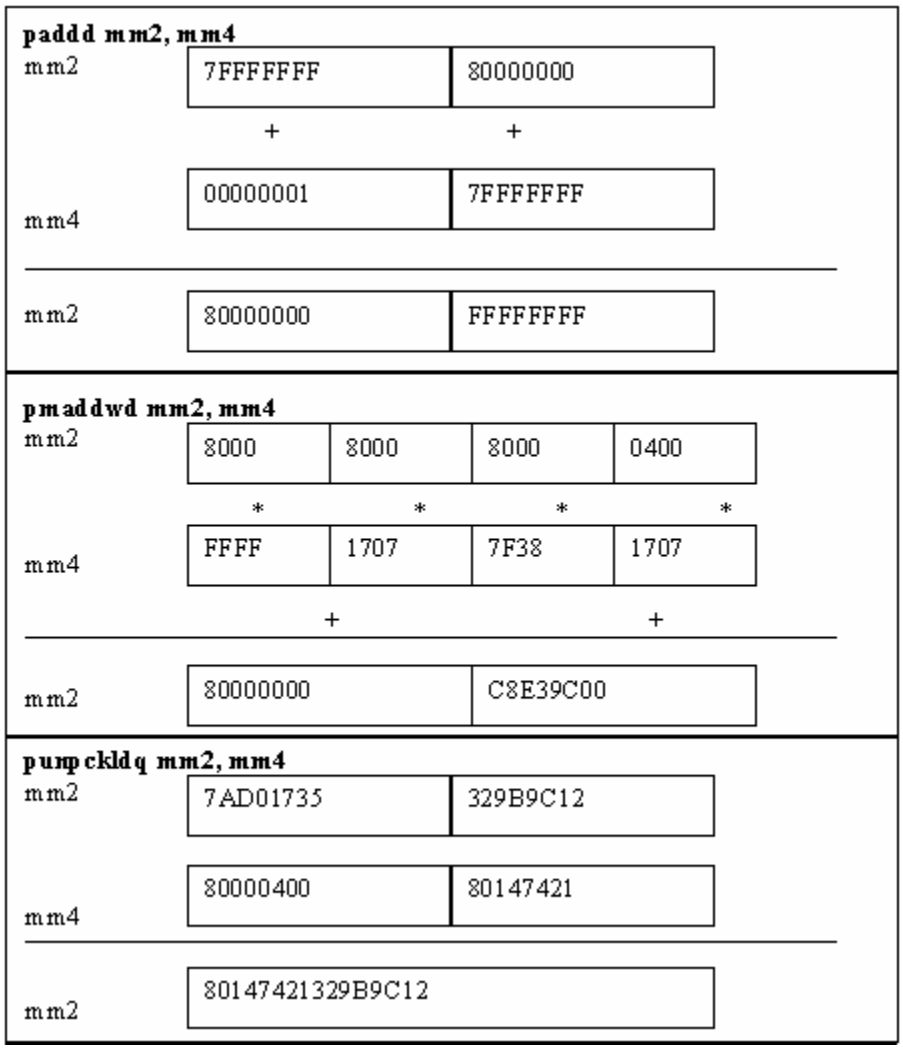

Figure A1. Operation Of MMX Instructions

The padd instructions add the data elements (bytes, words, or doublewords) in the two operands. If the sum is greater than the data-range limit for the data type, it wraps

around. The source operand can be an MMX register or an operand in memory. The destination operand must be an MMX register.[15]

The pmaddwd instruction multiplies each of the 4 signed words in the source operand by each of the 4 signed words in the destination operand. This creates four double words. The lower doublewords are then summed and stored in the lower doubleword of the destination register. The higher doublewords are summed and stored in the higherdouble word of the destination register. The result wraps around to 80000000 only when all the source data elements are 8000. The source operand can be an MMX register or an operand in memory. The destination operand must be MMX register.[15]

The punpckl instructions convert smaller data elements (bytes, words, or double words) into larger data elements, by coping and interleaving the low-order elements of the source and destination operands. The high order data elements are ignored. The source operand must be MMX register or an operand in memory. The destination operand must be MMX register.[15]

Data transfer instructions are used to copy data to and from MMX registers, copy data from MMX register to memory or from memory to a MMX register, copy data from a MMX register to an integer register or from an integer register to a MMX register (movd only). The data transfer instructions are the only MMX instructions that can have a memory address as a destination operand.[15]

APPENDIX B
CODE USED

The C-code [16] for the row method, the modified row method of MMX from the mpeg2 decoder and its developed equivalent C code are shown below. For the complete IDCT program, please refer the idct_mmx.c file in mpeg2dec [16].

```
static inline void idct_row (int16_t * row, int offset, int16_t * table, int32_t * rounder)
{
   int a0, a1, a2, a3, b0, b1, b2, b3;
   row += offset;
   a0 = table[4]*row[0]+table[2]*row[2] +  table[4]*row[4]+table[6]*row[6] + *rounder;
   a1 = table[4]*row[0]+table[6]*row[2]  -  table[4]*row[4]-table[2]*row[6] + *rounder;
   a2 = table[4]*row[0]-table[6]*row[2]   -  table[4]*row[4]+table[2]*row[6] + *rounder;
   a3 = table[4]*row[0]-table[2]*row[2]  +  table[4]*row[4]-table[6]*row[6] + *rounder;

   b0 = table[1]*row[1]+table[3]*row[3] +  table[5]*row[5] + table[7]*row[7];
   b1 = table[3]*row[1]-table[7]*row[3]   -  table[1]*row[5]-table[5]*row[7];
   b2 = table[5]*row[1]-table[1]*row[3]  +  table[7]*row[5]+table[3]*row[7];
   b3 = table[7]*row[1]-table[5]*row[3]  +  table[3]*row[5]-table[1]*row[7];

   row[0] = (a0 + b0) >> ROW_SHIFT;
   row[1] = (a1 + b1) >> ROW_SHIFT;
   row[2] = (a2 + b2) >> ROW_SHIFT;
   row[3] = (a3 + b3) >> ROW_SHIFT;
   row[4] = (a3 - b3) >> ROW_SHIFT;
   row[5] = (a2 - b2) >> ROW_SHIFT;
   row[6] = (a1 - b1) >> ROW_SHIFT;
   row[7] = (a0 - b0) >> ROW_SHIFT;
}

static inline void idct_row (int16_t  row[], int offset, int16_t  table[], int32_t  rounder[])
{
   // from mmx_row_head
   movq_m2r(*(row+offset), mm2);          // mm2 = x6 x4 x2 x0
   movq_m2r(*(row+offset+4), mm5);        // mm5 = x7 x5 x3 x1
   movq_r2r(mm2, mm0);                    // mm0 = x6 x4 x2 x0
   movq_m2r(*table, mm3);                 // mm3 = C6 C4 C2 C4
   movq_r2r(mm5, mm6);                    // mm6 = x7 x5 x3 x1
```

```
    punpckldq_r2r (mm0, mm0);           // mm0 = x2 x0 x2 x0
    movq_m2r (*(table+4), mm4);         // mm4 = -C2 -C4 C6 C4
    pmaddwd_r2r (mm0, mm3);             // mm3 = C4*x0+C6*x2 C4*x0+C2*x2
    movq_m2r (*(table+8), mm1);         // mm1 = -C7 C3 C3 C1
    punpckhdq_r2r (mm2, mm2);           // mm2 = x6 x4 x6 x4

    // from mmx_row
    pmaddwd_r2r (mm2, mm4);             // mm4 = -C4*x4-C2*x6 C4*x4+C6*x6
    punpckldq_r2r (mm5, mm5);           // mm5 = x3 x1 x3 x1
    pmaddwd_m2r (*(table+16), mm0);     // mm0 = C4*x0-C2*x2 C4*x0-C6*x2
    punpckhdq_r2r (mm6, mm6);           // mm6 = x7 x5 x7 x5
    movq_m2r (*(table+12), mm7);        // mm7 = -C5 -C1 C7 C5
    pmaddwd_r2r (mm5, mm1);             // mm1 = C3*x1-C7*x3 C1*x1+C3*x3

    paddd_m2r (*rounder, mm3);          // mm3 += rounder
    pmaddwd_r2r (mm6, mm7);             // mm7 = -C1*x5-C5*x7 C5*x5+C7*x7
    pmaddwd_m2r (*(table+20), mm2);     // mm2 = C4*x4-C6*x6 -C4*x4+C2*x6
    paddd_r2r (mm4, mm3);               // mm3 = a1 a0 + rounder
    pmaddwd_m2r (*(table+24), mm5);     // mm5 = C7*x1-C5*x3 C5*x1-C1*x3
    movq_r2r (mm3, mm4);                // mm4 = a1 a0 + rounder
    pmaddwd_m2r (*(table+28), mm6);     // mm6 = C3*x5-C1*x7 C7*x5+C3*x7
    paddd_r2r (mm7, mm1);               // mm1 = b1 b0
    paddd_m2r (*rounder, mm0);          // mm0 += rounder
    psubd_r2r (mm1, mm3);               // mm3 = a1-b1 a0-b0 + rounder
    psrad_i2r (ROW_SHIFT, mm3);         // mm3 = y6 y7
    paddd_r2r (mm4, mm1);               // mm1 = a1+b1 a0+b0 + rounder
    paddd_r2r (mm2, mm0);               // mm0 = a3 a2 + rounder
    psrad_i2r (ROW_SHIFT, mm1);         // mm1 = y1 y0
    paddd_r2r (mm6, mm5);               // mm5 = b3 b2
    movq_r2r (mm0, mm7);                // mm7 = a3 a2 + rounder
    paddd_r2r (mm5, mm0);               // mm0 = a3+b3 a2+b2 + rounder
    psubd_r2r (mm5, mm7);               // mm7 = a3-b3 a2-b2 + rounder

    // from mmx_row_tail
    psrad_i2r (ROW_SHIFT, mm0);         // mm0 = y3 y2
    psrad_i2r (ROW_SHIFT, mm7);         // mm7 = y4 y5
    packssdw_r2r (mm0, mm1);            // mm1 = y3 y2 y1 y0
    packssdw_r2r (mm3, mm7);            // mm7 = y6 y7 y4 y5
    movq_r2m (mm1, *(row+offset));      // save y3 y2 y1 y0
    movq_r2r (mm7, mm4);                // mm4 = y6 y7 y4 y5
    pslld_i2r (16, mm7);                // mm7 = y7 0 y5 0
    psrld_i2r (16, mm4);                // mm4 = 0 y6 0 y4
    por_r2r (mm4, mm7);                 // mm7 = y7 y6 y5 y4
    movq_r2m (mm7, *(row+offset+4));    // save y7 y6 y5 y4
}
```

```
static inline void idct_row (int16_t  row[], int offset, int16_t table[], int32_t  rounder[])
{
   int32_t  t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12,t13,t14,t15,t16;
   row += offset;
   // pmaddwd_r2r (mm0, mm3);
   t1 = table[0]*row[0] + table[1]*row[1];             //  C4*x0+C2*x2  =a0a
   t2 = table[2]*row[0] + table[3]*row[1];             //  C4*x0+C6*x2  =a1a

   // pmaddwd_r2r (mm2, mm4);
   t3 = table[4]*row[2] + table[5]*row[3];             //  C4*x4+C6*x6  =a0b
   t4 = table[6]*row[2] + table[7]*row[3];             // -C4*x4-C2*x6  =a1b
   // pmaddwd_m2r (*(table+16), mm0);
   t5 = table[16]*row[0] + table[17]*row[1];           //  C4*x0-C6*x2  =a2a
   t6 = table[18]*row[0] + table[19]*row[1];           //  C4*x0-C2*x2  =a3a
   // pmaddwd_r2r (mm5, mm1);
   t7 = table[8]*row[4]  + table[9]*row[5];            //  C1*x1+C3*x3  =b0a
   t8 = table[10]*row[4] + table[11]*row[5];           //  C3*x1-C7*x3  =b1a

   // paddd_m2r (*rounder, mm3);   mm3 += rounder
   t1+= *rounder;
   t2+= *rounder;

   // pmaddwd_r2r (mm6, mm7);
   t9 = table[12]*row[6] + table[13]*row[7];           //  C5*x5+C7*x7  =b0b
   t10= table[14]*row[6] + table[15]*row[7];           // -C1*x5-C5*x7  =b1b
   // pmaddwd_m2r (*(table+20), mm2);      mm2
   t11= table[20]*row[2] + table[21]*row[3];           // -C4*x4+C2*x6  =a2b
   t12= table[22]*row[2] + table[23]*row[3];           //  C4*x4-C6*x6  =a3b

   // paddd_r2r (mm4, mm3);        mm3 = a1 a0 + rounder
   t1 = t1 + t3;  // a0 = a0a + rounder + a0b
   t2 = t2 + t4;  // a1 = a1a + rounder + a1b

   // pmaddwd_m2r (*(table+24), mm5);
   t13= table[24]*row[4] + table[25]*row[5];           //  C5*x1-C1*x3  =b2a
   t14= table[26]*row[4] + table[27]*row[5];           //  C7*x1-C5*x3  =b3a
   // pmaddwd_m2r (*(table+28), mm6);
   t15= table[28]*row[6] + table[29]*row[7];           //  C7*x5+C3*x7  =b2b
   t16= table[30]*row[6] + table[31]*row[7];           //  C3*x5-C1*x7  =b3b

   // paddd_r2r (mm7, mm1);        mm1 = b1 b0
   t7 = t7 + t9;                                       // b0 = b0a + b0b
   t8 = t8 + t10;                                      // b1 = b1a + b1b
   // paddd_m2r (*rounder, mm0);   mm0 += rounder
   t5 += *rounder;
   t6 += *rounder;
```

```
    // psubd_r2r (mm1, mm3);        mm3 = a1-b1 a0-b0 + rounder
    t3 = t1 - t7;
    t4 = t2 - t8;

    // psrad_i2r (ROW_SHIFT, mm3);  mm3 = y6 y7
    row[6] = t4 >> ROW_SHIFT;
    row[7] = t3 >> ROW_SHIFT;

    // paddd_r2r (mm4, mm1);        mm1 = a1+b1 a0+b0 + rounder
    t3 = t1+t7;
    t4 = t2+t8;
    // paddd_r2r (mm2, mm0);        mm0 = a3 a2 + rounder
    t5 = t5 + t11;                                  // a2 = a2a+rounder + a2b
    t6 = t6 + t12;                                  // a3 = a3a+rounder + a3b

    // psrad_i2r (ROW_SHIFT, mm1);  mm1 = y1 y0
    row[0] = t3 >> ROW_SHIFT;
    row[1] = t4 >> ROW_SHIFT;

    // paddd_r2r (mm6, mm5);        mm5 = b3 b2
    t13 = t13 + t15;                                // b2 = b2a + b2b
    t14 = t14 + t16;                                // b3 = b3a + b3b
    // paddd_r2r (mm5, mm0);        mm0 = a3+b3 a2+b2 + rounder
    t3 = t5+t13;
    t4 = t6+t14;
    // psubd_r2r (mm5, mm7);        mm7 = a3-b3 a2-b2 + rounder
    t9  = t5 - t13;
    t10 = t6 - t14;

    // psrad_i2r (ROW_SHIFT, mm0);  mm0 = y3 y2
    row[2] = t3 >> ROW_SHIFT;
    row[3] = t4 >> ROW_SHIFT;
    // psrad_i2r (ROW_SHIFT, mm7);  mm7 = y4 y5
    row[4] = t10  >> ROW_SHIFT;
    row[5] = t9 >> ROW_SHIFT;
}
```

LIST OF REFERENCES

[1] Intel Corp., Intel MMX technology at a glance, order number 243100-003, June 1997.

[2] Sundararajan Sriram, Ching-Yu Hung, "MPEG-2 video decoding on the TMS320C6X DSP architecture," Texas Instruments, Dallas, Texas, 1998.

[3] Yen-Kuang Chen, Nicholas Yu, Birju Shah, "Digital signal processing on MMX technology," Microprocessor research labs, Intel Corp, New York, 2000.

[4] Thomas Chen, MPEG2 Technology, University of Southern California, 2000. (Online: http://www-classes.usc.edu/engr/ee-ep/597/student_b.ppt). Accessed 09/16/2002.

[5] Dr. Konrad Froitzheim, Heiner Wolf, "A knowledge-based approach to JPEG acceleration," Department of Distributed Systems, University of Ulm, Germany.

[6] Millind Mittal, Alex Peleg, Uri Weiser, "MMX technology architecture overview," Intel technology journal, 1997, pp.1-12.

[7] Alex Peleg, Uri Weiser, "MMX technology extension to the Intel architecture," IEEE Micro, 1996, pp.42–50 vol. 16, No. 4.

[8] Intel Corp., Intel architecture optimization reference manual, Order number 245127-001, 1998.

[9] Agner Fog, "How to optimize for the Pentium family of processors," 1996, (Online http://www.asdf.org/~fatphil/x86/pentopt/). Accessed 09/16/2002.

[10] Jar-Ferr Yang, "An efficient two-dimensional inverse discrete cosine transform algorithm for HDTV receivers," IEEE, vol. 5, no.1, Feb 1995.

[11] Free Software Foundation, GNU gprof, 1998. (Online http://www.gnu.org/manual/gprof-2.9.1/gprof.html). Accessed 09/16/2002.

[12] Intel Corp., "Using the RDTSC instruction for performance monitoring," 1997 (Online http://cedar.intel.com/software/idap/media/pdf/rdtscpm1.pdf). Accessed 09/16/2002.

[13] Free Software Foundation, GCC online documentation, 2002 (Online http://gcc.gnu.org/onlinedocs). Accessed 09/16/2002.

[14] IEEE Org., IEEE POSIX Certification Authority, 2002. (Online http://standards.ieee.org/regauth/posix/). Accessed 09/16/2002.

[15] Intel Corp., Introduction to Intel MMX technology tutorial, (Online: http://www.intel.com/software/products/college/ia32/mmxdown.htm). Accessed 09/16/2002.

[16] Aaron Holtzman, Michel Lespinasse, libmpeg2 - a free MPEG-2 video stream decoder, 2002. (Online http://libmpeg2.sourceforge.net/). Accessed 09/16/2002.

BIOGRAPHICAL SKETCH

Adil A. Gheewala was born in Bombay, India. He received his Bachelor of Engineering degree in computer engineering from Fr. Conceicao Rodrigues College of Engineering, Bombay, India, in 1999. He will receive his Master of Science degree in computer engineering from the Computer and Information Science and Engineering Department, University of Florida, Gainesville, in December 2002.