Evaluating MMX Technology Using DSP and Multimedia Applications

Ravi Bhargava, Lizy K. John, Brian L. Evans and Ramesh Radhakrishnan *
Electrical and Computer Engineering Department
The University of Texas at Austin
{ravib,ljohn,bevans,radhakri}@ece.utexas.edu

Keywords: Digital Signal Processing, Machine Measurement, MMX, Performance Monitoring, Workload Characterization.

Abstract

Many current general purpose processors are using extensions to the instruction set architecture to enhance the performance of digital signal processing (DSP) and multimedia applications. In this paper, we evaluate the X86 architecture's multimedia extension (MMX) instruction set on a set of benchmarks. Our benchmark suite includes kernels (filtering, fast Fourier transforms, and vector arithmetic) and applications (JPEG compression, Doppler radar processing, imaging, and G.722 speech encoding). Each benchmark has at least one non-MMX version in C and an MMX version that makes calls to an MMX assembly library. The versions differ in the implementation of filtering, vector arithmetic, and other relevant kernels. The observed speedup for the MMX versions of the suite ranges from less than 1.0 to 6.1. In addition to quantifying the speedup, we perform detailed instruction level profiling using Intel's VTune profiling tool. Using VTune, we profile static and dynamic instructions, microarchitecture operations, and data references to isolate the specific reasons for speedup or lack thereof. This analysis allows one to understand which aspects of native signal processing instruction sets are most useful, the current limitations, and how they can be utilized most efficiently.

* *L. John is supported by the National Science Foundation under Grants CCR-9796098 (CAREER Award), and EIA-9807112, and a grant from the Texas Advanced Technology Program. B. Evans was supported by the US National Science Foundation CAREER Award under grant MIP-9702707, the US Defense Advanced Research Projects Agency under contract DAAB07-97-C-J007, Accelerix, and Rockwell.*

1 Introduction

Demand for digital signal processing (DSP) and multimedia capabilities on a personal computer has been increasing to accommodate 3-D graphics, video conferencing, and other applications. The PC industry's attempt to satisfy these demands resulted in the first addition to the X86 instruction set architecture (ISA) in almost a decade. This extension, introduced in 1996, has been dubbed MMX (for MultiMedia eXtension) [1, 2] and can outperform lower-end DSP processors [3]. This technology adds new assembly instructions and data types to the existing ISA to exploit the data parallelism that is often available in DSP and multimedia applications. In this study, we investigate the performance of a suite of programs on Intel Pentium processors with MMX technology.

MMX and native signal processing (NSP) extensions to general purpose processors [4, 5] are single-instruction multiple-data (SIMD) architectures. One SIMD data type which contains several pieces of data is sent to a processing unit. By packing many pieces of data into one 64-bit MMX register, several calculations can take place simultaneously [6]. For example, image processing applications typically manipulate matrices of 8-bit data. Eight pieces of this data could be packed into an MMX register, arithmetic or logical operations could be performed on the pixels in parallel, and the results could be written to a register.

To achieve this functionality, MMX technology adds 57 new assembly instructions to the X86 instruction set. These instructions can operate on any of the packed data types and on unsigned or signed data. Saturation and wrap-around arithmetic are also supported. Multiply-accumulate (MAC), a frequent operation in DSP applications, is also part of the ISA. MMX can multiply 8-bit and 16-bit fixed-point data, but not 32-bit data. Data widths of 8 and 16 bits are sufficient for speech, image, audio, and video processing applications as well as 3-D graphics.

MMX maintains full compatibility with existing X86 operating systems and applications. MMX registers and state are aliased onto the floating-point registers and state, so no new registers or states are introduced by MMX. Maintaining compatibility places limitations on MMX. The MMX registers are limited to the width of the floating-point registers (MMX uses 64 of the 80 available bits) and mixing of floating point and MMX code becomes costly.

In DSP applications, several algorithms surface more frequently than others [3, 4, 7, 8, 9, 10, 11, 12]. The most common DSP kernels are the finite impulse response (FIR) filter, infinite impulse response (IIR) filter, fast Fourier transform (FFT), least mean square (LMS) adaptive filters, and matrix-vector arithmetic. Applications commonly benchmarked are speech, audio, image and video compression systems.

Previous efforts have analyzed NSP on general purpose processors [5, 13]. However, efforts to compare applications with MMX instructions versus applications without MMX on the same X86 processor have been incomplete [8]. The results found in [8] are only anticipated results based on simulation. A benchmarking of several applications on the UltraSPARC processor [4] using the Visual Instruction Set (VIS) showed a performance speedup for some DSP applications over non-VIS versions. Applications with FIR filters showed the most improvement while IIR filters and FFTs exhibited little or no performance increase [4].

To study the effects of DSP and multimedia programs, we need code with and without MMX for these applications. It is important to note that there are no publicly available compilers that support MMX instructions. This means that the burden of incorporating MMX is placed solely on the developers of the application. Achieving the largest performance increase would involve tailoring MMX assembly code for each specific application or kernel and then in-lining this assembly code into the application source code. A less time-consuming method would be to write generic MMX libraries for common algorithms and kernels which can be accessed via function calls.

Intel provides a suite of optimized assembly libraries on their Web site [14] and with VTune [15]. Recent versions, which include the Signal Processing Library 4.0, Recognition Primitives Library 3.1, and Image Processing Library 2.0, support fixed-point functions that utilize MMX and floating-point functions. Not all DSP algorithms have corresponding MMX functions (e.g. the LMS algorithm). We developed some C code and acquired the remainder from several sources [10, 16, 17, 18]. We looked for code that is fast and efficient, yet somewhat modular and easy to interpret so that we could interface it with the Intel libraries.

Although MMX presents opportunity for performance increase, to our knowledge no independent evaluations of applications on an X86 processor with MMX corroborate and explain the performance increase. We quantify speedup for our benchmark suite of kernels and applications and offer insight into development of DSP and multimedia applications on the Pentium family of processors. We analyze variations in the execution time, dynamic code size, static code size, number of memory references, function calls, number of MMX instructions, and mix of MMX instructions. Speedup is achieved on some, but not all, DSP and multimedia applications. The effects of packing and unpacking do not undermine the advantages of MMX, and some applications do not require packing and unpacking of data because of properly aligned data. On applications such as JPEG image compression that ran slower with MMX, we found that the core kernels of the applications show speedup, but numerous calls to NSP assembly libraries as well as the formatting of data for these libraries prove to be significant overhead. The most effective MMX applications require buffered data and high data parallelism. The instruction mix of the applications allows us to provide more insight into these specific observations and elucidate other concerns.

In this paper, Section 2 describes the benchmark programs. Section 3 describes the experiment methodology, and Section 4 analyzes the results. Section 5 concludes the paper.

2 Benchmarks

For our study, we profile four DSP and multimedia kernels and four applications. Table 1 summarizes the implementations and general characteristics of the four kernels and four applications that comprise our MMX benchmark suite. We provide more information about our benchmark source code at the following Web site: `http://www.ece.utexas.edu/~ljohn/mmxdsp/`.
The rest of this section provides some details on the benchmarks.

2.1 Kernels

Finite Impulse Response (FIR) Filters allow certain frequency components of the input to pass unchanged to the output while blocking other components. FIR filters are moving average filters. Their response to an impulse dies away in a finite number of samples. The output $y(n)$ is a weighted average of the

Table 1: Summary of Benchmark Kernels and Applications

Kernels	
Fast Fourier Transform (`fft`)	4096 point, in-place FFT
Finite Impulse Response Filter (`fir`)	Low-pass filter of length 35 (i.e. 35 coefficients and 35 entry history).
Infinite Impulse Response Filter (`iir`)	Butterworth, direct form, eighth-order bandpass filter. Filter length of eight with 17 coefficients.
Matrix and Vector Arithmetic (`matvec`)	Matrix-vector multiplication of a 512×512 matrix with a vector of length 512. Dot product on two vectors of length 512.
Applications	
JPEG Image compression (`jpeg`)	Compresses an image into JPEG format. Converted an 118 kB Windows bitmap image into a JPEG image. Primary kernels include vector arithmetic for imaging and the discrete cosine transform (DCT) kernel.
Image Manipulation (`image`)	Dimming and switching the colors of a Windows bitmap. 480×640 Red-Green-Blue (RGB) image in which each pixel is represented by 24 bits. Essentially vector addition and multiplication.
G.722 Speech Encoding (`g722`)	Standard for digital encoding and compression of speech and audio signals. Uses adaptive differential pulse code modulation (ADPCM). Encoded a 6 kB speech file.
Doppler Radar Processing (`radar`)	Subtracts successive complex echo signals to remove stationary targets from a radar signal and estimates the power spectrum of the resulting samples. The dominant frequency is then estimated using the peak of the FFT spectrum. The FFT is a 16-point, in-place, radix-2, decimation-in-time FFT.

input values $x(n)$.

$$y(n) = \sum_{k=0}^{M-1} c_k \; x(n-k). \tag{1}$$

On each invocation, the (`fir`) filter takes one new input value and returns one new output value. The non-MMX versions perform 32-bit floating-point arithmetic, while the MMX version performs 16-bit fixed-point calculations. Applications of an FIR filter include speech processing, audio processing modem channel equalization, linear predictive coding, and general filtering.

Infinite Impulse Response (IIR) Filters are also frequency selective and include autoregressive filters. IIR filters feed back a weighted sum of previous output values $y(n)$ and add this to a weighted sum of previous and current input values $x(n)$

$$y(n) = \sum_{q=0}^{Q-1} b_q x(n-q) - \sum_{p=0}^{P-1} a_p y(n-p). \tag{2}$$

A given order IIR filter can be made more frequency selective than the same order FIR filter, making them more computationally efficient for the same behavior. Unlike the FIR filter, our IIR filter (`iir`) performs block filtering, passing eight samples to the IIR filter function per invocation. The non-MMX versions perform 64-bit floating-point arithmetic, while the MMX version performs 16-bit fixed-point calculations. Applications include audio equalization, speech compression, linear predictive coding and general filtering.

Fast Fourier Transform (FFT) is an efficient algorithm for computing the discrete Fourier transform (DFT) of a sequence. The DFT is represented as follows:

$$X(k) = \sum_{n=0}^{N-1} x(n) \; e^{-j2\pi kn/N} \tag{3}$$

Letting $W^{nk} = e^{-j2\pi kn/N}$, (3) can be simplified to

$$X(k) = X_{ev}(n) + {W^k}_{N/2} X_{od}(n), \tag{4}$$

where X_{ev} represents the even-indexed elements and X_{od} represents the odd-indexed elements. The DFT can be divided into even and odd halves repeatedly until only two-point DFTs remain to evaluate.

Our implementation of the FFT (`fft`) supplies all of the data at once to the functions that compute it. The non-MMX version of the FFT perform 32-bit, floating-point calculations. The MMX version uses 16-bit fixed-point data. FFT-based applications include radar processing, sonar processing, MPEG audio compression, ADSL modems, and spectral analysis.

Matrix-Vector Arithmetic includes dot products and matrix-vector multiplications (`matvec`). All versions use 16-bit fixed-point data. Any numeric computation with some degree of data parallelism can take advantage of vector arithmetic, especially image applications.

The DSP kernels benchmarked in our suite contain a C-only code version, a version that uses Intel's MMX assembly library functions, and, if applicable, a version that uses Intel's floating-point assembly library functions. It is regular practice to optimize common DSP

kernels in assembly code, so we compare the MMX version to both a compiler-optimized version and a hand-optimized version. There is no hand-optimized floating-point version for the vector arithmetic (`matvec`) because it uses only integer data.

2.2 Applications

Doppler Radar Processing (`radar`) subtracts successive complex echo signals to remove stationary targets from a radar signal and then estimates the power spectrum of the resulting samples. The mean frequency is then estimated by finding the peak of the FFT spectrum. The input is complex and represents 12 range locations from each echo [16]. The MMX version of this application uses the vector arithmetic kernels and FFT kernel. There is little measured change in the output precision (10^{-6}) between the MMX and non-MMX versions.

JPEG is a standardized compression method for full-color and gray-scale images. We use JPEG to compress images with loss of information — the output image is not necessarily identical to the input image. For natural images, medium compression ratios may produce no visible change, and high compression ratios will produce low-quality images which may be tolerable. Our JPEG benchmark program (`jpeg`) [17] compresses but not decompresses images. It performs color conversion, two-dimensional forward discrete cosine transform (DCT), and quantization of the DCT coefficients. We use it to reduce an 118 kB Windows bitmap file into a 7 kB JPEG file. The MMX version shows no visible difference in quality than the non-MMX version, although some precision is lost in the pixel calculations. We validated the JPEG results produced by both the MMX and non-MMX versions of the JPEG encoder by using the *Imaging for Windows NT* program.

G.722 Speech Encoding (`g722`) is a standard for compressing and decompressing speech using adaptive differential pulse code modulation (ADPCM). The input signal to the encoder is 16-bit data sampled at 16 kHz. Output from the encoder is 8 bits at an 8 kHz sample rate. The decoder operates in exactly the opposite fashion. An 8-bit coded input signal is decoded by the ADPCM decoders. The result is a 16 kHz sampled output [16] [19]. We encoded a 6 kB speech file. Both versions of this application perform real-time encoding and decoding. Only one sample of speech is encoded and decoded at a time. The quality of speech in the MMX version is tolerable, but slightly inferior to that of the C-only version.

Imaging program (`image`) manipulates the pixels in a 640×480 bitmap image uniformly. First, the program takes an image stored as a long array of 8-bit values and properly scales the values to produce a dimming effect. This primarily consists of vector multiply operations. Second, the program increases or decreases the value of certain pixels to produce a switch in colors. This function primarily involves vector addition. This application shows no loss of quality between the MMX and C-only versions when viewed using the Intel image library viewing routine.

All of the non-MMX versions of the applications generously use 32-bit fixed-point, 32-bit floating-point, and 64-bit floating-point numbers to perform arithmetic. This strategy is safe, increases precision, and is not very costly on a general-purpose processor. Since efficient use of MMX requires either 16-bit or 8-bit data, MMX versions reduce some data to 16 or 8 bit values where appropriate. The applications run without significant loss of precision. For larger applications, we use VTune to profile the C version of each application, so that we can optimize the most frequently called functions with MMX assembly functions if possible. For each application, the functions that are optimized account for 65-75% of the samples during the non-MMX application's runtime.

In the process of creating our benchmark with and without MMX, we made efforts to ensure we were comparing equivalent sections of code. In some programs, data is obtained from a file or written to a file. In these cases, we buffer the data and monitor only the reading from the buffer and not the I/O. We do not monitor the initialization, setup routines, operating system work, or file I/O for any of the programs. We strive only to monitor the core of the kernels and applications and measure the effects of MMX performing useful DSP and multimedia work.

3 Methodology

We compile the benchmarks using Microsoft Visual C++ 5.0 on a Pentium II processor running Windows NT 4.0. All programs are compiled with the optimization "Maximize for Speed." Intel provides a separate version of their libraries for different Pentium versions. For MMX, we choose the Pentium II MMX version since it is their most recent, and presumably most efficient, version. We obtain performance data for the benchmarks by using the dynamic analysis utility in VTune 2.5.1 [15].

3.1 Using Intel's Assembly Libraries

We create the benchmarks by taking efficient, reliable, C programs and then modifying them to use

assembly libraries. Intel's assembly libraries [14] provide versions of many common signal processing, vector arithmetic, and image processing kernels. While the Intel library functions are generally robust, intuitive, and functionally correct, limitations exist in both the libraries and existing C code.

First, the C programs often use floating-point and 32-bit fixed-point numbers, but the Intel MMX libraries do not provide functions that accept 32-bit fixed-point inputs. We rarely can simply quantize parameters (e.g., filter coefficients) to 16-bit or 8-bit fixed-point formats and achieve acceptable results. The library functions require that the output data be the same length as the input data. A "scale factor" is provided to handle overflow, but scaling often results in a loss of precision. In their implementation, this scale factor must be known a priori to use MMX and therefore must allow for the largest possible overflow.

Second, C code can present obstacles to efficiently using MMX. In order to exploit the data parallelism in C loops that access vector elements sequentially, all of the temporary variables in the loop must be placed in vectors as well. Significant overhead may be associated with this extra allocation of memory, especially if allocated dynamically. Similarly, some signal processing library calls require library-specific data structures to be created and initialized before calling kernels such as FIR and IIR filters.

Third, because the same registers and state are used for both floating-point and MMX code, potential overhead exists when switching between modes. The `emms` (Empty MMX State) instruction that switches from MMX to floating-point mode can incur up to a 50-cycle penalty [18].

3.2 Using VTune

VTune is a system performance profiling tool created by Intel [15]. The dynamic analysis utility provides a way to monitor which assembly instructions in a given application are being executed. VTune takes this stream of instructions and uses them to simulate timing and performance information.

VTune is capable of simulating timing using the Pentium superscalar, two-pipeline, in-order architecture. It traces the execution of the entire application. In order to isolate code, the user may specify starting and stopping execution points so that only results for that section of the program are reported. When computing cache, branching, and other penalties, however, the entire application is considered. Clock cycles are calculated from the known latency of each assembly instruction and known latency of each penalty on the Pentium, e.g., cache misses and branch target buffer misses. The most recent version available while performing this research (VTune 2.5.1, May 1998) does not yet have the full capabilities to simulate the dynamic execution micro-core of the Pentium Pro and Pentium II processors.

3.3 Metrics

Execution time and the resulting speedup are the primary metrics we use to evaluate MMX. We also analyze the number of times individual assembly instructions, including MMX instructions, are executed (instruction mix), number of instructions executed during actual runtime (dynamic instructions), number of unique instructions executed (static instructions), number of micro-operations that are dynamically executed, and architecture related penalties (cache misses, branch target buffer misses, and other CPU-related penalties).

Not all of the obtained statistics are accurate for every processor in the Pentium family. Instruction mix, dynamic instructions, and static instructions are valid for any Pentium family processor with MMX. Micro-operations only apply to the Pentium Pro with MMX and Pentium II processors. Clock cycles and architecture related penalties are unique to the Pentium with MMX processor.

Table 2 provides the basic characteristics of the benchmarks as measured by VTune. Included are the number of dynamic X86 instructions, the number of static X86 instructions that produced the dynamic instructions, the number of dynamic micro-operations or μ-ops (Pentium Pro and Pentium II), the percentage of X86 instructions that use any memory referencing mode, and the percentage of X86 instructions that are MMX instructions.

Figure 1(a) shows the percentage of MMX instructions within the MMX version of each program. The programs are arranged in ascending order of speedup. (The speedup obtained by each program is shown above each bar.) In addition, the MMX instructions are broken down into their individual categories: packing and unpacking, MMX arithmetic, MMX moves (64-bit), and the `emms` instruction. Figure 1(b) presents information on static and dynamic instruction count as the ratio of C-only version to MMX version.

4 Analysis of Results

Table 3 presents some of the results of the study as ratios of non-MMX version totals to MMX version totals. Figure 2(a) presents C-only to MMX ratios for execution time (speedup), dynamic instructions, and

Table 2: Benchmark Instruction Characteristics

Benchmark Program	Static Instructions	Dynamic μ-ops	Dynamic Instructions	% Memory References	% MMX Instructions
fft.c	110	8,429,851	5,619,929	53.64	
fft.fp	1,446	3,285,827	2.389.118	54.61	
fft.mmx	1,640	2,585,564	1,842,347	49.54	4.69
fir.c	32	2,580,000	2,112,000	40.62	
fir.fp	78	2,922,288	2,190,000	42.46	
fir.mmx	218	2,040,889	1,332,051	31.98	20.27
iir.c	60	2,924,802	2,678,258	22.37	
iir.fp	223	1,652,784	1,325,964	37.16	
iir.mmx	227	1,299,588	1,010,568	28.33	71.23
matvec.c	35	2,106,409	2,105,355	25.04	
matvec.mmx	159	1,085,055	395,125	45.83	91.6
radar.c	389	12,953,062	10,110,365	47.04	
radar.mmx	1,105	11,193,249	7,190,019	36.36	8.64
g722.c	1,281	16,258,744	11,618,849	59.92	
g722.mmx	1,752	25,898,326	17,582,880	43.44	1.58
jpeg.c	3,755	12,901,353	9,700,077	43.25	
jpeg.mmx	4,434	25,343,001	16,294,772	44.29	6.52
image.c	68	37,934,090	26,870,550	27.47	
image.mmx	175	5,063,817	2,707,314	38.29	85.10

Static Instructions are the static instructions that are executed. *Dynamic μ-ops* are dynamic micro-operations that occur on a Pentium II. *Dynamic Instructions* get executed during the running of the program. *Memory References* are assembly instructions that use any memory referencing mode. `.c` denotes the C-only code version of the program. `.fp` denotes the C version with calls to the floating-point library. `.mmx` denotes C version with calls to the MMX library.

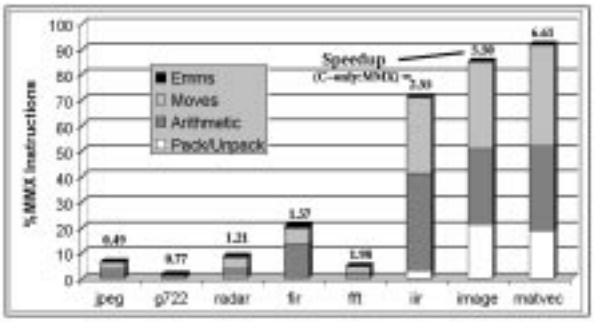
memory references. Figure 2(b) does the same for the C with floating point library version to C with MMX ratio. In these figures, the reduction of memory references and dynamic instructions in the MMX versions correspond closely with the decrease in execution time. The reduction in memory references is important to notice because of the significant cost of accessing off-chip memory. We will further discuss these results and each application's performance in the forthcoming paragraphs.

4.1 Results from Kernels

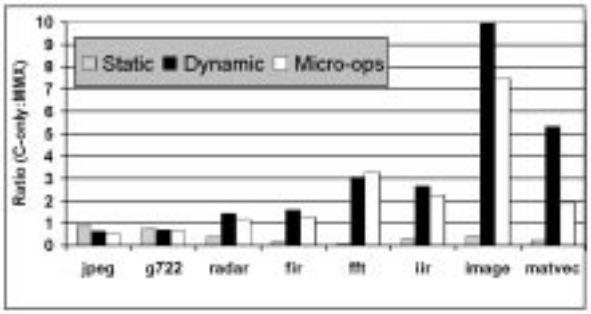
As shown in Table 3, the kernels show reasonable speedup with MMX code. The MMX speedup relative to the optimized floating-point library (`.fp`) versions is less than the speedup relative to the C-only (`.c`) code versions in all cases. Additional speedup is achieved using MMX instead of hand-optimized floating-point assembly code. In all cases, the static instruction size is increased when using MMX, even versus floating-point assembly code. Although speedup is achieved and dynamic instructions are reduced significantly, the static code size increases when using MMX. This is the result of the combination of software optimization techniques (such as loop unrolling), MMX packing and unpacking of SIMD types, calling and returning from function calls, and handling precision problems (i.e. scaling).

The FIR filter kernel shows a decent speedup factor of 1.57 against C-only code and a factor of 1.34 versus the floating-point library. Although this kernel loses some data parallelism because it only processes one input at a time, the relatively high number of taps allows many parallel operations to take place while calculating the summation (20% of instructions are MMX instructions). The MMX version reports zero packing and unpacking instructions as a result of properly aligned stores and moves. The MMX version of this kernel spends 11% of the reported clock cycles doing `call` and `ret` and twice as many total clock cycles as the C-only version. Finally, the FIR filter suffers little loss of precision in the MMX fixed-point version (order 10^{-4}) because the error loss is not cumulative at any point.

The IIR filter kernel shows a more impressive speedup factor of 2.55 compared to the C-only code and 1.71 compared to the floating-point library routine. The static code size increases in the MMX version, but not as significantly as `fir`. The higher speedup results from block processing of the input samples (71% MMX



(a) Breakdown of MMX instructions.



(b) C-only vs. MMX implementations.

Figure 1: Instruction mix analysis of MMX code and instruction count comparison between MMX and non-MMX versions of code. For the benchmarks given, speedup increases from left to right.

instructions) which increases the data parallelism and reduces the number of functions called. The output of the MMX version of this IIR filter becomes unstable. Although the non-MMX versions produce reasonable results, the loss of precision in the MMX version compounds iteration after iteration and the output overflows the 16-bit data type. In a previous study [20], our implementation of an IIR filter yielded better precision but that version achieved a slowdown rather than a speedup.

The FFT kernel has speedup of 1.98 compared with the C-only version, but only 1.25 compared with the floating-point library version. Factors other than MMX are speeding up this kernel. One factor is the large difference in the number of data memory references. Accessing memory can be very expensive (three cycles for a data cache miss, 8 cycles for an L2 access, and 15 cycles for an L2 miss). Also, the MMX version of the FFT only uses 4.6% MMX instructions, which is the fewest of all the kernels. Although the implementation of the FFT is not outlined in the Intel documentation, the assembly code indicates that the samples are converted to floating-point, and then the FFT is computed in a similar manner to the floating-point library version. In the initial stages of our study, we used an earlier version of the Pentium MMX library. In this case, the FFT used 40% MMX instructions, but provided only 1.49 speedup over a C-only version. This suggests that computing the FFT with MMX integer calculations is not an efficient strategy. The limited use of MMX does provide a speedup over the floating-point version with little loss of precision (order 10^{-2}) using the 16-bit data in our implementation.

The matrix-vector kernel is predictably well-suited for an MMX implementation. Approximately 90% of all `matvec`'s instructions are MMX instructions. The execution time due to MMX is reduced by a factor of 6.61 and the dynamic instructions are reduced by a factor of 5.3. Note that this kernel operates on 16-bit data, so four pieces of data can be operated on in parallel, yet the speedup is much greater than 4.0. The superlinear speedup is largely due to the `imul` instruction which does integer multiplication in 10 cycles versus the `pmaddwd` MMX instruction which can perform two multiplications in 3 cycles. The dynamic instruction size reduction is due in large part to more efficient management of the loop structure in the MMX code. It may be observed that `matvec` has the largest percentage of packing and unpacking MMX instructions (20.5%), yet has a significant speedup.

4.2 Application Results

Overall, the results of the applications are disappointing. Two of the four applications (JPEG compression and G.722 speech encoding) did not yield any speedup. The image application is clearly the best suited for MMX. It shows speedup of 5.5 and a dynamic instruction reduction of 5.3. The most important factor is that the use of 8-bit data allows twice the parallelism compared to the use of 16-bit data. Also, the images are stored in a large array of 8-bit data and are properly aligned on 8-byte boundaries. This allows "automatic" packing and unpacking of data by simply loading and storing quad-words (64 bits) from memory. Also contributing to the speedup are a 9.92 times reduction in dynamic instructions, a 7.12 times reduction in data memory references, and 85% usage of MMX instructions.

The radar application has a low factor of speedup

Table 3: Results as ratios of Non-MMX program to MMX program

Benchmark Program	Speedup	Static Instructions	Dynamic Instructions	Micro-ops	Memory References
fft.c	1.98	0.067	3.05	3.26	3.30
fft.fp	1.25	0.881	1.29	1.27	1.42
fir.c	1.57	0.146	1.58	1.26	2.01
fir.fp	1.34	0.357	1.64	1.43	2.18
iir.c	2.55	0.264	2.65	2.25	2.09
iir.fp	1.71	0.982	1.31	1.27	1.72
matvec.c	6.61	0.220	5.32	1.94	2.91
g722.c	0.77	0.731	0.66	0.62	0.91
image.c	5.50	0.388	9.92	7.49	7.12
jpeg.c	0.49	0.847	0.62	0.51	0.61
radar.c	1.21	0.352	1.40	1.15	1.81

Speedup is the ratio of clock cycles. *Static instructions* are the static instruction that are executed. *Dynamic instructions* get executed during the running of the program. *Micro-ops* are dynamic micro-operations that occur on a Pentium II. *Memory References* are assembly instructions that use any memory referencing mode. `.c` in the Benchmark Program column denotes the C-only version of the program and `.fp` denotes the C version with calls to the floating-point library.

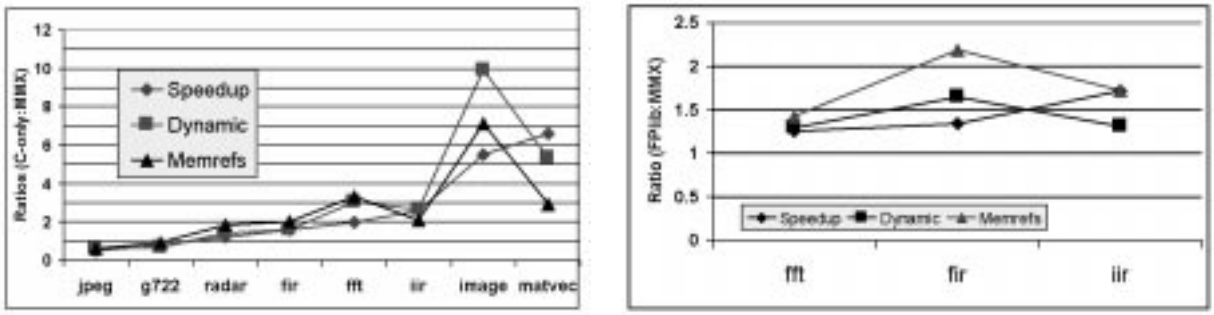
even though all of the arithmetic is accomplished using MMX vector and FFT routines. The execution time speedup is 1.21 and the dynamic instructions are reduced by 1.40. Although several MMX routines are called, only 9.58% of the instructions are MMX instructions. One shortcoming of the MMX version is that 27 times more function calls are made, many of which are unseen to the user because they are called within the libraries themselves. The `ret` and `call` functions themselves consume 23.88% of the total cycles without including the penalty for passing parameters.

We expect speedup for the JPEG compression program, but the final results show the C-only version to be 1.92 times faster although 5% of `jpeg` uses MMX instructions. The slowdown is the result of forcing the C application to use MMX library functions. The JPEG program's computation is dominated by two-dimensional DCT calculations, quantization, and color conversion. These three functions account for 74% of the C-only version clock cycles. We inserted MMX versions for these functions. The core operations of these three functions which contain 24% MMX instructions actually show a 1.6 speedup.

The JPEG program operates only on 8×8 blocks of image pixels. The MMX version would benefit from a larger block size to minimize function calls. In addition, there is no two-dimensional DCT call in the MMX libraries. So, instead of one call to a MMX 2-D DCT function, there are 16 calls to a one-dimensional DCT function. A highly optimized 2-D DCT written using MMX assembly code is found to have a speedup of 1.7 compared to a C implementation [21] while our MMX computation of the 2-D DCT shows only 1.1 speedup. Function calling accounts for 8.3 times more clock cycles in the Intel MMX version than the C-only version. In general, accessing the pixels in the JPEG program is rarely performed in sequential order. This increases the difficulty of performing MMX operations at either the function call level or the assembly level because efficient MMX coding requires data that is contiguous. Another problem that exists is the precision handling and type conversion that accompanies using the libraries in this program. The disruption that the above factors cause outweighs the benefit of using MMX in our implementation.

The G.722 speech compression and encoding actually shows a slow-down as well when implemented with MMX function calls. Some of the same problems from the JPEG application apply here as well. The primary drawback of this particular program as an MMX program is that it only processes one input at a time while encoding and decoding. Operating on blocks of data at once would definitely increase the opportunity to use MMX code. The MMX version uses 1.5% MMX instructions and spends 7.7% of its clock cycles on function call overhead and 3.3 times more cycles than the C-only program. In addition, the quality of speech deteriorates even with a low MMX instruction count.

The disappointing results are due, in large part, to our strategy of editing existing code instead of coding the applications from scratch. The programs we obtained are all highly optimized, especially the JPEG compression program, and ran very fast with high compiler optimizations. Editing the optimized core loops



(a) Ratio of C-only program to C program with MMX library calls.

(b) Ratio of C program with optimized floating-point calls to C program with MMX library calls.

Figure 2: Comparison of speedup, dynamic instruction count, and memory references.

and trying to convert them to MMX often disrupts the continuity of the code with function calls and initializations.

5 Conclusion

We analyzed the usage of MMX enhanced libraries in implementing DSP and multimedia programs. We found the various changes in execution time that occur when our benchmarks are run on the Pentium with MMX. We used several parameters to evaluate native signal processing performance enhancement, including execution time, dynamic instruction size, instruction mix, and number of data references. We observed that:

- MMX can provide significant speedup in certain DSP and multimedia applications, even over hand-optimized floating-point assembly code. The speedups range from 1.25 to 6.6 for the kernels and 0.49 to 5.5 for the applications.
- Effectively using MMX to produce speedup is a difficult chore for large or complicated C applications, even with assembly libraries. It can require significant restructuring of existing code, recoding with libraries in mind, or using exclusively assembly code.
- Function libraries are a viable option for obtaining speedup; however, the best performance increase will always be obtained by tailoring MMX assembly code to fit the application and refraining from hierarchical function calling.
- Although `jpeg` and `g722` experienced speedup in their core kernels, they did not experience speedup as a whole application. Potential overhead and other efficiency issues which can deteriorate performance arise when using flexible, robust library functions. These issues include factors such as excessive function calls, preformatting the data to fit the library format, and switching between floating-point and MMX code.
- MMX has the potential to reduce dynamic instructions and micro-operations, but can dramatically increase the static code size of the applications.
- Although packing and unpacking of data generates overhead, programs with high percentage of these instructions are shown to be faster than their non-MMX counterparts due to high utilization of SIMD parallelism.
- MMX seems well-suited for imaging applications because they often have plenty of contiguous, 8-bit data available and rarely require precision beyond 8 bits. Higher precision signal processing applications seem to cause large problems due to their real-time and high-precision requirements.
- Although the Intel MMX library is useful, intuitive, and time-saving, providing 32-bit outputs for 16-bit integer operations would enable preservation of precision across function calls. The interleaving of high and low words during multiplication is a significant problem. Image and video compression programs would benefit from a two-dimensional DCT function in the MMX library.
- Reducing memory references is just as important as reducing the number of arithmetic operations

because accessing off-chip cache can be very expensive on a general purpose processor.

Based on our observations, MMX assembly code interspersed in C code might not be the best use of the MMX instructions set. Benchmarks that can truly exploit MMX will require more restructuring of C code and hand-coding some functions not available in the Intel assembly libraries, such as the 2-D DCT. Future work on the subject could include these modifications in addition to more benchmarks, such as an MPEG video codec or commercial applications.

References

[1] A. Peleg and U. Weiser, "The MMX Technology Extension to the Intel Architecture," *IEEE Micro*, vol. 16, no. 4, pp. 42-50, Aug. 1996.

[2] S. Wilkie and U. Weiser, "MMX Technology for Personal Computers," *Communications of the ACM*, vol. 40, pp. 24–38, Jan. 1997.

[3] G. Blalock, "Microprocessors Outperform DSPs 2:1," *MicroProcesor Report*, vol. 10, no. 17, pp. 1-4, Dec. 1995.

[4] W. Chen, H. J. Reekie, S. Bhave, and E. A. Lee, "Native Signal Processing on the UltraSparc in the Ptolemy Environment," *Proc. IEEE Asilomar Conference on Signals, Systems, and Computers*, pp. 1368-1372, Nov. 1996.

[5] R. B. Lee, "Accelerating Multimedia with Enhanced Mircoprocessors," *IEEE Micro*, vol. 15, no. 2, pp. 23-32, Apr. 1995.

[6] D. Bistry, C. Dulong, M. Gutman, M. Julier, and M. Keith, *The Complete Guide to MMX Technology*. McGraw-Hill, 1997.

[7] G. Blalock, "The BDTIMark: A Measure of DSP Execution Speed"," 1997. White Paper from Berkeley Design Technology, Inc. http://www.bdti.com/articles/wtpaper.htm.

[8] L. Gwennap, "Intel's MMX Speeds Multimedia," *MicroProcesor Report*, vol. 10, no. 3, p. 1, 1996.

[9] P. Lapsley and G. Blalock, "Evaluating DSP Processor Performance," 1996. White Paper from Berkeley Design Technology, Inc. http://www.bdti.com/articles/wpeval.htm.

[10] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *IEEE Micro*, vol. 30, no. 1, pp. 330-335, Dec. 1997.

[11] M. A. Saghir, P. Chow, and C. G. Lee, "Exploiting Dual Data Memory Banks in Digital Signal Processors.," *Proc. Conf. Architectural Support for Prog. Lang. and Operating Sys.*, pp. 234-243, Oct. 1996.

[12] V. Zivojnovic, H. Schraut, M. Willems, and R. Schoenen, "DSP's, GPP's, and Multimedia Applications - an Evaluation of DSPstone," *Proc. Int. Conf. on Signal Proc. Appl. and Tech.*, pp. 1779-1783, Oct. 1995.

[13] R. B. Lee, "Multimedia Extensions For General-Purpose Processors," *Proc. IEEE Workshop on Signal Processing Systems*, pp. 9-23, Nov. 1997.

[14] Intel, "Performance Library Suite." *http://developer.intel.com/design/perftool/perflibst/.*

[15] Intel, "VTune CD." *http://developer.intel.com/design/perftool/vtcd/.*

[16] P. M. Embree, *C Algorithms for Real-Time DSP*. NJ: Prentice Hall, 1995.

[17] Independent JPEG Group. *http://www.ijg.org/.*

[18] Intel, "Developers' Insight." *http://developer.intel.com/drg/mmx/manuals/overview/.*

[19] F. G. Stremler, *Introduction to Communication Systems*. Reading, MA: Addison-Wesley Publishing Company, 3rd ed., 1990.

[20] R. Bhargava, R. Radhakrishnan, B. L. Evans, and L. K. John, "Characterization of MMX-enhanced DSP and Multimedia Applications on a General Purpose Processor," *Digest of the Workshop on Performance Analysis and Its Impact on Design held in conjunction with ISCA98)*, pp. 16–23, June. 1998. *http://www.ece.utexas.edu/~ljohn/mmxdsp/.*

[21] B. Erol, F. Kossentini, and H. Alnuweiri, "Implementation of a Fast H.263+ Encoder/Decoder," *Proc. IEEE Asilomar Conf. on Signals, Systems, and Computers,* (Pacific Grove, CA), Nov. 1998.