Characterization of MMX-enhanced DSP and Multimedia Applications on a General Purpose Processor

R. Bhargava, R. Radhakrishnan, B. L. Evans and L. John *
Electrical and Computer Engineering Department
University of Texas at Austin
Austin, TX 78712
(512) 233-1455
{ravib, radhakri, bevans, ljohn}@ece.utexas.edu

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

Abstract

Proper use of native signal processing (NSP) instruction set enhancements can result in speedup for targeted applications. In this paper, we study the behavior of the X86 architecture's Multimedia Extension (MMX) instruction set on signal processing and multimedia algorithms and applications. In addition to quantifying speedup, we make comparisons based on detailed dynamic instruction profiling. We compare a suite of digital signal processing (DSP) and multimedia programs implemented in C code and the same programs implemented with calls to an MMX assembly library that performs filtering, vector arithmetic, and other relevant kernels. As expected, our analysis shows decreased execution time for most, but not all, of our MMX programs compared to their unmodified equivalents. The observed speedup for the programs using MMX ranges from 1.2 to 7.5. For each set of programs, we perform a detailed instruction level analysis using VTune. This allows us to isolate the specific reasons for speedup or lack thereof. This analysis allows one to understand which aspects of native signal processing are most useful and how it can be utilized most efficiently.

*L. John is supported in part by grants from the National Science Foundation and the Texas Advanced Technology Program. B.L. Evans is supported on a US National Science Foundation CAREER Award under Grant MIP-9702707.

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) and can outperform lower-end DSP processors [1]. 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 DSP and multimedia programs on an Intel Pentium processor with MMX technology.

MMX and implementations of native signal processing (NSP) on other general purpose processors [2] [3] belong to the single-instruction multiple-data (SIMD) class of machines. One SIMD data type is sent to an arithmetic logic unit but it actually contains several pieces of data. These pieces of data are then operated on in parallel in the arithmetic unit. 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

added to the instruction set as well. The MMX multiply and MAC can do multiplication of 8-bit and 16-bit fixed-point data. These data widths are sufficient for speech, image, low-end audio, and video processing applications as well as 3-D graphics.

The Pentium processor achieves superscalar performance by utilizing two pipelines called the U pipeline and the V pipeline [4]. MMX technology allows two MMX instructions to be executed per instruction. From a programmer's point of view, MMX technology maintains full compatibility with existing operating systems and applications by aliasing the MMX registers and state onto the floating-point registers and state. Therefore, floating-point and fixed-point operations cannot be mixed without a performance penalty. The Empty MMX State (`EMMS`) instruction must be called before switching from MMX assembly code to floating-point assembly code and may take as many as 50 cycles [5] [6].

When discussing the implementation of many traditional DSP applications, several algorithms surface more frequently than others [1] [7] [2] [6] [8] [9] [10] [11]. 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, matrix-vector arithmetic, and variations on these. Applications most often benchmarked are speech, audio, image and video compression systems.

Previous efforts have analyzed NSP on general purpose processors [12] [3]. However, efforts to compare applications with MMX instructions versus applications without MMX on the same X86 processor have been incomplete [6]. The results found in [6] are only anticipated results based on simulation. A benchmarking of several applications on the UltraSparc processor [2] using the Visual Instruction Set (VIS) showed a performance speedup for some DSP applications. Applications with FIR filters showed the most improvement while IIR filters and FFT exhibited little or no performance increase [2].

Although MMX presents the opportunity for performance increase, to our knowledge there have been no independent evaluations of applications on MMX PCs to corroborate and explain this. Our first objective is to observe speedup results for our benchmark of kernels and applications. Based on our observations, we offer insight into developing DSP and multimedia applications on the Pentium. Our observations include variations in the execution time, dynamic code size, number of memory references, function calls, number of MMX instructions, and mix of MMX instructions. From these observations we hope to answer several questions. How much speedup can one realistically expect to achieve? Does the parallel execution of data sufficiently make up for the packing and unpacking overhead of using SIMD instructions? Should one manually in-line MMX code or place it in libraries? What types of algorithms are worth the development effort to write MMX assembly code? The instruction mix of the applications allows us to provide insight into these specific questions and elucidate other concerns.

To study the effects of DSP and multimedia programs, we first needed to obtain source code for such programs and the equivalent MMX assembly code. 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. 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 performance libraries both on their Web site [5] and with VTune. Recent versions of the libraries (including the Signal Processing Library version 4.0) include functions that utilize MMX, although not all DSP algorithms have corresponding MMX functions (e.g. the least mean square (LMS) algorithm). We developed C code and acquired the remainder from several resources [13]. We were looking for code that is fast and efficient, yet somewhat

modular and easy to interpret so that we could interface them with the Intel libraries. These resources allowed us to develop reliable and efficient programs in a relatively short amount of time.

The paper is organized as follows. In Section 2, we describe the benchmark programs used in the study. Section 3 describes the experiment methodology and Section 4 presents the results. Section 5 offers a summary and concluding remarks.

2 Benchmark Programs

Table 1 summarizes the four DSP kernels and three applications that comprise our MMX benchmark suite. In the process of creating our two sets of benchmarks with and without MMX, we are required to make compromises when modifying the original code so that it could accept MMX library calls. The setup and initialization for the input and output data structures are the same for the original version of the C programs and the MMX-enhanced version. In some cases, the MMX data needs to be passed to the library functions in library-specific data structures. In some programs, data is obtained from a file or written to a file. Programs are altered to access data from buffers. We do not monitor data initialization routines or file I/O. Instead, we only monitor the core of the kernels and applications.

The FIR and IIR filters perform real-time filtering. In each invocation, each filter takes one new input and returns one new output. In the case of the FIR, IIR, and FFT kernels and the radar processing application, the unmodified C programs use 32-bit floating point values throughout. For the MMX versions of these programs, 16-bit fixed-point data is required. The coefficients and inputs are scaled and truncated appropriately to minimize the margin of error. The FIR, FFT, and radar programs showed a very small error at their outputs due to this conversion (order of 10^{-6}). The IIR filter, on the other hand, showed similar outputs for the first few passes, but soon became unstable due to the loss of precision.

The data parallelism in the FIR and IIR filters comes from the constant filter coefficients and the previously computed values (history) retained while calculating moving averages. In the FFT kernel, there is parallelism available in the buffered input array of known data values. In our image processing applications, all of the data is present at the beginning and the pixels have no history or relationship to their neighbors. More information on the digital signal processing algorithms used in our suite can be obtained from [13][14] .

3 Methodology

First, we produced a working version of the C program for each benchmarked kernel and application. Then, we wrote a program so that it could utilize the MMX function libraries and initialize data in a similar fashion. Next, we compiled the original C program and MMX version using Microsoft Visual C++ 5.0, with the highest level of optimization for maximum speed.

Once the programs were compiled, the outputs were compared to see if the results were similar and to validate that no significant errors were being made. At this point, we used VTune [15] to analyze programs and obtain dynamic instruction information. We parsed the VTune output files and collected the relevant statistical data.

4 Analysis of Results

Table 2 presents the dynamic instruction count of our benchmarks. Percentage of data access instructions and cycles spent in data access are also presented. Table 3 presents a comparison of the C code and the MMX version of each individual benchmark. Our primary interest is to find out specifically when MMX is performing well and when it is not.

The FFT kernel shows modest speedup (about 1.5 times reduction in cycles) and dynamic instructions are reduced by a factor of 2.35. The FFT uses the widest variety of MMX instructions including the multiply-accumulate

Kernels	
Finite Impulse Response Filter	Low-pass filter of length 35 (i.e. 35 coefficients and 35 entry history).
Infinite Impulse Response Filter	Direct form, second-order bandpass filter.
Fast Fourier Transform	1024 point, in-place, radix-2 decimation in-time FFT
Matrix and Vector Arithmetic	Matrix-vector multiplication of a 16x16 matrix with a vector of length 16. Dot product on two vectors of length 16.
Applications	
Doppler Radar Processing	Subtracts successive complex echo signals to remove stationary targets from a radar signal and estimates the power spectrum of the resulting samples. The main 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.
Image Dim (image1)	Reduce the intensity of a Windows bitmap. 480x640 RGB (Red Green Blue) image where each pixel is represented by 24 bits. Essentially vector multiplication.
Image Color Switch (image2)	Switch the colors of a Windows bitmap. 480x640 RGB image where each pixel is represented by 24 bits. Essentially vector multiplication.

Table 1: Summary of Benchmark Kernels and Applications

Benchmark programs	Non MMX code			MMX code		
	Dyn_inst	% Dat_refs	% Dat_ref cycles	Dyn_inst	% Dat_refs	% Dat_ref cycles
FFT	259,524	53.72	47.64	110,553	75.25	72.30
FIR	2,178,003	39.67	38.36	954,000	42.34	62.54
IIR	1,540	47.27	32.33	4676	60.48	31.34
MatVec	2,927	31.43	43.81	493	34.60	87.92
Radar	65,552	46.97	37.28	47,437	52.81	41.19
Image1	15,678,746	30.76	29.42	2,246,460	42.56	17.14
Image2	7,380,026	49.97	85.70	403,232	55.24	96.53

Table 2: Benchmark characteristics

Dyn_inst is the total number of instructions executed. *% Dat_refs* is the percentage of instructions in the program that perform data access. *% Dat_ref cycles* is the percentage of cycles that the program spends in executing instructions that refer data.

instruction `PMADDWD`. The multiply accumulate is an expensive MMX instruction relative to other MMX instructions, requiring three cycles in the Pentium. There is packing and unpacking overhead (6% of all instructions and 14% of MMX instructions in the FFT) that accompanies multiplication, but the MMX version is still more efficient than the non-MMX equivalent.

The FIR kernel shows similar results to the FFT kernel, with 1.5 times speedup from the non-MMX to MMX program and dynamic instruction are reduced by a factor of 2.3. Also, like the FFT, the FIR is bogged down by MACs which represent 6% of all the instructions and 12% of the total cycles. The IIR kernel is the only program we studied that does not show a speedup when using the MMX library calls. The unmodified C program actually runs 3.7 times faster than the MMX enhanced program and the dynamic instructions increase by a factor of 3.0. The IIR filters for this kernel are small. The short filter length and corresponding number of coefficients remove data parallelism and the amount of useful work that can be done on each pass. In addition, a cursory look into the MMX assembly code shows that on each pass the filter implementation is performing a large amount of error checking and conditional

Benchmark programs	Speedup	Dynamic Instructions	Data Memory References	Data Memory Ref. Cycles
FFT	1.49	2.35	2.65	1.52
FIR	1.54	2.28	2.36	0.99
IIR	0.27	0.33	0.48	0.38
MatVec	5.71	5.94	4.26	2.49
Radar	1.26	1.38	1.73	1.32
Image1	6.16	6.97	6.67	2.32
Image2	5.29	18.30	10.67	4.81

Table 3: Results in ratios of the Non-MMX program to MMX program
Speedup is calculated as the ratio of clock cycles (obtained using VTune). *Data Memory reference* is any assembly instruction that uses any memory referencing mode. *Dynamic instructions* are instructions that actually get executed during the running of the program.

branching based on this error checking. In this particular C implementation for the IIR, the use of Intel's MMX library degrades performance.

On the other end of the MMX spectrum, the matrix-vector kernel is well-suited for an MMX implementation. The execution time speedup due to MMX is 5.7 times and the dynamic instruction reduction is 5.9 times. Note that this kernel operates on 16-bit data, so four pieces of data can be operated on in parallel, yet the improvements are by factors of nearly six. The difference in execution time is largely due to the `imul` instruction which does integer multiplication in about 10 cycles versus the `pmaddwd` MMX instruction which can do two multiplications in 3 cycles. The dynamic instruction size reduction is due in large part to more efficient maintenance of the loop in the MMX code.

The radar application has somewhat disappointing results even though all of the arithmetic is accomplished using MMX vector or FFT routines. The execution time speedup is 1.3 times more with MMX code and the dynamic instructions are reduced by 1.38 times. Although several MMX routines are called, only 20% of the instructions turn out to be MMX instructions. One shortcoming of the MMX application is that 33 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 9.1% of the total cycles without including the penalty for passing parameters.

The image applications show the highest speedup of all programs. 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 some "automatic" packing and unpacking of data by simply loading and storing quad-words (64 bits) from memory. Finally, the image processing routines require no arithmetic with neighboring pixels. Since only the pixels from another image or data array are used, the processing has high data parallelism that can be exploited by using MMX.

The dimming image program primarily performs multiplication. Since MMX multiplication interleaves the high and low bytes of the result, some unpacking and re-packing is required. About 25% percent of the instructions in this program are pack and unpack instructions. Although this seems like a large overhead for MMX multiplication, we see that the dynamic instructions are still reduced by a factor of seven from the highly optimized original code, and the execution time reduces by a little more than six times. The color switching image program does a logical XOR between two arrays. Our results show that no packing or unpacking is performed, the dynamic instructions are reduced by a factor of 18, and the execution time is reduced by almost a factor of six.

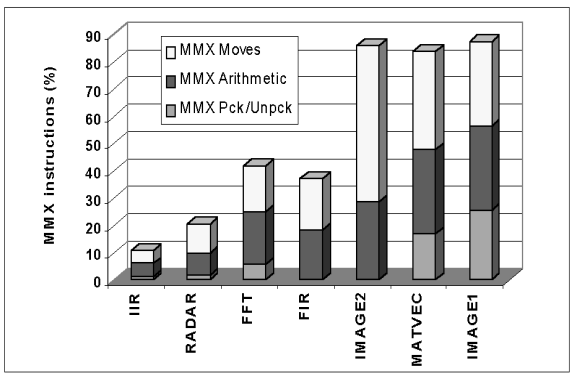

Figure 1. Breakdown of MMX instructions

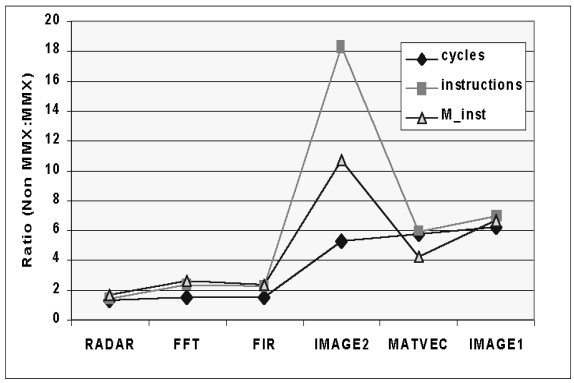

Figure 2. Correlation between speedup, ratio of dynamic instructions and data references

Figure 1 shows the breakdown of the MMX instructions for the MMX-enhanced program. The benchmarks are shown in the increasing order of speedup that we observe. It is seen that programs with a higher percentage of MMX instructions result in more speedup. Note that the percentage of MMX instructions that can be used in a program is a characteristic of the algorithm or the kernel and some kernels and algorithms are better suited for MMX. It is also seen that programs with the highest speedup happen to have data packing and unpacking instructions (15%-22%). It is a common notion that packing and unpacking data would offset the advantages of using MMX. However, we see that the programs which have this overhead show a significant speedup over the corresponding non MMX programs. Figure 2 shows the benchmarks sorted in ascending order of speedup along the X-axis. The graph also shows the ratio of dynamic instructions and instructions that are data references for each benchmark. It is observed that in all the benchmarks except `image2`, the ratios are nearly identical. The speedup for `image2` does not correspond to the ratio of instruction and data references as in the other benchmarks. This is partially due to the Pentium multiply. The multiply instruction with MMX technology takes three cycles, while the corresponding integer multiply instruction on the Pentium takes 10 cycles. This is what causes Matvec and `image1` to have a higher speedup than `image2`.

5 Conclusions

We have 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 developed several parameters on which to evaluate native signal processing performance enhancement, including execution time, dynamic instruction size, instruction mix, and number of data references. In addition, we made observations about the designs of the kernels and how they affect the level of performance that MMX can provide.

The following are some of our observations:

- MMX technology can provide significant speedup in digital signal processing and multimedia applications. The speedups ranged from 1.2 to 7.5 for the various benchmarks.
- 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.

- The best performance increase will always be obtained by tailoring MMX code to fit the application and refraining from hierarchical function calling, but function libraries are a viable option for obtaining speedup. However, there is potential overhead, and efficiency issues arise when using flexible, robust library functions.
- 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 [16].

Future work will consist of incorporating larger and more common applications such as JPEG image compression, MPEG video decoding, and various methods of speech coding [9] [17]. We would also like to reimplement the IIR filter as a higher-order filter or as an IIR biquad. Analysis on a state-of-the-art processor, specifically Intel's Pentium II, is being performed. Instead of obtaining C code and forcing the MMX version to fit that code, we will try targeting our kernels for MMX. It will be also worth trying techniques like data alignment, array padding, and in-lining to see what effect they can have.

References

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

[2] 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.

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

[4] Intel Literature, Mt. Prospect, IL, USA, *Pentium Processor Family Developer's Manual Volume 3: Architecture and Programming Manual*, 1995.

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

[6] L. Gwennap, "Intel's MMX Speeds Multimedia," *MicroProcesor Report*, vol. 10, no. 3, 1995.

[7] G. Blalock, "The BDTIMark: A Measure of DSP Execution Speed"," 1997. Berkeley Design Technology, Inc.

[8] P. Lapsley and G. Blalock, "Evaluating DSP Processor Performance," 1996. Report from Berkeley Design Technology, Inc.

[9] 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.

[10] 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.

[11] 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.

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

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

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

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

[16] D. C. Burger, J. Goodman, and A. Kagi, "Memory Bandwidth Limitations of Future Microprocessors," *Int. Symp. on Computer Architecture*, pp. 78-89, May 1996.

[17] A. S. Spanias, "Speech coding: a tutorial review," *Proc. of the IEEE*, vol. 82, pp. 1541-1582, Oct. 1994.