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

Ravi Bhargava and Ramesh Radhakrishnan

May 8, 1998

Abstract

It has become apparent that proper use of native signal processing(NSP) instruction set enhancements can result in speedup for targeted applications. Our studies are intended to 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 further comparisons based on detailed dynamic instruction profiling. The comparison is done between a suite of Digital Signal Processing (DSP) and multimedia programs implemented in C code and the same programs making calls to an MMX library to perform filtering, vector arithmetic, and other relevant kernels. As expected, our analysis shows decreased execution time for most, but not all, of our MMX programs versus 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 that 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 to utilize it most efficiently.

1 Introduction

Recent times have produced an increasing demand for digital signal processing and multimedia capabilities on a personal computer. 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 [2]. This technology adds new assembly instructions and data types to the existing ISA in an attempt to exploit the data parallelism that is often available in these types of applications.

1.1 Background

MMX and implementations of NSP [14] [5] on other processors exploit the single-instruction multiple-data (SIMD) instruction format. One SIMD data type is sent to a 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 only multiply only 8- and 16-bit fixed-point data.

The Pentium processor provides superscalar performance without dynamic execution by utilizing two pipelines called the U (integer) pipe and the V (floating point or integer) pipe [10]. MMX technology allows two MMX instructions to be executed per instruction. From a programmers point of view, MMX technology maintains full compatibility with existing operating systems and applications by aliasing the MMX registers and state on to the floating-point registers and state. Therefore, floating point and fixed point operations can not be mixed without a high performance cost (the `EMMS` instruction) [9] [6].

When discussing the implementation of many traditional DSP applications, some algorithms resurface more frequently than others [3] [11] [5] [9] [2] [18] [15] [12]. The most common kernels to benchmark are the finite impulse response (FIR) filter, infinite impulse response (IIR) filter, fast Fourier transform (FFT), least mean squared (LMS) adaptive filters, matrix-vector arithmetic, and variations on these. Applications most often benchmarked are image processing, audio compression, speech and signal processing, and MPEG video decoders.

There have been some efforts to analyze NSP on a general-purpose processors [14] [13], and efforts to compare applications with MMX instructions versus applications without MMX on the same X86 processor are incomplete [9]. The results found in [9] are only anticipated results based on simulation. A benchmarking of several applications on the Ultra-Sparc using the comparable

Visual Instruction Set (VIS) showed a performance speedup for all applications. Applications with FIR filters showed the most improvement while IIR filters and FFT exhibited little or no performance increase [5]. In this study, we investigate the performance of a suite of DSP and multimedia programs. Our studies are conducted on a 166 MHz Intel Pentium processor with MMX technology.

1.2 Objectives

Although MMX presents the opportunity for performance increase, to our knowledge there have been no evaluations to corroborate this. Our first objective is to observe speedup results for our benchmark of kernels and applications. Based on our observations, we would like to 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, functions calls, number of MMX instructions, and mix of MMX instructions. From these observations we hope to answer some 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 hassle of writing MMX code? The instruction mix of the applications will allow 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. To achieve 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 and more realistic method would be to write generic MMX libraries for common algorithms and kernels and then allow developers to make function calls to these libraries like they would a C library.

Intel provides a suite of performance libraries at their web site [6]. Recent versions of the libraries (including a Signal Processing Library) include functions that utilize MMX. We developed some C code and acquired the remainder from several resources [8]. We were looking for code that is fast and efficient, yet somewhat modular and easy to interpret so that we could interface the Intel libraries. These resources allow us to develop reliable and efficient programs in a relatively short amount of time.

2 Benchmark Programs

We chose four common DSP kernels and two applications that use variations of these kernels to

comprise our MMX benchmark suite. The details about the programs in the suite are provided in Table 1. In the process of creating our two sets of benchmarks (with MMX and without), we were required to make some decisions and concessions 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 needed 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. We do this indirectly through a buffer so that the main engine of any program is reading and writing to memory and not to disk. It is this section of code that we monitor for performance.

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. Filter length of one, history array of two and five coefficients.
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 does power spectral estimation on 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	Reduce the intensity of a Windows bitmap. 480x640 RGB image where each pixel is represented by 24 bits. Essentially vector multiplication.
Image Color Switch	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

The FIR and IIR filter do real-time filtering. This means that the filter functions take in one new input and returns one new output per invocation. 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. The FIR, FFT, and radar programs showed a very small error due to this conversion (order of 10^{-6}). The IIR, on the other hand, showed similar outputs for the first few passes, but soon became unstable due to this loss of precision.

The data parallelism in the FIR and IIR filters could come from the constant filter coefficients and the previously computed values (history) retained while calculating the moving average. In the FFT kernel, there is parallelism available in the input array of known values. In the images applications, all 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 [8][17] .

3 Methodology

First, we produced a working version of the C program. Then, we rewrite the 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 highest level of optimization for maximum speed.

Intel's VTune tool[7] is the source for all of the timing and profiling information. VTune acquires real-time processor data using Intel's on-chip performance counters and is designed for analyzing "hot spots" of code and optimizing them. Though dynamic instruction profiling is not the intended use of this tool, it can do the dynamic analysis we desire but not very gracefully. Profiling is done interactively on a function by function basis. VTune does make a few approximations to obtain the cycle counts, but gives a more accurate timing than timing methods that have to go to the operating system.

Once the programs were compiled, the outputs were compared to see if the results are similar and to verify no significant errors were being made. At this point, we used VTune to analyze programs and obtain dynamic instruction information. Using a parser, which we wrote in C++, we parse VTune output files and collect the relevant statistical data.

4 Analysis of Results

Most of our results are presented as comparisons (ratios) between the original C code and the MMX version of each individual benchmark (see Table 2). Our primary interest is to find out specifically when MMX is performing well and when it is not.

	Real-time Measured Data				Projected
Program	Speedup	Dynamic Instructions	Data Memory References	Data Memory Ref. Cycles	Absolute Speedup
FFT	1.49	2.35	2.65	1.52	1.61
FIR	1.54	2.28	2.36	0.99	2.59
IIR	0.27	0.33	0.48	0.38	0.27
MatVec	5.71	5.94	4.26	2.49	3.59
Radar	1.26	1.38	1.73	1.32	1.15
Image (Mult)	6.16	6.97	6.67	2.32	3.11
Image (XOR)	5.29	18.30	10.67	4.81	8.40

Table 2: 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* are any assembly instructions that use any memory referencing mode. *Dynamic instructions* are instructions that actually get executed during the running of the program. *Absolute speedup* is the projected *speedup* on a scalar Pentium machine with all other hardware remaining the same.

The `FFT` kernel shows modest speedup (about 1.5 times reduction in cycles) and a dynamic instruction reduction of 2.35 times. The FFT uses the widest variety of MMX instructions

including the multiply-accumulate instruction `PMADDWD`. The multiply accumulate is an expensive MMX instruction relative to other MMX instructions, requiring more cycles in the Pentium (three versus one for an add). There is also packing and unpacking overhead(6% of all instructions and 14% of MMX instructions) that goes with multiplication , but it 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 a dynamic instruction reduction of 2.3 times. Also, like the FFT, the FIR is bogged down by multiply-accumulates which represent 6% of all the instructions and 12% of the absolute 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 ran 3.7 times faster than the MMX enhanced program and the dynamic instructions increase by 3.0 times. A more detailed look at the MMX version of the IIR filter provides a little insight into the issues here.

First, 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 is doing a good deal of error checking and conditional branching based on this error checking. In this case, the overhead using Intel's robust, MMX IIR filter is not worth it.

The radar application had 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. Most of the remaining 80% are required to maintain the application. One shortcoming of the MMX application is that 33 times more function calls are made, many of unseen to the user within the libraries themselves. The `ret` and `call` functions themselves consume 9.1% of the total cycles. This does not include the additional penalty for passing parameters.

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 almost six times. 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 maintenance of the loop which is apparently handled more efficiently in the MMX code.

The image application shows the highest speedup of all programs. The most important factor is that 8-bit data allows twice the parallelism 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, note the image processing that we have done requires no arithmetic with neighboring pixels, only pixels from another image or data array and therefore has high data parallelism that can be exploited by using MMX.

The dimming image program Image (Mult) does mostly multiplication. Since MMX multiply interleaves the high and low bytes of the result, some unpacking and re-packing is required. Around 25% percent of the instructions in this program were packs and unpacks. Although, this seems like a large overhead for doing 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 Image (XOR) does a logical XOR between two arrays Our results show that no packing or unpacking is done at all and now the dynamic instructions reduce by a factor of 18 and the execution time reduces by almost six times.

5 Generalizing Results

We realize that the scope of our study is limited to the hardware configurations of one particular X86 architecture. While it is nearly impossible to give a direct relationship between the speedup obtained on the Pentium and speedup one might find on other superscalar X86 architectures, we would like to speculate based on information we have gathered.

We noticed that that the two versions of the code did instruction pairing with differing degrees of success. On a hypothetical X86 machine where pairing would be handled exactly in the same manner, the ratio of projected Pentium absolute speedup (as shown in the last column Table 2) would be the actual speedup experienced.

Another observation that we made is that the reduction in dynamic instructions did not directly translate into increased speedup. This is for a variety of reasons including varying instruction mix and instruction latencies between the two versions. The dynamic instructions do not vary on X86 compatible machines (if code is not recompiled). Therefore a more direct relationship to speedup could occur on a hypothetical aggressive superscalar processor whose ISA requires instructions with similar frequency and latencies.

A final attempt to project MMX results is an observation about memory and resource constraints. For instance, in the FIR kernel the number of memory data references is reduced by 2.4 times with MMX, yet the number of absolute memory cycles actually stays the same. This seems to indicate that the resources of the Pentium are more constraining for MMX code than the non-MMX code. We see that this is the case for all of the benchmarks as seen in Table 2. This could be related to the fact that MMX instructions are often trying to reference 64-bit

pieces of data and C code is more likely to reference smaller data (less than 32 bits) and on a hypothetical processor that treats all data memory references equally, MMX would show greater speedups over non-MMX code.

6 Conclusions

We have analyzed the usage of MMX enhanced libraries to implement 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, absolute cycles, dynamic instruction size, instruction mix, and number of memory references. In addition, we made observations about the designs of the kernels and how that effects the level of performance that MMX can provide.

The following are some of our more significant findings. MMX technology can provide significant speedup in digital signal processing and multimedia applications. The best performance increase will always be obtained by tailoring MMX code to fit the application and refraining from doing hierarchical function calling, but function libraries are a viable option for obtaining speedup. Although, there is a potential overhead and efficiency issues to using flexible, robust library functions.

It is important to note that reducing memory references is just as important as reducing the number of arithmetic operations, especially since going to off-chip cache can be very expensive on a general purpose processor [4]. These results are specific to the Pentium and one can only speculate on speedups on other X86 architectures.

7 Future Directions

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 [16] [12]. An analysis on a state-of-the-art processor, specifically Intel's Pentium II, will be done. 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, etc to see what effect this has on MMX and possible in-lining possibilities[1].

References

[1] D.B. Alpert and M.J. Flynn. "Performance Trade-offs for Microprocessor Cache Memories". *IEEE Micro*, pages 44–53, Aug. 1988.

[2] Garrick Blalock. "Microprocessors Outperform DSPs 2:1". *MicroProcesor Report*, 10:17:1–4, Dec. 1995.

[3] Garrick Blalock. The BDTIMark: A Measure of DSP Execution Speed, 1997. Berkeley Design Technology, Inc.

[4] D.C. Burger, J.R. Goodman, and A. Kgi. "Memory Bandwidth Limitations of Future Microprocessors". *23rd Inter. Symp. on Computer Architecture*, May 1996.

[5] William Chen, H. John Reekie, Sunil Bhave, and Edward A. Lee. "Native Signal Processing on the UltraSparc in the Ptolemy Environment". *Proc. Asilomar Conference on Signals, Systems, and Computers*, pages 1368–1372, Nov. 1996.

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

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

[8] Paul M. Embree. *C Algorithms for Real-Time DSP.* Prentice Hall PTR, NJ, 1995.

[9] L. Gwennap. "Intel's MMX Speeds Multimedia". *MicroProcesor Report*, 10, 1995.

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

[11] Phil Lapsley and Garrick Blalock. "Evaluating DSP Processor Performance", 1996. Report from Berkeley Design Technology, Inc.

[12] Lee, Potkonjak, and Mangione-Smith. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems". *IEEE Micro*, 30:1, Dec. 1997.

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

[14] Ruby B. Lee. "Accelerating Multimedia with Enhanced Mircoprocessors". *IEEE Micro*, 15:2:23–32, Apr. 1995.

[15] 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.*, pages 234–243, Oct. 1996.

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

[17] Ferrel G. Stremler. *Introduction to Communication Systems.* Addison-Wesley Publishing Company, Reading, MA, third edition, 1990.

[18] Vojin Zivojnovic, Harald Schraut, M. Willems, and R. Schoenen. "DSP's, GPP's, and Multimedia Applications - an Evaluation of DSPstone". *Proc. Inter. Conf. on Signal Proc. Appl. and Tech.*, pages 1779–1783, Oct. 1995.