# Quantifying the Effectiveness of MMX in Native Signal Processing

Deependra Talla and Lizy K John
Laboratory for Computer Architecture, Dept. of ECE
The University of Texas at Austin, Austin, TX 78712, USA
{deepu, ljohn}@ece.utexas.edu

***Abstract* - Effectiveness of MMX for Digital Signal Processing on general-purpose processors (often referred to as Native Signal Processing) is evaluated. We benchmarked a variety of signal processing algorithms and obtained speedup (ratio of execution speeds) ranging from 1.0 to 4.0 by using MMX technology over non-MMX code. Efficient, reliable and standard C code is also evaluated with respect to NSP library assembly code from Intel.**

## I. INTRODUCTION

Digital Signal Processing (DSP) and multimedia applications are increasingly being implemented on general-purpose microprocessors found in common desktop computers [1]. General-purpose microprocessors have entered the signal processing oriented stream by adding DSP functionality to the instruction set and also providing optimized assembly libraries. Perhaps, one of the earliest and most visible example of this approach was Intel's Native Signal Processing (NSP) initiative with the introduction of the MMX (commonly dubbed as MultiMedia eXtension) set for the Pentium line of microprocessors. In addition to providing such a media extension, Intel also provides optimized C-callable assembly libraries for signal and image processing [2]. Several other processor vendors have responded similarly with media extensions such as the Visual Instruction Set (VIS) for the UltraSparc processors from Sun, AltiVec for the PowerPC processors from Motorola, Motion Video Instructions (MVI) for the Alpha processors from Digital/Compaq, etc. Also, general-purpose processors have very high clock speeds and advanced architectural features such as superscalar implementation, dynamic execution and branch prediction. In this paper, we evaluate the benefits of MMX technology on a suite of signal processing algorithms. Performance of NSP assembly library versus industry standard C code is also presented.

MMX technology is a single-instruction multiple-data (SIMD) approach to exploit data parallelism in signal processing applications. One SIMD data type contains several pieces of data that is simultaneously processed in the computation unit. For example, processing of image signals typically involves manipulating matrices of 8-bit data. Each MMX register is 64-bits wide. Eight pieces of the image data can be packed into one MMX register and arithmetic or logical operations can be performed on the pixels in parallel with the results being written back to a MMX register. By taking advantage of such a technology, we can operate on 8-bit and 16-bit fixed-point data that is commonly used in speech, audio, image, and other signal processing applications. Fig. 1 below shows an example of the multiply-accumulate operation as performed in MMX technology.

Previous efforts have analyzed the benefits of performing native signal processing using multimedia extensions [3][4]. Performance benefits of MMX instructions over non-MMX code for a Pentium processor were evaluated in [4] on a suite of benchmarks. In addition to benchmarking filters and matrix-vector arithmetic, applications such as JPEG, G.722 speech coding and Doppler radar were also evaluated. A number of DSP and general-purpose processors have been benchmarked by BDTI [5] including a Pentium processor with MMX, however details on performance of individual benchmarks are not publicly available. A Pentium II processor with MMX was compared with a C6x DSP processor in [6]. Performance of NSP library code was also compared with operating system effects included in the results in [6].

In this paper, we quantify the benefits of MMX technology over a wide range of signal processing algorithms (not benchmarked elsewhere in literature) provided in the NSP library from Intel [2]. For quantifying the benefits of MMX technology, we have two versions of each of the signal processing algorithms, one without MMX (non-MMX version) and one with MMX instructions. Based on the number of execution clock cycles taken by each version, we quantify a speedup value taken as a ratio of the two versions. The rest of the paper is organized as follows. Section 2 presents the suite of signal processing algorithms/benchmarks evaluated in this work. Section 3 discusses the methodology adopted. Section 4 reports and analyzes the results. Based on our observations, we provide insight into NSP on general-purpose processors and conclude the paper in Section 5.

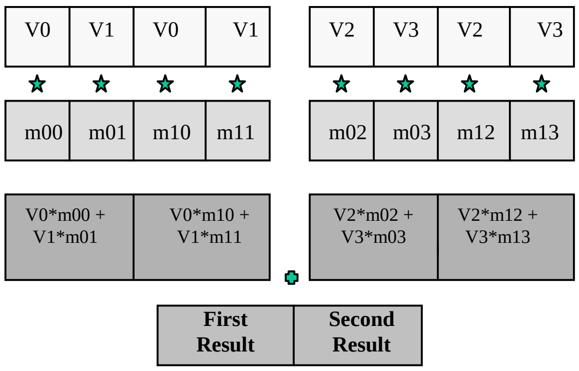

Fig. 1. Matrix-Vector multiplication and add

## II. SIGNAL PROCESSING BENCHMARKS

To evaluate the benefits of MMX technology in signal processing, we chose to obtain code that is well optimized and vendor supplied. Intel provides a comprehensive library of signal and image processing algorithms for use on the Pentium line of processors. We benchmarked a majority of the signal processing algorithms provided in the NSP library. All of our benchmarks use 16-bit fixed-point data type, which is common in digital signal processors. The rest of this section provides some details on the benchmarks.

**Signal Generation Functions** are an integrated part of many applications and are commonly used in various signal processing algorithms for test inputs and equalization. Three signal generation algorithms, Tone (cosine), Triangle, and Gaussian (white noise) waves generating 4,096 samples each were benchmarked.

**Windowing** is an operation that is typically performed to narrow the frequency spectrum and eliminate 'glitches' that result when certain operations such as the Fast Fourier Transform (FFT) are done. In simplistic terms, windowing is multiplying two data vectors together after generating the window coefficients. Five different window functions were evaluated on 16,384 data points each. We benchmarked Hamming, Hanning, Kaiser, Bartlett, and Blackman windows.

**Filters** have the capability to allow certain frequency components in the input to pass unchanged to the output while attenuating other components by desired amounts. These are used in applications such as linear predictive coding, equalization and speech and audio processing. Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters were evaluated. The FIR filter (35th order low-pass) operates on a sample-by-sample basis for 6,000 samples while the IIR filter (5th order high-pass) is a block-filter applied to 10,000 data points.

Three major **transforms**, the FFT, the discrete cosine transform (DCT), and the discrete wavelet transform (DWT) are benchmarked. The FFT converts a signal from the time domain into the frequency domain. FFT-based applications include radar processing, MPEG audio compression, ADSL modems and spectral analysis. A 16,384-point complex FFT was performed. The DCT is commonly used in JPEG and MPEG coding standards. Both the forward and the inverse DCT was performed on an image of 16,384 data elements. The DWT is used in image processing and time-frequency analysis. A 4th order Daubechies wavelet was applied to an 8,192 sized data vector. Both the forward and inverse DWT is performed.

**Convolution** and **Vector-Matrix** operations are commonly performed in several signal processing and multimedia applications. Two 512-data element vectors were convolved. The matrix-vector operations include dot products and matrix-vector multiplication performed on 256 length data points.

## III. METHODOLOGY

We compile the benchmarks using Intel C/C++ 2.0 compiler on a Pentium II processor (with MMX) running Windows NT. All programs are compiled with the optimization "Maximize for Speed". Intel provides a different set of libraries, one for each of its Pentium line of processors. A Pentium Pro processor is exactly the same architecture (P6 microarchitecture) as a Pentium II processor except that it does not have MMX technology. Each of our benchmarks has two versions, one non-MMX version and one MMX version. The non-MMX algorithms were obtained from the Pentium Pro library while the MMX versions were obtained from the Pentium II library. We evaluated the NSP library version 4.0 (the latest available at the time of this work).

We create the benchmarks by taking reliable C programs and then modifying them to use assembly libraries. Each of the assembly library functions has a C-callable syntax. The data to be processed is initialized in buffers and passed on as arguments to the assembly NSP function. We measure the number of execution cycles taken by each benchmark for both the non-MMX and MMX versions. The Pentium line of processors have in-built hardware performance counters for measuring several execution statistics such as number of instructions, clock cycles, and MMX instructions. Gathering information from the performance counters is simple and non-obtrusive (the application is allowed to execute at the normal speed), and allows us to make multiple runs and statistically average the results. We only measure the performance of the main algorithm/function and not the initialization routines, reading from and writing to files and cleanup after processing. Since input data for signal processing applications comes from sources like a sound card, a network card, or an analog-to-digital converter, initialization and file routines are not measured to minimize distortion. Also we do not include any operating system effects in the results. To minimize the effects of cold caches, data is preloaded into the cache before measuring the execution statistics.

We quantified speedup as the ratio of the execution cycles of the MMX version over the non-MMX version. Since all of the benchmarks use 16-bit fixed-point data type, there is an available 4-way parallelism due to MMX (each MMX register is 64-bits wide). But the measured speedup may differ from the theoretical speedup of 4 due to several reasons. Not all the instructions executed in the MMX version are MMX instructions - non-MMX related instructions are still executed and experimental speedup would be less than 4. There is an overhead associated with packing and unpacking the four pieces of data into an MMX register and memory. This contributes to a detriment in the speedup. However, there are factors that contribute to a superlinear speedup (speedup greater than 4). MMX technology has advanced DSP instructions that are not available in the original instruction set such as the multiply-accumulate instruction. Also the latency of MMX instructions is far less than the corresponding non-MMX instructions. For example, the scalar integer multiply in the Pentium family takes 10 cycles, while the multiply-accumulate instruction has only a 3 cycle latency. Furthermore, non-MMX instructions such as the integer multiply are not pipelined and hence throughput is only one in 10 cycles, while MMX instructions are fully pipelined with a throughput of one.

## IV. RESULTS

Table 1 below presents the results of the study with the number of execution clock cycles taken by the non-MMX and MMX versions. In addition the total number of instructions executed for the MMX version and the number of MMX related instructions are also shown. Fig. 2 shows the speedup and percentage of MMX instructions in each benchmark. Benchmarks not shown here did not have any MMX instructions.

None of the three signal generation functions use any MMX instructions. The MMX version of the NSP library is the same as the non-MMX version for the signal generation functions (do not take advantage of MMX). This is because for all of the three benchmarks, floating-point data is internally used to generate the signals and converted to fixed-point. MMX is used only for fixed-point data type and not floating-point arithmetic. In fact, mixing MMX and floating-point code is costly because MMX registers are aliased to floating-point registers. Either MMX or floating-point instructions can be used at one time and not both.

Four of the five window functions except the Kaiser window (no MMX related instructions were executed for Kaiser window) function exhibits an average speedup of 2.0 with each of them executing over 80% MMX related instructions. Each of the window functions involves generating the windowing coefficients and then convolution with the data signal. The overhead associated with packing and unpacking of MMX related instructions exceed the positive factors affecting theoretical speedup.

The DWT did not use any MMX instructions like the signal generation functions. In fact, there is an internal conversion to float data type from the 16-bit fixed-point data input in the source code of the assembly library. The DCT has around 62% MMX related instructions and shows a speedup of 2.55. The DCT involves multiplication of an 8x8 DCT coefficient matrix with an 8x8 data matrix. The DCT coefficients are generated internally by the NSP library function, which is why the speedup is only 2.55 and not higher. Generating DCT coefficients earlier and measuring only the matrix-matrix multiplication can achieve significant speedup. The FFT has a very low quantity of MMX related instructions (3% only) and shows a speedup of 1.02. Careful observation of the assembly code for the FFT showed an internal conversion from 16-bit data to floating-point data and hence uses very few MMX related instructions.

Table. 1. Execution times and instruction counts

| Benchmark | Non-MMX cycles | MMX cycle count | No. of dynamic instrs. | No. of MMX instrs. |
|---|---|---|---|---|
| *Hamming* | 127756 | 66134 | 114931 | 98296 |
| *Hanning* | 125570 | 68336 | 114296 | 98297 |
| *Bartlett* | 58890 | 50948 | 82082 | 65534 |
| *Blackman* | 166334 | 76268 | 131380 | 114682 |
| *DCT/IDCT* | 8203431 | 3220815 | 3816719 | 2383892 |
| *FFT* | 3146705 | 3099504 | 2731424 | 86018 |
| *Matrix-Vec* | 290406 | 106278 | 99115 | 90640 |
| *IIR* | 2171359 | 457600 | 620298 | 455006 |
| *FIR* | 1865746 | 1573396 | 1440093 | 270000 |

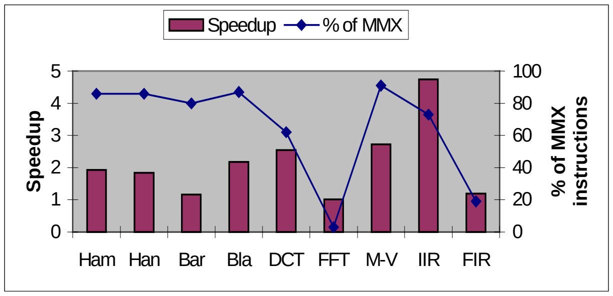

Fig. 2. Speedup and % of MMX instructions

The IIR filter has 73% MMX related instructions and exhibited a maximum speedup of 4.75. This is the only benchmark where there is an experimental speedup that is greater than the theoretical speedup. The benefits of shorter latencies and pipelining of MMX execution units exceeds the packing and unpacking overhead. The IIR filter we used is a block-based filter operating on a large set of data stored in a buffer. The FIR filter on the other hand has 19% MMX related instructions and shows a very low speedup of 1.20. The FIR filter benchmarked is a sample-by-sample basis filter and there is less parallelism available than a block-based filter. Finally, the vector-matrix benchmark had the highest (91%) percentage of MMX related instructions and showed a good speedup of 2.5 over the non-MMX version.

Overall, MMX implementation does provide improvements over non-MMX implementations of the DSP algorithms. But, as was discovered in this work, not all algorithms utilize MMX instructions. Sometimes, due to precision constraints, 16-bit data width may not be sufficient. Floating-point arithmetic is required for certain applications – general-purpose processors have floating-point hardware unlike many digital signal processors that perform floating-point arithmetic in software. Therefore, we benchmark the performance of floating-point versions of the benchmarks. Many times C code is used for several signal processing applications, particularly on general-purpose processors due to the availability of excellent code developing tools. We also evaluated the performance of industry standard C code obtained from [7][8] versus the NSP library.

For the same benchmark suite described in Section 2, we have four versions of each benchmark. A C code version using *float* data type (single precision floating-point), a NSP *float* version, a NSP *double* version (double precision floating-point) and a NSP *short* version (same as the MMX version if available or non-MMX version if the benchmark does not use any MMX related instructions). Figs. 3, 4 and 5 show the results obtained for this comparison.

Speedup in this case is quantified as the ratio of execution clock cycles for each version with the C code as the baseline (speedup of C version is always 1). Performance improvement is obtained when using NSP assembly code over C code in a number of benchmarks. Slowdown is seen in Gaussian and Triangle wave generation functions, Kaiser window function and FIR filter. It is interesting to note that the *short* versions for DWT, Gaussian, Triangle and Kaiser are slower than the *float* and *double* versions. This is because in the assembly code from the NSP library, there is an internal conversion from *short* to *float*. Computation is being performed on floating-point data and results converted back to 16-bit data. To provide robustness, NSP library code has a lot of error-checking routines that slow down the execution performance. As expected the *double* versions of the benchmarks are slower than the *float* versions because of higher latency for floating-point operations in double-precision than single-precision on the Pentium II processor.

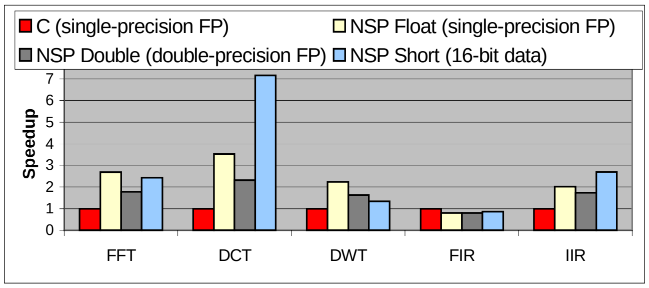

Fig. 3. Speedup for NSP library over C code

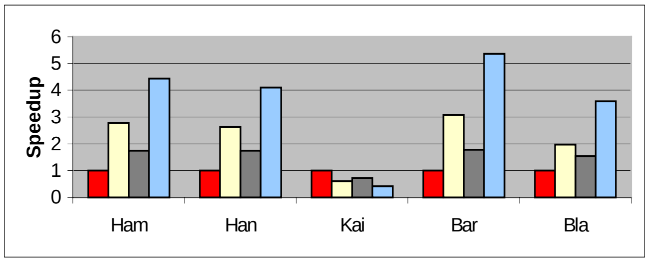

Fig. 4. Speedup for NSP library over C code

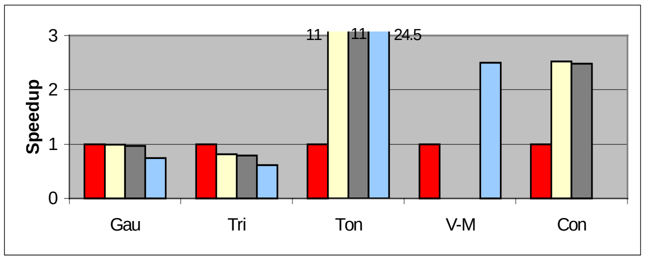

Fig. 5. Speedup for NSP library over C code

## V. CONCLUSION

We quantified the effectiveness of MMX technology in native signal processing. Results show a speedup ranging from 1.0 to 4.7 over the wide range of benchmarks. But not all signal processing algorithms take advantage of MMX technology. Moreover, one has to resort to the NSP library to exploit the benefits of MMX because C compilers to date cannot generate MMX instructions adequately. Unless known earlier for each algorithm, potential of MMX in NSP library cannot be fully realized. C code from industry standard sources and assembly code with different data types was also benchmarked. In general using NSP library code offers an improvement over C code because the assembly code is highly optimized. But in some algorithms the NSP code is slower than C code. Furthermore, 16-bit processing is slower than floating-point processing in some cases because there is an internal conversion from short to float data types and vice versa. Such anomalies make it difficult to predict the performance of an algorithm when using NSP library code.

## ACKNOWLEDGMENTS

This work is supported in part by the State of Texas Advanced Technology Program grant #403, the National Science Foundation under Grants CCR-9796098 and EIA-9807112, and by Dell, Intel, Microsoft and IBM. We would like to thank Numerix Co. Ltd for providing a free copy of Siglib, an industry standard C library for DSP.

## REFERENCES

[1] G. Blalock, “Microprocessors Outperform DSPs 2:1”, *Micro-Processor Report*, vol. 10, no. 17, pp. 1-4, Dec. 1995.
[2] Intel, “Performance Library Suite”. *http://developer.intel.com/vtune/perflibst/index.htm*.
[3] R. B. Lee, “Multimedia Extensions for General-Purpose Processors”, *Proc. IEEE Workshop on signal Processing Systems*, pp. 9-23, Nov. 1997.
[4] R. Bhargava, L. John, B. Evans and R. Radhakrishnan, “Evaluating MMX Technology Using DSP and Multimedia Applications”, *Proc. IEEE Int. Sym. on Microarchitecture*, pp. 37-46, Dec. 1998.
[5] BDTI. *http://www.bdti.com*.
[6] D. Talla and L. John, “Performance Evaluation and Benchmarking of Native Signal Processing”, *Proc. European Conf. on Parallel Processing*, Sep. 1999, “in press.”
[7] Siglib version 2.4. *http://www.numerix.co.uk*.
[8] P.M. Embree, “C algorithms for Real-Time DSP”, NJ: Prentice Hall, 1995.
[9] R.B. Lee, “Accelerating Multimedia with Enhanced Microprocessors”, *IEEE Micro*, vol. 15, no. 2, pp. 23-32, Apr. 1995.
[10] D. Bhandarkar and J. Ding, “Performance Characterization of the Pentium Pro Processor”, *Proc. HPCA-3*, pp. 288-297, Feb. 1997.
[11] J. Bier and J. Eyre, “Independent DSP Benchmarking: Methodologies and Latest Results"” *Proc. Int. Conf. on Signal Processing Applications and Technology*, Sep. 1998.