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

Ravi Bhargava and Ramesh Radhakrishnan

March 10, 1998

Abstract

We intend to investigate the behavior of MMX optimized applications on an X86 general purpose processor. The applications will be written as C programs. The specific applications are yet to be determined, but will be chosen carefully to represent the behavior of DSP applications as a whole. We will analyze the instruction mix and memory access behavior of both applications with MMX instrucions and without using dynamic performance modeling tools. We hope to isolate the specific features of MMX technology that are responsible for speedup then locate and eliminate any potential performance bottlenecks. As part of the literature survey, we looked at various journal papers and other resources desribing the applications, performance evaluation and effective cache utilization.

1 Introduction

The first addition to the X86 instruction set architecture in almost a decade was implemented to increase the performance and handling of emerging multimedia and signal processing applications[8]. This extension has been dubbed MMX (for MultiMedia eXtension). This technology adds new instructions and data types to the existing instruction architecture to exploit the parallelism of these types of applications. General-purpose processors (GPPs) are starting to evolve as the types of applications which they run are changing. The ability to eliminate dedicated Digital Signal Processors(DSP) chips on workstations and PC's in favor of a more productive real-time GPP would result in reduced cost, lower power consumption, fewer software platforms, and increased design flexibility[3].

DSPs remain ideal for most embedded systems. They are relatively inexpensive, small, fast, real-time and low-power. GPPs are continuously making strides in all of these areas.[6] Various benchmarks have shown GPPs, in regard to performance, are now able to handle tasks that until recently were exclusively thought to be for DSPs[3].

2 Native Signal Processing and MMX

Although native signal processing (NSP) has been receiving more attention recently, the basic ideas should are not new. GPPs are simply implementing a few concepts from the DSP realm. This concept is not unique to the Intel Pentium with MMX. Similar extensions have been implemented on the Sun UltraSparc (VIS), the HP PA-7100LC, the Motorola 88110 and the MIPS R10000 [5] [8]. One concept that NSP extensions exploit is that of single-instruction multiple-data (SIMD) instructions. SIMD instruction can operate in parallel on multiple data elements packed into one large 64-bit register. With MMX, this packing can be done using MMX instructions or preferably by doing a 64-bit load on properly alligned data from memory [8] [12].

Including packing instructions, MMX technology adds 57 new 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 by the new instructions. Multiply-accumulate (MAC) was one DSP application specific instruction added to the instruction set. It can only multiply upto 16-bit data and does this calculation in three cycles.

At an architecture level, two MMX instructions can be executed per instruction in the

Pentium II. Also, 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. [8] [6]

3 DSP Benchmarks

Benchmarking has been done for processors with MMX technology. There are two types of benchmarks in which we were interested. One type is GPPs versus the DSPs. We are interested in whether the GPP is capable of performing at the level of the DSP. Berkeley Design Technologies, Inc does this benchmarking using execution time as their sole criteria. The BDTI Benchmarks are 11 DSP algorithm kernels found in "popular applications", including variations on the ones we discuss in this paper. To model applications, BDTI uses a combination of these kernel results and what they call "application profiling" [4] [11]. There results show that some general-purpose processors outperform dedicated DSPs on DSP tasks by more than 2:1 [3] [2].

The second type of benchmark compares DSP applications with MMX instructions versus applications without MMX on the same GPP. [8] All applications showed some performance increase, with speech recognition and videocompression on the low end and image procesing and audio applications showing as much as 8X performance increase. Benchmarking of several applications on the UltraSparc using the comparable Visual Instruction Set (VIS) showed a performance speedup for all applications. Applications with FIRs showed the most improvement while IIRs and FFTs exhibited little or no performance increase. [5]

4 Proposed Kernels

When discussing the implementation of many traditional DSP applications, the same algorithms keep resurfacing more frequently than others. [4][11][5][8][3][16][14] Although many of these kernels can be obtained in MMX libraries[6], it is important that we understand these algorithms so that we can be most efficient in our use of MMX. The following algorithms are a few of these common kernels and ones that we will use in our study.

Finite Impulse Response Filter A filter allows certain frequency components of the input to pass unchanged to the output while blocking other components. One class of digital

filters is the finite impulse response (FIR) filter. These filters are moving average filters and allow their response to an impulse to die away in a finite number of samples. The output is simply a weighted average of the inputs values. $y(n) = \sum_{k=0}^{M-1} c_k \dot{x}(n-k)$ There is a window of these weights (c_k) that takes exactly the M most recent values of $x(n)$ and combines them to produce the output. [7] [5] Given an input sequence of length N and a filter of length M, the computation will include N*M multiply-accumulate operations. [6]

Infinite Impulse Response Filter The other class of digital filters are the infinite impulse response (IIR) filters. This includes the autoregressive aspect of filtering. IIR filters are realized by feeding back a weighted sum of past output values and adding this to a weighted sum of the previous and current input values. A given order IIR filter can be made more frequency selective than the same order FIR filter making them more computationally efficient. The tradeoff is that the implementation is much more difficult. [7] IIR filters can be either just the autoregressive case $y(n) = x(n) - \sum_{p=0}^{P-1} a_p y(n-p)$ or the most general case with moving averages and auto-regression $y(n) = \sum_{q=0}^{Q-1} b_q x(n-q) - \sum_{p=0}^{P-1} a_p y(n-p)$.

Least Mean Square Algorithm Adaptive filters attempt to find an optimum set of filter parameters based on the time varying input and output signals. The least mean square (LMS) algorithm is one such adaptive algorithm. LMS can be applied to both IIRs and FIRs. The adaptive FIR system transfer function is: $y(n) = \sum_{q=0}^{Q-1} b_q(k) \dot{x}(n-q)$ where $b(k)$ indicates the time-varying coefficients of the filter. The LMS algorithm updates the filter coefficients based on the method of steepest descent, which is the following (vector notation): $\mathbf{B}_{k+1} = \mathbf{B}_k + 2\mu\epsilon_k \mathbf{X}_k$ where $\mathbf{B}_k$ is the coefficient column vector, μ is a parameter that controls the rate of convergence (very critical), and ϵ_k is the error signal. [7]

Fast Fourier Transform The fast Fourier transform (FFT) is a very efficient algorithm for computing the discrete Fourier transform (DFT) of a sequence. The DFT is represented as follows: $X(k) = \sum_{n=0}^{N-1} x(n) \dot{e}^{-j2\pi kn/N}$ and can be simplified to $X(k) = X_{ev}(n) + W^k{}_{N/2} X_{od}(n)$ where $W^{nk} = e^{-j2\pi kn/N}$, X_{ev} represents the even elements and X_{od} represents the odd elements. From this point, the DFT can be divided into even and odd halves repeatedly until one is left with only two point DFTs to evaluate. [7] This core computational block in an FFT is referred to as a butterfly. Each butterfly only requires one multiplications and two additions. In the original DFT, $O(N^2)$ multiplies are required, but using FFT the number of butterfly computations required is $\frac{N}{2}\log_2(N)$. [6] [7]

Vector-Matrix Multiplication Taking advantage of these algoritms will require that each be reformulated as a vector-matrix multiplication. [5] A native vector/matrix multiply algorithm traverses the matrix in columns. Matrices are typically arranged in row order leaving the column elements scattered in memory. Therefore, the straight forward approach - applying SIMD techniques to the inner loop - is not feasible. Instead, in an algorithm for X86 media extensions, the matrix should be split into elements of 4x2, and the input vector is split into elements of two. The overall task can be broken down into multiplications of these fragments. The revised algorithm would work on four columns of the matrix in parallel, and accumulates results in a set of four accumulators. Overall, the number of iterations through the loops is reduced by a factor of eight, at the cost of a little more work per iteration. [6] It is this type of enhancement that makes MMX most effective.

5 Proposed Applications

We propose to do at least two DSP applications for this project and possibly more based on time and feasibility. The researched benchmarks showed little consistency in chosen applications, but all used the same core kernels as mentioned above. The following are applications for which we have acquired C code and are strongly considering.

Linear predictive coding (LPC) is a method for speech coding. This particular version of an LPC is for a 2400 bits-per-second voice coder and operates on 16-bit or 32-bit data. The LPC-10 coder uses a 10-th order predictor to estimate the vocal-tract parameters and has several filtering stages including one for an FIR.[15] [13]

CD to DAT Conversion takes audio intended for a compact disc(CD) player and converts it to digital audtio tape(DAT) format. This is basically a sampling and interpolation problem since CD is sampled at 44.1 KHz and DAT is sampled at 48 Khz. Our C code, obtained using Ptoelmy, requires three FIRs in series and the audio samples are 16-bits.

YUV to RGB Conversion does conversion between YUV, the broadcast video display format, and RGB, the format for computer monitors. This application uses mostly matrix multiplication. This application is attractive because colors are often represented with 8-bits and we would like to have at least one 8-bit data application.

Doppler Radar Processing is one possible FFT Applications for which we have obtained C code. This application does the processing necessary to remove stationary targets and estimate the frequency of the remaining Doplar Signal. The problem with the FFT is figuring out how complex numbers are represented in the C code, how they are represented

in the provided MMX library functions, and getting the formats to match.

ARMA modeling of signals, (Auto-Regressive Moving Averages) which is used in spectral estimation, can be done using an IIR with the LMS adaptive algorithm,. The core kernel in this application is a biquad IIR.

We feel that any combination of these DSP applications will provide a good feel of how MMX affects performance in this area as a whole. After some more in-depth look into the MMX libraries, compilers, and mentioned C code we will choose which applications we would like to use in our study.

6 Caching with Multimedia Applications

The applications for multimedia or DSP are very data intensive and therefore cache performance plays a significant role in the overall performance. Since the earlier work on benchmarking was mostly done on kernels we feel that this overlooked the effect of the cache and memory latency. The benchmarking results that we found consisted mainly of execution time and cost/performance ratios [4] [3]. In our study we will also look at the effects of of various caching schemes and how much the memory latency affects overall performance. We will try to improve performance by hiding the memory latency as much as possible.

Caching On Digital Signal Processors DSP processors require multiple memory accesses within one instruction cycle for most of the DSP algorithms like an FIR filter. Havard architectures achieve multiple memory accesses per instruction by using multiple independent memory banks connected to the processor data path via independent buses. Other ways to achieve higher bandwidth include using fast memories that support multiple, sequential accesses per instruction cycle over a single set of buses and using multi-ported memories that allow multiple concurrent memory accesses over two or more independent set of buses. A processor might use on-chip memory that can complete an access in one half of an instruction cycle. Caches in DSPs are generally much smaller and simpler than the caches associated with general-purpose processors [10]. The major types of DSP processor caches are:

Repeat Buffer: This is a one word instruction cache that is used with a special *repeat* instruction. A single instruction that is to be used multiple times is loaded into the buffer, and subsequent executions fetch the instruction from the cache, freeing the program memory to be used for a data read or write access.

Single-sector instruction cache: This is a cache that stores some number of the most recent instructions that have been executed. If program flow of control jumps back to an instruction that is in the cache, the instruction is executed from the cache instead of being loaded from the program memory and,

Multiple-sector instruction cache: This type of cache functions like the single-sector cache, except that two or more independent segments of the program memory can be stored.

Traditionally, the exploitation of memory organization in DSPs has been the responsibility of the programmer. The programmer has to allocate data manually by using assembler directives, if programming is done in assembly or giving the compiler hints, if programming in a high level language. This allows a deterioration in performance if the programmer is unaware of the memory hierarchy or if the cache is transparent to the programmer. Another approach to generating DSP code is using the Ptolemy package developed at the University of California, Berekely. Ptolemy enables designers to specify embedded applications in the form of hierarchical dataflow graphs of functional blocks. The resulting code may not take the cache into consideration and more effective code can be produced if code optimization is done.

Caching On General Purpose Processors Cache memories in a GPP bridges the gap between fast microprocessors and relatively slow memory. Cache memory holds recently referenced regions of memory and reduce the number of cycles the processor must stall while waiting for data. Caches work well for programs that exhibit good locality. Tweaking the source code or other hacks can help to increase the cache performance by altering the memory reference pattern. Literature has been published [1] on of the software techniques to improve cache performance. Some of these that could help in improving the performance of multimedia applications such as: (i) **Merging Arrays** (ii) **Padding and Aligning Structures** (iii) **Packing** (iv)**Loop Fusion and Fission** and (v) **Blocking**

Code Transformations for Efficient Caching in Embeddded Processors We have to take into account different parameters like locality of data, size of data structures, access structures of large array variables, regularity of loop nests and the size and type of cache in order to improve the cache performance. We also have to take the potential overheads into account due to the different transformations on the instruction count and the number of execution cycles. We cannot do optimization of cache incurring a large overhead on the instruction count or execution cycles, as in a general purpose processor since we have to

meet real time constrains and code size limitations. Some of the code related issues we will try to focus on will be:

(i) **Data Flow and Locality Analysis** In this we will try to identify the parts in the program where we can have potential gain by means of compiler optimizations. Parts of the program that are potential bottlenecks are also identified.

(ii) **Effect of In-lining of code** After the parts of the program that have potential for optimization are identified, the improvement due to inlining of code would be estimated.

(iii) **Global Loop Optimizations** The next step would be to apply loop transformation over the scope of the entire program so as to improve the regularity and locality of the program. This would result in reduced number of cache and memory accesses. Dependecy bottlenecks and redundant accesses have to be eliminated, but the constrains in code size should also be kept in mind at the same time.

(iv) **Data Layout in Cache** If possible, define the data layout in main memory and caches to obtain maximum cache utilization.

Caching and Power Correlation The relation between the extent of data caching and power has been explored by Kulkarni et al. [9], in their paper a power function is used which is dependent upon access frequency, size of memory, number of read/write ports and number of bits that can be accessed in every access. The total power is given by $\mathbf{P_t} = \mathbf{N_t}.\mathbf{F}(\mathbf{S_t})$, where n_t is the total number of access to memory, S_t is the size of the memory and F is a polynomial function with different coefficients representing a vendor specific energy model per access. It is seen that better usage of cache improves the power cost and therefore improving cache performance for embedded systems is very important.

7 Project Direction

We intend to look more closely at the increased performance provided by the new MMX instructions on X86 processors. First, we must benchmark some applications to indicate when, where, and how much performance increase we are achieving. Then, we would like to characterize the dynamic assembly level instructions and discover how and why the performance increases. Finally, we will propose ways to optimize the use of these instructions based on our findings. We feel that more advanced software and hardware techniques could be used to improve performance even further.

References

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

[2] Jeff Bier. "DSP on General Purpose Processors – An Overview". Berkeley Design Technology, Inc., January 1997. *Presentation to MicroDesign Resources dinner meeting.*

[3] Garrick Blalock. "Microprocessors Outperform DSPs 2:1". *MicroProcesor Report*, 10:17, December 1995.

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

[5] William Chen, H. John Reekie, Sunil Bhave, and Edward A. Lee. "Native Signal Processing on the UltraSparc in the Ptolemy Environment". *Proc. of the 30th Annual Asilomar Conference on Signals, Systems, and Computers*, November 1996.

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

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

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

[9] C. Kulkarni, F. Catthoor, and H. De Man. "Code Transformations for Low Power Caching in Embedded Multimedia Processors". *To be published in IPPS/SPDP*, 1998.

[10] Phil Lapsley, Jeff Bier, Amit Shoham, and Edward A. Lee. *DSP Processor Fundamentals.* Berkeley Design Technology, Inc, Fremont, CA, 1996.

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

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

[13] Theodore S. Rappaport. *Wireless Communications.* Prentice Hall PTR, NJ, 1996.

[14] M.A. Saghir, P. Chow, and C.G. Lee. "Exploiting Dual Data Memory Banks in Digital Signal Processors.". *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–243, October 1996.

[15] A.S. Spanias. "Speech coding: a tutorial review". *Proceedings of the IEEE*, 82:1541–1582, October 1994.

[16] Vojin Zivojnovic, Harald Schraut, M. Willems, and R. Schoenen. "DSP's, GPP's, and Multimedia Applications - an Evaluation of DSPstone". *Proceedings of the International Conference on Signal Processing Applications and Technology*, pages 1779–1783, October 1995.