Evaluating Signal Processing and Multimedia Applications on SIMD, VLIW and Superscalar Architectures

Deependra Talla, Lizy K. John, Viktor Lapinskii, and Brian L. Evans

Department of Electrical and Computer Engineering
The University of Texas, Austin, TX 78712
{deepu, ljohn, lapinski, bevans}@ece.utexas.edu

Abstract

This paper aims to provide a quantitative understanding of the performance of DSP and multimedia applications on very long instruction word (VLIW), single instruction multiple data (SIMD), and superscalar processors. We evaluate the performance of the VLIW paradigm using Texas Instruments Inc.'s TMS320C62xx processor and the SIMD paradigm using Intel's Pentium II processor (with MMX) on a set of DSP and media benchmarks. Tradeoffs in superscalar performance are evaluated with a combination of measurements on Pentium II and simulation experiments on the SimpleScalar simulator. Our benchmark suite includes kernels (filtering, autocorrelation, and dot product) and applications (audio effects, G.711 speech coding, and speech compression). Optimized assembly libraries and compiler intrinsics were used to create the SIMD and VLIW code. We used the hardware performance counters on the Pentium II and the stand-alone simulator for the C62xx to obtain the execution cycle counts. In comparison to non-SIMD Pentium II performance, the SIMD version exhibits a speedup ranging from 1.0 to 5.5 while the speedup of the VLIW version ranges from 0.63 to 9.0. The benchmarks are seen to contain large amounts of available parallelism, however, most of it is inter-iteration parallelism. Out-of-order execution and branch prediction are observed to be extremely important to exploit such parallelism in media applications.

1. Introduction

Digital signal processing (DSP) and multimedia applications are becoming increasingly important for computer systems as a dominant computing workload [1] [2]. The importance of multimedia technology, services and applications is widely recognized by microprocessor designers. Special-purpose multimedia processors such as the Trimedia processor from Philips, Mpact from Chromatics research, Mitsubishi's multimedia processor and the Multimedia signal processor from Samsung usually have hardware assists in the form of peripherals for one or more of the multimedia decoding functions. The market for such special-purpose multimedia processors is primarily in low-cost embedded applications such as set-top boxes, wireless terminals, digital TVs, and stand-alone entertainment devices like DVD players. On the other hand, general-purpose CPUs accelerate audio and video processing through multimedia extensions.

The architecture of choice for general-purpose media extensions has been the Single Instruction Multiple Data (SIMD) paradigm. The Sun UltraSPARC processor enhanced with the "Visual Instruction Set" (VIS) [16], the "MultiMedia eXtensions" (MMX) and streaming-SIMD instructions from Intel [17], the 3DNow! extension from AMD [18], and AltiVec technology from Motorola [19] are examples of SIMD signal processing instruction set extensions on general-purpose processors. Such CPUs will likely take over the multimedia functions like audio/video decoding and encoding, modem, telephony functions, and network access functions on a PC/workstation platform, along with the general-purpose computing they currently perform.

Another paradigm to exploit the fine- and coarse-grained parallelism of DSP applications is the very long instruction word (VLIW) architecture. VLIW processors rely on software to identify the parallelism and assemble wide instruction packets. VLIW architectures can exploit instruction-level parallelism (ILP) in programs even if vector style data-level parallelism does not exist. A high-end DSP processor, the Texas Instruments TMS320-C62xx uses the VLIW approach.

DSP and media applications involve vectors and SIMD style processing is intuitively suited for these applications. Many of the DSP and multimedia applications can use

L. John 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. B. Evans was supported under a US National Science Foundation CAREER Award under grant MIP-9702707.

vectors of packed 8-, 16- and 32-bit integers and floating-point numbers that allow potential benefits of SIMD architectures like the MMX and VIS. Most of these applications are very structured and predictable, and parallelism is potentially identifiable at compile-time, favoring statically scheduled architectures compared to complex dynamically scheduled processors such as state-of-the-art superscalar processors. However, superscalar out-of-order processors have been commercially very successful, and favored by many over architectures that heavily depend on efficient compilers.

Although SIMD and VLIW techniques present opportunity for performance increase, to our knowledge no independent evaluations of applications comparing the two architectural paradigms are reported in literature. Are the aforementioned paradigms equivalent or is any approach particularly favorable for DSP and media applications? This paper is an attempt to understand this issue based on a few media applications and several kernels. First we evaluate the effectiveness of VLIW and SIMD processors for signal processing and multimedia applications choosing one modern representative commodity processor from each category – Texas Instruments Inc.'s TMS320C62xx processor as the VLIW representative and Intel's Pentium II with MMX as the SIMD representative. The Pentium II evaluation utilized the on-chip performance monitoring counters, while the evaluation of the TMS320C62xx processor was done with a simulator from Texas Instruments. Although the C62xx processor is comparable to state of the art general-purpose microprocessors in machine level parallelism, no performance comparison is available except for BDTI's rating, which portrays C62xx's BDTImark to be twice as that of the Pentium-MMX's score [9]. The BDTI rating is based on kernels only.

In addition to the major contribution of evaluating SIMD and VLIW architectures, we also perform an analysis of the performance of these applications on superscalar processors. The Pentium II is a superscalar processor and is used as the baseline (non-SIMD) for comparing SIMD and VLIW architectures. However, since we are working with actual hardware and hardware monitoring counters, we cannot change the processor configuration or perform analysis of tradeoffs. Hence we use the SimpleScalar simulator tools [29] to evaluate the impact of dynamic scheduling on media applications.

Compilers for SIMD and VLIW paradigms are still in their infancy, and the burden of generating optimized code for these processors is still largely on the developers of an application. Achieving the largest performance increase would involve tailoring the source code for each specific kernel or application, often utilizing generic SIMD and VLIW libraries for common algorithms and kernels that can be accessed via function calls. Both Intel and TI provide a suite of optimized assembly libraries on their Web sites [20][21], and both Intel's C/C++ compiler [22] and TI's C62xx compiler [23] allow the use of '*intrinsics'*. The MMX and VLIW technology intrinsics are coded with the syntax of C language, but trigger the compiler to generate corresponding MMX instructions and optimized VLIW code, respectively. Using assembly language libraries and compiler intrinsics, we create SIMD and VLIW versions of our benchmark suite of kernels and applications and analyze the performance impact of media applications on these paradigms. We found that the SIMD versions of our benchmarks exhibited a speedup ranging from 1.0 to 5.5 (over non-SIMD), while the speedup of the VLIW version ranges from 0.63 to 9.0. We also observe that out-of-order execution techniques are extremely important to exploit data parallelism in media applications.

Several efforts have analyzed the benefits of SIMD extensions on general-purpose processors [3][4][5][6][7][8]. An evaluation of MMX on a Pentium processor on kernels and applications was presented in [3]. However, such an analysis on a modern out-of-order speculative machine like the Pentium II is not reported in literature. Performance of image and video processing with VIS extensions was analyzed in [4] and benefits of VIS were reported. It was shown that conventional ILP techniques provided 2x to 4x performance improvements and media extensions provided an additional 1.1x to 4.2x performance improvements. However, our work includes the VLIW paradigm as well. A performance increase by using AltiVec technology for DSP and multimedia kernels was reported in [6]. Performance analysis of MMX technology for an H.263 video encoder was presented in [8]. A number of commercial general-purpose and DSP processors have been benchmarked by BDTI [9] on a suite of 11 kernels. However, only a single performance metric denoting the execution time is released in the public domain for all of the benchmarks together. Moreover, the Pentium II has not been evaluated in their work. In addition their benchmark suite includes only kernels and no applications.

A reasonable benchmark suite was presented in [10], but there are no SIMD or optimized VLIW versions of the benchmarks. Available parallelism in video workloads was measured in [11] with a VLIW architecture. But they assume infinite number of functional units and a powerful compiler with 100% accurate prediction capabilities. Instead, our approach was to use state-of-the-art SIMD and VLIW commodity processors and realistic commercial compilers. An implementation of MPEG-2 video decoder on a C62xx was presented in [12] and they also compare the performance of the various MPEG components with MMX, HP MAX [7], and VIS. Several DSP processors and compilers were benchmarked in the DSPstone methodology [13], but only one application is benchmarked. A recent industrial consortium is the EEMBC effort [14] for

benchmarking commercial processors for embedded applications. However, source code is not available in the public domain. Effectiveness of Intel's Native Signal Processing (NSP) libraries was evaluated in [15]. Performance of a C62xx versus a Pentium II with MMX on DSP kernels is also reported. However, no applications are incorporated. Jouppi and Wall [30] demonstrated the approximate equivalence of superscalar and superpipelined architectures, however, a decade and several ILP and SIMD processors later, we still do not have adequate quantitative studies comparing the performance of the various paradigms.

The rest of the paper is organized as follows. Section 2 describes the architectures modeled and Section 3 describes the benchmarks and the experimental methodology, and Section 4 analyzes the results. Section 5 concludes the paper.

2. Architectures

In this section, we describe the commodity processors chosen for this study – a Pentium II processor with MMX as a SIMD representative and the C62xx as a VLIW representative. The popularity of MMX, the availability of performance monitoring counters and tools, coupled with the availability of Intel media-specific libraries made MMX a natural choice for the SIMD paradigm. Also, the major features of MMX technology are representative of other SIMD-style media processing extensions as well. The C62xx is the highest performance VLIW DSP processor available. (Although it is a DSP processor, it is often classified as a general-purpose DSP processor). The rest of this section describes the architectures.

2.1 Pentium II (with MMX)

The Intel Pentium II processor is a three-way superscalar architecture (capable of retiring up to three micro-instructions per cycle). It implements dynamic execution using an out-of-order, speculative execution engine, with register renaming of integer, floating-point and flag variables, carefully controlled memory access reordering, and multiprocessing bus support [25]. Two integer units, two floating-point units, and one memory-interface unit allow up to five micro-ops to be scheduled per clock cycle. In addition, it provides the MMX execution unit. In our analysis, we used a 300 MHz Pentium II with 16 KB of L1 instruction and data caches and 512 KB of L2 cache.

The MMX technology allows SIMD style computations by packing many pieces of data into one 64-bit MMX register [26]. For example, image processing applications typically manipulate matrices of 8-bit data. Eight pieces of this data could be packed into a MMX register, arithmetic or logical operations could be performed on the pixels in parallel, and the results could be written to a register. Saturation and wrap-around arithmetic are also supported. Multiply-accumulate (MAC), is a fundamental operation in vector dot product, which is common in signal processing, graphics, and imaging applications, and is part of the instruction set extension. Data widths of 8 and 16 bits are sufficient for speech, image, audio, and video processing applications as well as 3D graphics. MMX registers and state are aliased onto the floating-point registers and state, so no new registers or states are introduced by MMX. Maintaining this compatibility places limitations on MMX. The MMX registers are limited to the width of the floating-point registers (MMX SIMD data type uses 64 of the 80 available bits) and mixing of floating-point and MMX code becomes costly because of potential overhead when switching modes (up to a 50 cycle penalty for switching from MMX to floating-point).

2.2 TMS320C62xx

Texas Instruments TMS320C62xx, the first general-purpose VLIW DSP processor, is a 32-bit fixed-point chip. It is capable of executing up to eight 32-bit instructions per cycle. The C62xx processor is designed around eight functional units that are grouped into two identical sets of four units each, and two register files, as shown in Fig. 1. The functional units are the D unit for memory load/store and add/subtract operations; the M unit for multiplication; the L unit for addition/subtraction, logical and comparison operations; and the S unit for shifts in addition to add/subtract and logical operations. Each set of four functional units has its own register file, and a cross path is provided for accessing both register files by either set of functional units.

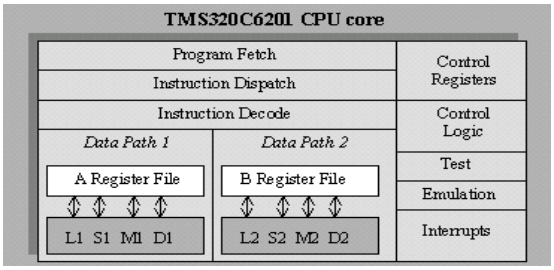

Figure 1. CPU core of C62xx (courtesy of TI)

The pipeline phases are divided into three stages – the fetch stage having four phases, the decode stage being two phases, and an execute stage that requires one to five phases. In the fetch stage of the pipeline, first the program address is generated followed by sending this address to memory, reading the memory and receiving the fetch packet at the CPU. Instructions are always fetched eight at a time and they constitute a fetch packet.

A new fetch packet is not executed until the last instruction in the earlier fetch packet is executed. A fetch packet can take 1 cycle (fully parallel execution), 8 cycles (fully serial execution), or 2 to 7 cycles (partially serial).

Table 1. Summary of benchmark kernels and applications

Kernels	
Dot product (dotp)	Dot product of a randomly initialized 1024-element array repeated several times. Executes 362 million instructions.
Autocorrelation (auto)	Autocorrelation of a 4096-element vector with a lag of 256 repeated several times. Executes 444 million instructions.
Finite Impulse Response Filter (fir)	Low-pass filter of length 32 operating on a buffer of 256 elements repeated several times. Executes 693 million instructions.
Applications	
Audio Effects (aud)	Adding successive echo signals, signal mixing, and filtering on 128-block data repeated several times. Executes 4 billion instructions.
G.711 speech coding (g711)	A-law to μ-law conversion and vice versa as specified by ITU-T standard on a block of 512 elements repeated several times. 163 million instructions.
ADPCM speech compression (adpcm)	16-bit to 4-bit compression of a speech signal (obtained from Intel) on a 1024-element buffer repeated several times. Executes 448 million instructions.

In the first phase of the decode stage of the pipeline, the fetch packets are split into execute packets. Execute packets may consist of one instruction or from two to eight parallel instructions. The instructions in an execute packet are assigned to the appropriate functional units. During the second phase of the decode stage, the source registers, destination registers, and associated paths are decoded for the execution of the instructions in the functional units. The execute stage of the pipeline is subdivided into five stages and different types of instructions require different numbers of these phases to complete their execution. Delay slots are associated with several types of instructions. A multiply instruction has one delay slot, a load instruction has four delay slots, and a branch instruction has five delay slots. Most ALU instructions and store instructions have no delay slots. The interested reader is referred to [27] for a comprehensive description of the C62xx CPU and instruction set. The C62xx has a 4KB L1 direct-mapped cache for instruction and a 4KB 2-way set associative cache for data. It also has an L2 memory of 64 KB that is divided into four banks. Each of the L2 banks can be configured to be either a cache or RAM.

3. Experimental methodology

3.1 Benchmarks

We profile the three signal processing and multimedia kernels and three applications described in Table 1. The kernels and applications in our benchmark suite form significant components of several real-world current and future workloads. All of the benchmarks are implemented using 16-bit fixed-point data except the *g711* speech coding benchmark that operates on 8-bit data. The rest of this section provides some details on the benchmarks. The instruction counts shown in Table 1 correspond to the non-SIMD Pentium II execution.

To study the effects of DSP and multimedia applications on VLIW and SIMD architectures, we created several versions of each of our benchmarks. The baseline non-SIMD versions of the benchmarks are created in the C language. Optimized SIMD and VLIW version were then created utilizing both assembly libraries and compiler intrinsics. We largely developed the non-SIMD code from public sources such as *speech coding resources* [24]. We compile the non-SIMD version of the benchmarks using Intel C/C++ compiler [22] for the Pentium II (running Windows NT 4.0) with the maximum optimization – "Maximize for Speed".

In the process of creating our benchmark versions, we made efforts to ensure that we were comparing equivalent sections of code. Each of the three versions (non-SIMD, SIMD and VLIW) was verified to be functionally equivalent to give the same results after execution. In all the 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. Operating system component of these workloads has been showed to be negligible, and is not expected to skew the results of this study.

Several other kernels such as convolution, Infinite Impulse Response (IIR) response, Discrete Cosine Transform (DCT), and the Fast Fourier Transform (FFT) were also studied, but due to space constraints, we report results for only three kernels. In addition, we prefer to emphasize on applications rather than kernels. The rest of this section describes the code development for SIMD and VLIW versions, tools used and the metrics of performance evaluation.

3.2 Using Assembly Libraries

Intel's assembly libraries [20] provide versions of many common signal processing, vector arithmetic, and image processing kernels that can be called as C functions. However, some signal processing library calls require library-specific data structures to be created and initialized before calling kernels such as *fir*. Using assembly libraries is thus restricted and we used Intel's libraries only for the dot product and the autocorrelation bench-

marks (since only these two benchmarks have the same calling sequence for the C and library functions and the library versions do not use any extra data structures). Unless the code developer can replace a complete function call in C with a call to the library function, the benefit of assembly libraries cannot be utilized completely. There is no loss of accuracy by using MMX because all versions of the benchmarks operate on 16-bit data (except the *g711* speech coding that uses 8-bit data). Another issue with the use of Intel's libraries is that they are generally robust and intuitive, but employ a lot of error checking code to guarantee functional correctness that can potentially increase execution time. Also overhead of using MMX instructions (misalignment-related instructions, and packing & unpacking data related instructions) should be less than the potential benefit of MMX.

TI provides optimized assembly code for the C62xx in [21]. These assembly libraries are C callable and also have the same calling sequence as the C-code counterpart. We used assembly function calls in three of the six benchmarks – *auto*, *fir*, and *aud*. Several restrictions apply for using these C62xx optimized VLIW assembly codes – the *fir* code requires that the number of filter coefficients must be a multiple of 4 and length of *auto* vector must be a multiple of 8.

3.3 Using Compiler Intrinsics

Both Intel and TI libraries are useful only if we can replace an entire function written in C with an equivalent C-callable assembly function call. But in many applications such easily replaceable functions are difficult to find, especially for applications that do not use any of the kernels such as the *g711* speech coding and *adpcm* benchmarks in our case. Also as mentioned using Intel's libraries for creating SIMD versions of the code introduces special data structures and overhead (*fir*). The Intel C/C++ compiler [22] and the C62xx compiler [23] provide intrinsics that inline MMX and C62xx assembly instructions respectively. The compilers allow the use of C variables instead of hardware registers and also schedule the instructions to maximize performance.

For creating the SIMD versions of the benchmarks, we profiled the benchmarks to identify key procedures that can incorporate MMX instructions. The major computation was then replaced with an equivalent set of MMX instructions with original functionality maintained. We unrolled the loops manually to isolate multiple iterations of the loop body and then replaced with equivalent intrinsics. Both non-SIMD and SIMD versions of the benchmarks have the same calling sequence and parameters. The SIMD version that uses intrinsics does an internal conversion to the required SIMD data type (MMX data type with 64-bits), operate on the four data elements in parallel (each data being 16-bits wide except the *g711* that uses bytes) and write the results in 16-bit words. For applications that use intrinsics, the overhead of using MMX should be less than the benefits gained using MMX instructions. We incorporated the intrinsics into *fir*, *g711* and *aud* benchmarks. The *adpcm* could not use any intrinsics.

The C62xx compiler similarly provides intrinsics for inlining assembly instructions into the C code. Some of the compiler intrinsics provided are "multiply two numbers, shift, and saturate", "approximate reciprocal square root", "subtract lower and upper halves of two registers". All of the compiler intrinsics and their detailed descriptions can be obtained from [23]. We used the C62xx compiler intrinsics for creating the *dotp* VLIW version of the benchmark suite. We could not use intrinsics for the *g711* speech coding and *adpcm* benchmarks and rely solely on the compiler to generate optimized code. While generating code using the C62xx compiler, the maximum optimization flag (-o3) was used. This level of optimization performs various optimizations including software pipelining, function inlining, loop unrolling, etc.

3.4 Tools

We used VTune [20] and the on-chip performance counters for the case of the Pentium II and the stand-alone simulator for the C62xx processor. VTune is Intel's performance analysis tool that can be used to get the complete instruction mix (assembly instructions) of the code, and is designed for analyzing "hot spots" in the code and optimizing them. We used VTune to profile static instructions (for the case of non-SIMD and SIMD). Hardware performance counters present in the Pentium family of processors are used for gathering the execution characteristics of both the non-SIMD and SIMD versions of the benchmarks. Gathering information from the counters is simple and non-obtrusive (the benchmark is allowed to execute at normal speed). In addition to the execution clock cycles, the performance counters can be used to obtain the instruction mix and many runtime execution characteristics of the application. For the case of the VLIW code, the execution cycle counts were obtained from the stand-alone simulator that is especially useful for quick simulation of pieces of code [22]. The "clock( )" function provided in the simulator returns the execution times of the benchmarks.

3.5 Performance Measures

We use the execution time of the application as the primary performance measure for this study. Speedup is quantified as the ratio of the execution clock cycles of the SIMD and VLIW versions with respect to the non-SIMD C code. Execution time is expressed in clock cycles and not in absolute units of time as the clock speeds of the two

processors differ and because this paper is a comparison between SIMD and VLIW processing capabilities rather than a comparison of the Pentium II versus the C62xx processor. For the case of SIMD processing, we also collected statistics such as effect of SIMD on CPI, micro-operations, and branch frequencies. While measuring the execution cycle counts of each version of our benchmarks, we only monitor the processing of data already pre-loaded into memory. Input data for DSP and multimedia applications typically come from sources like sound cards, video cards, network cards, or analog-to-digital converters.

The amount of L1 and L2 caches is different between the SIMD and the VLIW processor (16 KB L1 for SIMD & 4KB L1 for VLIW and 512 KB L2 for SIMD & 64 KB for VLIW). We configured the 64 KB memory for the C62xx to be used as RAM and not as cache. We try to fit our data sets as much as possible on the L1 cache to eliminate the effects of memory latencies and use the fastest possible memory for each processor.

4. Analysis of Results

4.1 Comparison of SIMD and VLIW performance

Fig. 2 illustrates the performance of SIMD and VLIW codes over the non-SIMD version. The execution time is presented in Table 2. While interpreting the results, it should be remembered that the baseline (non-SIMD) performance is derived from a 3-way superscalar processor that performs dynamic scheduling to exploit ILP.

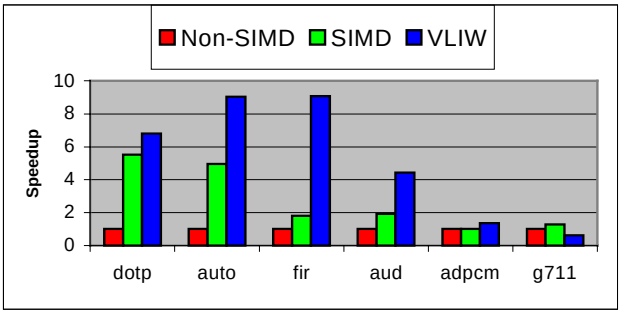

Figure 2. Ratios of execution times

Table 2. Execution clock cycles

Benchmark	Non-SIMD (cycles)	SIMD (cycles)	VLIW (cycles)
Dot Product	181242573	32804388	26600107
Autocorrelation	222023315	44738100	24577801
FIR filter	374628170	208238181	41370004
Audio Effects	2191761094	1148164486	494700006
ADPCM	381143255	381143255	281980004
G.711	109593602	85404734	173190004

4.1.1 Performance of Kernels

Significant speedup is achieved for both SIMD and VLIW versions over the non-SIMD code for the three kernels. The *dotp* kernel shows an improvement of approximately 5.5 times for the SIMD version over the non-SIMD version, despite using 16-bit data. Super-linear speedup is possible due to the presence of the pipelined multiply-accumulate instruction in MMX (latency of 3 cycles). For the non-SIMD case, the integer multiply operation takes 4 cycles. Over 80% of the dynamic instructions in the case of the SIMD version have been found to be MMX-related instructions. The performance of the VLIW version of *dotp* is even better than the SIMD version, with a speedup close to 7. The VLIW code is capable of executing two data elements per clock cycle (in the case of a 1-way scalar processor it would take at least 5 clock cycles for each data element – two for loads, one multiply, one add and one store). Moreover, the C62xx code takes advantage of software pipelining to prefetch data three iterations before it is used. For the *dotp* benchmark, the VLIW code is able to utilize all the eight functional units for the kernel execution as shown in Fig. 3.

Opcode	Unit	Registers	Comment
LOOP: LDW	.D1	*A4++, A0	Load 32-bit word from memory (Two 16-bit words)
LDW	.D2	*B4++, B0	Load 32-bit word from memory (Two 16-bit words)
MPY	.M1X	A0, B0, A1	Multiply two lower 16-bits
MPYH	.M2X	A0, B0, B1	Multiply two upper 16-bits
ADD	.L1	A1, A5, A5	Accumulate
ADD	.L2	B1, B5, B5	Accumulate
[A2] SUB	.S1	A2, 2, A2	Decrement Loop Counter
[A2] B	.S2	LOOP	Branch to *LOOP*

Figure 3. Dot product kernel in one VLIW instruction packet (courtesy of TI)

The *auto* kernel also shows similar performance increase for both the SIMD and VLIW versions. As in the case of the *dotp*, *auto* uses several multiply and accumulates. For the SIMD case, 88% of the dynamic instructions are MMX-related instructions. In the case of the VLIW processor, over 90% of the fetch packets have only one execute packet (indicating eight instructions are able to execute in parallel) and a majority of the remaining 10% of the fetch packets have only two execute packets (indicating an average of four instructions in parallel).

The *fir* benchmark shows a modest performance increase (1.8 speedup) for the SIMD version over the non-SIMD code when compared to the other two kernels. The amount of MMX related instructions in the overall dynamic stream is far less than the other two kernels (29%). Also, the SIMD version needs four copies of filter coefficients to avoid data misalignment. The Intel library version of the *fir* filter actually exhibited a speedup of only

1.6. This was due to additional data structures that had to be defined and error checking code that can potentially decrease performance at improved robustness. The VLIW version exhibits stronger performance boost than the SIMD version. Again as was in the case of dot product and autocorrelation, over 95% of the fetch packets had only one execute packet with all eight instructions executing in parallel. The VLIW kernel codes were hand optimized and presented as assembly libraries. Moreover, the VLIW code had constraints such as the number of filter coefficients should be a multiple of 4 and the size of the *auto* vector should be a multiple of 8.

4.1.2 Results from Applications

Overall, the results of the VLIW versions of the applications are disappointing (when compared to performance improvements obtained in kernels). Both *g711* and *adpcm* involve significant control dependent data dependencies, wherein execution is based either on table lookup or conditional branch statements based on immediately preceding computations. The *aud* application was the only one where any appreciable parallelism could be exploited by the VLIW environment. The VLIW version of the *aud* application exhibits a speedup close to 4.5 over the non-SIMD version. The echo effects and signal mixing components of the VLIW version were unrolled manually eight times. The speedup achieved by the VLIW version of the *aud* application is almost half that of the kernels. This is because the C62xx version was primarily developed in C code and only the filtering component utilized optimized assembly code. The compiler generates the echo effects and signal mixing components. The "interlist" utility of the C62xx compiler provides the user with the ability to interweave original C source code with compiler-generated assembly code. The compiler-generated assembly code for the echo effects and signal mixing components indicates that the compiler is unable to fill all the pipeline slots (several execute packets in each fetch packet). The compiler was unable to software pipeline the echo effects component. This effectively introduced 3 NOPs after every load, which degraded performance. Moreover, even with a loop unrolling of 8, for each one of the eight computations the result was the same with 3 NOPs after every load. Since there is no out-of-order execution in the VLIW processor, loop unrolling in this instance contributes to no performance increase in terms of speed but only increases code size.

The VLIW code for *adpcm* shows a speedup of 1.35 over the non-SIMD and SIMD cases. In this application, the C62xx compiler did not perform any loop unrolling or software pipelining. Since there is no parallelism to be exploited, unrolling will drastically increase the code size with little or any performance increase. Software pipelining was difficult because loads in this application depended on the execution of the conditional branch statements. Thus the compiler-generated assembly code is non-optimal with several branches that are followed by 5 NOPs and loads followed by four NOPs. Most of the fetch packets have eight execute packets (serial as opposed to the desired parallel execution).

The VLIW version of *g711* shows a slowdown (0.63) over the non-SIMD code. However, analysis showed that the base non-SIMD model, which is a 3-way dynamically scheduled superscalar processor achieves an IPC of approximately 2.0. The C62xx code for *g711* has very few packets with more than one slot utilized. Branches are followed by NOPs for 5 cycles in the assembly code. There are also several loads due to the look-up table and NOPs for 4 cycles are inserted in the code. Because of static scheduling combined with no branch prediction, and the control nature of the application, no parallelism could be exploited. Also, the *g711* operates on 8-bit data and the rest of the 24-bits (the C62xx data width is 32-bits internally) is being wasted.

Even the speedup achieved by the applications from SIMD technology is not appreciable. The *aud* application shows a moderate speedup of around 2.0 for the SIMD code over the non-SIMD code. About 28% of the dynamic instructions are MMX-related. Loop unrolling of 4 was used for each of the echo effects, filtering and signal mixing portions of this application. The *adpcm* benchmark does not have any MMX instructions because this algorithm is inherently sequential in that each computation on a data sample depends on the result of the immediately preceding sample. The *g711* SIMD version exhibited a speedup of 1.28 over the non-SIMD code. The number of MMX related instructions are only around 4% and the performance increase is partly due to manual loop unrolling.

4.2 Available Parallelism in Media Applications

Signal processing and media applications typically contain a large amount of parallelism, and the low performance of *adpcm* and *g711* prompted us to examine the benchmarks in detail. Using Tetra [28], a tool from Wisconsin, we analyzed the applications to find the available parallelism assuming infinite functional units, perfect branch prediction, perfect memory disambiguation, and register and memory renaming. Table 3 indicates the results. Ironically, *g711* exhibits the highest parallelism. These results were obtained using Sun Ultra-SPARC code, with *gcc* compiler, with the highest level of optimization possible.

Table 3. Available ILP

Benchmark	ILP with Infinite Window	ILP with window size=32
dotp	3132	3.053
auto	17.53	8.74
fir	20.58	4.13
aud	21.16	4.26
adpcm	33.65	6.12
g711	5316	9.63

4.3 Performance of a Superscalar Processor with similar Functional Unit Mix as C62xx

The poor performance of *g711* on C62xx, combined with the observation that available parallelism in *g711* is extremely high led us to examine the performance of a superscalar processor with similar functional units as C62xx. We configured the SimpleScalar simulator [29] to a configuration approximately equivalent to the C62xx. Due to instruction set architecture and compiler differences, the comparison of execution times is not very meaningful, however, for the prudent experimentalist who will cautiously interpret the results, the comparison is illustrated in Table 4. Except for *adpcm* and *g711*, where the C6x compiler fails to exploit the parallelism, the VLIW architecture performance is better than superscalar architecture performance. We performed studies on 11 other kernels with optimized assembly, and on all of them the VLIW performance was superior. When we used naïve C compilation, C6x was still better than superscalar for one kernel, comparable for 3 kernels and worse for 7 kernels, reaffirming the dependency of VLIW architectures on compiler efficiency.

Table 4. C62x vs. approx. equivalent superscalar

Benchmark	C62xx Execution time (million cycles) (naïve C in bracket)		Superscalar Execution time (million cycles)
dotp	26	(53)	102.9
auto	24.6	(49.5)	147.6
fir	41.3	(148)	271.5
aud	494	(1201)	1557.1
adpcm	281		97.8
g711	173		38.4

We also varied the instruction issue width from 1 to 8, for the superscalar processor with C62xx-like functional unit mix and latencies. The interesting observation from this experiment (see Table 5) is that performance scales up to issue width of 4 and after that return starts diminishing. It is also interesting to make an inference that if the C62xx VLIW compiler could generate packets with at least 4 out of the 8 slots filled, the performance of the VLIW processor would almost match the superscalar equivalent (85%). However, if the compiler is able to fill only 1 or 2 slots, the superscalar equivalent would excel the VLIW counterpart. Similarly, if the VLIW compiler succeeded in filling all eight slots of the packet, which it is able to in the *dotp*, *auto* and *fir* kernels, the VLIW processor's performance will be at least twice better than the equivalent superscalar. We could not obtain the absolute CPI for the case of the C62xx because the simulator does not give the number of instructions executed. For kernels, as mentioned earlier, almost all the eight functional units are occupied in each cycle, but for the applications most of the code is serial.

Table 5. ILP obtained from a superscalar processor with different instruction issue widths

	C62xx-like functional unit mix				*Twice C6x resources*
Benchmark	IPC (N=8)	IPC (N=4)	IPC (N=2)	IPC (N=1)	IPC (N=16)
dotp	4.980	3.325	1.665	0.833	5.531
auto	3.997	3.347	1.714	0.857	4.995
fir	3.707	3.221	1.665	0.847	4.677
aud	3.597	3.086	1.623	0.836	4.490
adpcm	1.591	1.532	1.192	0.782	1.785
g711	4.073	3.027	1.562	0.789	4.081

4.4 Other Observations

Assembly-optimized code vs. Naïve C code Performance: The C62xx processor performance is extremely sensitive to compiler optimizations. Naïve compilation results in 2 to 14 times increased execution time in 11 different kernels that we studied including DCT, IIR, FFT, etc.

Scalability of superscalar performance for media applications: As illustrated in Table 5, performance almost doubles when issue width changes from 1 to 2 and then from 2 to 4, on a superscalar processor with C62xx-like configuration, but issue width of 4 achieves approximately 85% of the performance of an equivalent 8-issue processor. A processor with twice the resources as C62xx and twice the issue width (i. e. 16) and twice the register update unit (RUU) size only achieves an additional 15% improvement in performance.

Performance of a Pentium II-like processor on SimpleScalar simulator: We thought it will be interesting to observe the parallelism achieved by an approximately Pentium II-like superscalar processor on the SimpleScalar simulator and compare it with the parallelism observed in our hardware performance monitoring counter based measurements on an actual Pentium-II. Although the comparison of IPCs between a RISC-style ISA as in SimpleScalar with the x86 ISA may seem unfair, the number of micro-operations per x86 instruction was very close to 1.0 (slightly more than 1.0) in these applications and, hence it is not totally unfair. The close correspondence in IPCs except for *g711* (in Table 6) is encouraging, consid-

ering the widespread use of the SimpleScalar tool suite in computer architecture research.

Table 6. Measurement vs simulation (PII vs SS)

	dotp	auto	fir	aud	adpcm	g711
IPC- PII	2.0	2.0	1.85	1.85	1.176	1.49
IPC-SS	2.22	2.18	2.12	2.06	1.235	2.03

Cases where SIMD parallelism is non-applicable, but VLIW-parallelism is applicable: ILP in applications with no data parallelism can still be exploited with a wider VLIW processor. *adpcm* is the only benchmark that even slightly demonstrates this phenomenon. In the version of *adpcm* that we used, neither compiler *intrinsics* nor assembly routines could exploit any SIMD style data parallelism. The Pentium II is a superscalar processor of machine level parallelism 3 while the C62xx processor has a machine level parallelism of 8. Using the SimpleScalar simulator, we found out that an approximate Pentium II configuration achieves an IPC of 1.23 for ADPCM while a superscalar with the approximate mix of functional units as in C62xx can achieve an IPC of 1.59. This residual ILP is exploited by the wider C62xx, leading to the slightly superior performance of C62xx over the Pentium II.

Cases where MMX performance exceeds C62xx performance: *g711* is the only benchmark where MMX performance (even baseline Pentium II performance) exceeds the C62xx performance. This is due to the inability of the C62xx compiler to schedule multiple instructions in a packet, and the ability of the Pentium II to exploit ILP and achieve an IPC of approximately 2.0.

Impact of out-of-order execution: We simulated an in order microarchitecture with the mix of C6x functional units. The IPCs improve approximately 15% as the issue width increases from 1 to 8, however the IPC values did not exceed 1.0. Hence we concur with Ranganathan et. al. [4] that conventional ILP techniques are important for media workloads. One characteristic feature of these workloads is that most of the parallelism in them is between iterations rather than within the iteration. In fact, most instructions inside an iteration are dependent on the preceding instructions inside the same iteration. Unless branch prediction and out-of-order execution are performed, it is difficult to extract the parallelism in these programs, using an ILP processor.

Impact of MMX on CPI: Although the MMX (SIMD) versions of the benchmarks take less time to execute than the non-SIMD code, the CPI of the MMX version is higher than that of the non-MMX version (see Figure 4). The number of dynamic instructions executed when using MMX is significantly lower than the non-MMX version. Although there is an increase in CPI, each SIMD instruction does four times more work than an earlier instruction. Figure 4 also illustrates the percentage of MMX instructions in the various SIMD style programs.

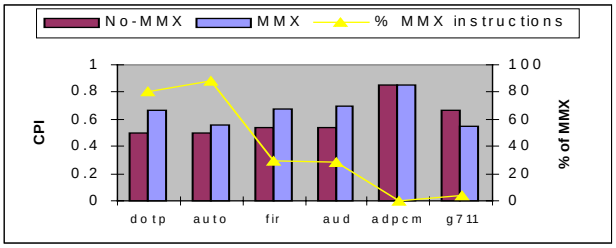

Figure 4. Effect of MMX on CPI

Effect of MMX on branch frequencies: As expected (Fig. 5.), the branch frequency decreases in general except in the case of *g711* that shows a marginal increase (but it has only 4% MMX instructions). The decrease is primarily due to processing multiple elements in parallel and loop unrolling.

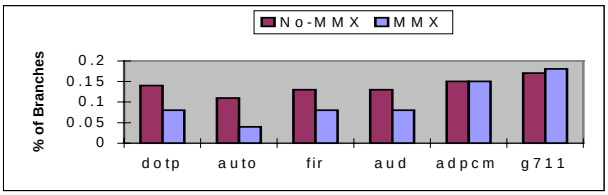

Figure 5. Effect of MMX on branch frequencies

5. Conclusion

This paper evaluated the effectiveness of SIMD, VLIW, and superscalar techniques for signal processing and multimedia applications choosing a representative commodity processor from each category. We observed that:

- SIMD techniques provide a significant speedup for signal processing and multimedia applications. The observed speedups over a 3-way superscalar out of order execution machine range from 1.0 to 5.5.
- VLIW techniques provide significantly greater benefits than SIMD on kernels, but do not maintain the same ratio on applications. Observed speedup over the entire suite of benchmarks ranged from 0.63 to 9.0. For optimized kernels, a factor of 8 improvement was observed over a 3-way out-of-order execution superscalar processor.
- Assuming perfect branch prediction and infinite resources, the available parallelism in our benchmark programs is seen to be between 20 and 5300. Most of this parallelism is between iterations rather than within iterations.
- Out-of-order execution and branch prediction as in state-of-the-art superscalar processors are observed to be extremely important for media applications.

- Although compiler dependency of SIMD and VLIW architectures favor superscalar paradigm, this paradigm does not scale much beyond 4 issue for media applications. Increase in resources required to double the performance would be enormously high.
- Compiler *intrinsics* provide the user with ways to develop both SIMD and VLIW code at a higher level of code development rather than resorting to hand-coded assembly or libraries that may not have required functions or are slow due to overhead.
- Assembly libraries assist in the development of optimized applications, however they often introduce new data structures and overhead in addition to the fact that they may not fit into all applications.
- VLIW techniques (without branch prediction) do not yield any significant performance increase in applications that contain frequent control-dependent data dependencies like the *g711* and *adpcm*.
- SIMD techniques have the potential to reduce the number of dynamic instructions and more importantly the branch frequency (up to half in some cases).
- Compiler technology still needs to improve with both SIMD and VLIW architectures (even though intrinsics provide some ease of programming when compared to hand-coded assembly, it is still up to the code developer to find the data and instruction parallelism).

In future work we would like to evaluate the newly announced Willamette (from Intel) and TMS320C64xx (from TI) processors. We also plan to implement more video and image processing applications such as H.263, JPEG, and MPEG on both SIMD and VLIW processors.

References

[1] K. Diefendorff and P.K. Dubey, "How multimedia workloads will change processor design", *IEEE Computer*, pp. 43-45, Sep. 1997.

[2] C.E. Kozyrakis and D.A. Patterson, "A new direction for computer architecture research", *IEEE Computer*, pp. 24-32, Nov. 1998.

[3] R. Bhargava, L. John, B. Evans and R. Radhakrishnan, "Evaluating MMX technology using DSP and multimedia applications", *Proc. 31st IEEE Int. Sym. on Microarchitecture*, pp. 37-46, Dec. 1998.

[4] P. Ranganathan, S. Adve and N. Jouppi, "Performance of image and video processing with general-purpose processors and media ISA extensions", *Proc. 26th Int. Sym. on Computer Architecture*, pp. 124-135, May 1999.

[5] W. Chen, H.J. Reekie, S. Bhave and E.A. Lee, "Native signal processing on the UltraSparc in the Ptolemy environment", *Proc. IEEE Asilomar Conf. on Signals, Systems, and Computers*, pp. 1368-1372, Nov. 1996.

[6] H. Nguyen and L. John, "Exploiting SIMD parallelism in DSP and multimedia algorithms using the AltiVec technology", *Proc. 13th ACM Int. Conf. on Supercomputing*, pp. 11-20, Jun. 1999.

[7] R.B. Lee, "Multimedia extensions for general-purpose processors", *Proc. IEEE Workshop on Signal Processing Systems*, pp. 9-23, Nov. 1997.

[8] V. Lappalainen, "Performance analysis of Intel MMX technology for an H.263 video encoder", *Proc. 6th ACM Int. Conf. on Multimedia*, pp. 309-314, Sep. 1998.

[9] J. Bier and J. Eyre, "Independent DSP benchmarking: methodologies and latest results", *Proc. Int. Conf. on Signal Processing Applications and Technology*, Sep. 1998.

[10] C. Lee, M. Potkonjak and W.H. Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems", *Proc. 30th Int. Sym. on Microarchitecture*, pp. 330-335, Dec. 1997.

[11] H. Liao and A. Wolfe, "Available parallelism in video applications", *Proc. 30th Int. Sym. on Microarchitecture*, pp. 321-329, Dec. 1997.

[12] S. Sriram and C.Y. Hung, "MPEG-2 video decoding on the TMS320C6x DSP architecture", *Proc. IEEE Asilomar Conference on Signals, Systems, and Computers*, Nov. 1998.

[13] V. Zivojnovic, J. Martinez, C. Schlager and H. Meyr, "DSPstone: A DSP-Oriented benchmarking methodology", *Proc. Int. Conf. on Signal Proc. Appl. and Tech.*, Oct. 1994.

[14] EDN Embedded Microprocessor Benchmark Consortium, http://www.eembc.org.

[15] D. Talla and L. John, "Performance evaluation and benchmarking of native signal processing", *Proc. 5th European Parallel Processing Conference*, pp. 266-270, Sep. 1999.

[16] L. Kohn et al. "The Visual Instruction Set (VIS) in UltraSPARC", *COMPCON Digest of Papers*, Mar. 1995.

[17] Intel, "Pentium III processor home", http://developer.intel.com/design/PentiumIII/prodbref/

[18] AMD, "Inside 3DNow! Technology", http://www.amd.com/products/cpg/k623d/inside3d.html

[19] Motorola, "AltiVec Technology", http://www.mot.com/SPS/PowerPC/AltiVec/index.html

[20] Intel, "Performance Library Suite", http://developer.intel.com/vtune/perflibst/index.htm.

[21] Texas Instruments, "TMS320C6000 benchmarks", http://www.ti.com/sc/docs/products/dsp/c6000/62bench.htm

[22] Intel, "C/C++ compiler", http://developer.intel.com/vtune/compiler/cpp/index.htm.

[23] Texas Instruments, "TMS320C6x Optimizing C Compiler User's Guide", *Lit. Num. SPRU187B.*

[24] Speech Coding Resource. http://www-mobile.ecs.soton.ac.uk/speech_codecs/.

[25] D. Bhandarkar and J. Ding, "Performance characterization of the Pentium Pro processor", *Proc. of 3rd Int. Sym. on High Performance Computer Architecture*, pp. 288-297, Feb. 1997.

[26] A. Peleg and U. Weiser, "The MMX technology extension to the Intel architecture", *IEEE Micro*, vol. 16, no. 4, pp. 42-50, Aug. 1996.

[27] Texas Instruments, "TMS320C6000 CPU and instruction set reference guide", *Lit. Num. SPRU189D.*

[28] Todd Austin and Guri Sohi, "Tetra: Evaluation of serial program performance on fine-grain parallel processors", Technical Report, University of Wisconsin; Also "Dynamic dependency analysis of ordinary programs", *Proc. of the 19th Int. Sym. on Computer Architecture*, pp. 342-351, 1992

[29] Todd Austin and Doug Burger, "The SimpleScalar Tool Set, Version 2.0" TR-1342, Computer Sciences department, University of Wisconsin, Madison.

[30] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines", *Proc. of Int. Sym. on Architectural Support for Programming Languages and Operating Systems*, pp. 272-282, 1989.