DSPs, GPPs, and Multimedia Applications — An Evaluation Using DSPstone

V. Živojnović, H. Schraut, M. Willems and R. Schoenen

Integrated Systems for Signal Processing
Aachen University of Technology
Templergraben 55, 52056-Aachen, Germany

ABSTRACT

The DSPstone evaluation methodology is applied to evaluate performance of fixed– and floating–point digital signal (DSP), and general purpose (GPP) processors with appropriate C compilers. Main goal was to estimate run–time efficiency on code which is representative for baseband processing in multimedia applications. The results show that for DSP–type code, like FIR filtering, DSP processors are superior compared to GPP processors, both in processor and C compiler performance. Also, it is shown that contrary to an established opinion, C compilers for floating–point DSP and GPP processors introduce significant run-time overhead on DSP-type code. This overhead mostly disappears if the programming of fixed– and floating–point DSP processors is done using language extensions.

I. Introduction

The digital signal processing (DSP) market is no longer a small niche of the overall computer market. Especially in multimedia and mobile communications, the market for fixed-point and floating-point DSP processors is constantly growing. Orders to the DSP chip industry have grown from quantities of tens of thousands to millions, or even tens of millions. The number of new companies that are using DSP technology increases from day to day. The DSP chip industry has all reasons to be satisfied with the current situation.

However, some new developments have to be taken seriously by the producers of DSP processors if they want to exploit the multimedia market. Although they are more expensive and less powerful for DSP applications, general-purpose processing (GPP) processors could take over the market opportunities of the DSP processor industry.

In a letter dated January 12, 1995, Microsoft announced the withdrawal of its Resource Management Interface (RMI) specification. The RMI was an attempt to enable the producers of multimedia boards and equipment to easily interface to Microsoft's new operating systems. In the letter to potential independent software and hardware vendors and OEMs Microsoft explained the reason for their withdrawal from RMI:

> *Independent software vendors are looking for solutions enabling them to write device drivers and DSP algorithms only once, regardless of the DSP instruction set. Exploiting the features and capabilities of each DSP-based platform is by nature a device-dependent operation, and existing resource management architectures, including the Microsoft RMI, will not provide this level of independence.*

It is obvious that this would not be a problem with high quality DSP compilers and that this decision enormously favors the native processors - GPP processors - as DSP engines. A huge GPP processor producer has already started an initiative under the name Native Signal Processing (NSP) in order to introduce GPP processors as basic components for multimedia processing. The others will follow in short order.

Compiler efficiency is surely not the main feature guiding processor selection for some DSP application. However, the constantly tightening time-to-market constraints and falling hardware prices could promote compiler efficiency to a factor of highest importance very soon.

The goal of this paper is to provide quantitative measurements of run-time performance on typical DSP algorithms running on commercial fixed-/floating-point DSP and GPP processors. Special attention is devoted to the efficiency of the compilers. Using the DSPstone methodology, the overhead introduced by the compiler is measured and reported.

In this paper we limit the benchmarking suite to the FIR filtering benchmark. As an inevitable part of each baseband signal processing, it is selected as the standard performance indicator. Concentration on one benchmark has surely its disadvantages. However, it enables us to provide an analysis of processor and compiler characteristics in a much deeper fashion than if the results of the whole DSPstone suite are discussed.

After presenting the main features of the DSPstone benchmarking methodology, the paper concentrates on benchmarking results for four fixed-point processors and appropriate C compilers (Analog Devices ADSP 2100, Motrorola DSP 56000, NEC μPD 77016, and TI TMS 320C51), two floating-point processors (Analog Devices ADSP 21060 - SHARC, and TI TMS 320C40) with three C compilers (Analog Devices, Tartan, and TI), as well

as one GPP processor (Intel i586) and its compiler (Borland). Measurement results are presented and discussed.

II. DSPstone — A DSP-Oriented Benchmarking Methodology

In order to explore quantitative characteristics of DSP compilers the Institute for Integrated Systems in Signal Processing of the Aachen University of Technology started the DSPstone project in 1993 [1]. During this project a DSP-related benchmarking methodology was defined which should help in evaluating DSP compilers. The main goal was to get exact quantitative data about the overhead which is introduced if a high-level language and compiler are used for DSP code design.

The basic feature of the methodology is the comparison between the code generated by the compiler and the hand-written assembly code which is used as a reference. The metric distance in execution time and memory utilization is compared and the results are reported. More information about the DSPstone benchmarking methodology can be found in [1].

III. Benchmarking Results

Digital filtering is one of the central operations in DSP applications. Especially in baseband processing, filtering-type algorithms consume most of the available processing time. We have selected the FIR benchmark as a representative for all DSP functions based on a similar computation — multiply-accumulate operation on vectors of variables.

The FIR benchmark of the DSPstone DSP suite multiplies an array of state variables x by an array of coefficients h and accumulates the result in the output variable y, as

$$y = \sum_{i=0}^{N-1} x_i * h_{N-i} \qquad (1)$$

State variables x_i are kept in a tapped delay line of length N that holds the input values from x_i to x_{i-N+1}. Each time a new value appears on the input of the filter, the last $N-1$ values are shifted within the delay line x. The result y is the weighted sum of the delay line content x and the coefficients h. In the benchmark N is set to 16, and the performance per input sample is reported.

The FIR benchmark was applied to a set of 7 processors and 8 compilers. The same C code is used for all targets. Only the data types differ in the case of fixed-point and floating-point DSP processors. For the Intel i586 processor and Borland compiler both, the integer and floating-point version of the FIR benchmark are measured. Table 1 presents the absolute performance of the processors and compilers. For compilers supporting C language extensions, the ANSI C code and the C code with extensions are benchmarked. These results are presented in the a/b form, where a is the result for ANSI C code and b for C code using extensions. For the Intel i586 processor the results for the integer and floating-point benchmark are provided in the form $a - b$, where a is the floating-point result, and b the result for the integer FIR filter. The clock frequency of the processors is selected as the maximum clock of the processor which is reported in [2].

Table 2 presents the overhead in percents.

In order to provide better visualization, the results from Tables 1 and 2 are presented in form of bar graphs on Figures 1, 2, and 3. Figure 1 presents the absolute execution time measured in microseconds for the reference code, ANSI C code, and C code with extensions. On Figure 2 the overhead in execution time expressed in percents for the ANSI C code and the C code using language extensions is provided. Figure 3 provides the results for the overhead in memory utilization expressed in percents for the same C code versions.

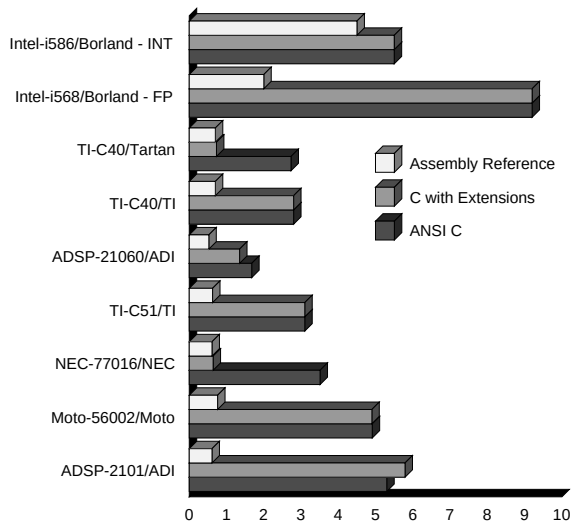

Figure 1: Reference Code - Execution Time [μs]

In sequel a more detailed discussion of the results is provided.

IV. Fixed-Point DSP Processors and Compilers

The fixed-point DSP processors have an architecture which is tuned to be highly efficient on FIR-type algorithms. The Harvard architecture with multiple memory banks, parallel multiply-accumulate instruction, zero-overhead loop, and modulo addressing (circular buffers) enables fast signal processing.

The Analog Devices *ADSP2100* compiler puts the translated code into a zero-overhead loop and uses one parallel instruction. The loop body contains 11 instructions. No `mac` operation is placed within the loop body. Particularly remarkable is the fact that the compiler defines the modify register value `m7` through the register `ay1`, which holds the product $x_i * h_{N-i}$ during computation. This modify register value should be defined out

proc.	ADI 2101 33MHz	Moto 56002 66MHz	NEC 77016 33MHz	TI C51 80MHz	ADI 21060 40MHz	TI C40 60MHz	TI C40 60MHz	Intel i586 90MHz
comp.	ADI 5.1	Moto 1.11	NEC 1.0	TI 6.5	ADI 2.0	TI 4.5	Tartan 2.01	Borland 4.5
t_r [μs]	0.61	0.76	0.61	0.625	0.525	0.7	0.7	2.0-4.5
t_g [μs]	5.3/5.79	4.91	3.51/0.64	3.1	1.675/1.35	2.83	2.73/0.73	9.2-5.5
$c_g(c_r)$	175/191(20)	324(50)	117/21(20)	124(25)	67/54(21)	85(21)	82/22(21)	828-495(180-405)
$p_g(p_r)$	21/23(6)	21(8)	18/7(6)	24(8)	11/10(7)	11(9)	13/9(9)	100-77(53-47)
$d_g(d_r)$	33/33(33)	33(33)	33/33(33)	33(33)	33/33(33)	33(33)	33/33(33)	144-74(130-70)
$m_g(m_r)$	54/56(39)	54(41)	51/40(39)	57(41)	44/43(40)	44(42)	46/42(42)	244-151(183-117)

t – execution time; c – clock cycle count; p – program memory; d – data memory; $m = p + d$
subscript g stands for code generated by the compiler and r for the reference code

Table 1: Absolute Performance

proc.	ADI 2101 33MHz	Moto 56002 66MHz	NEC 77016 33MHz	TI C51 80MHz	ADI 21060 40MHz	TI C40 60MHz	TI C40 60MHz	Intel i586 90MHz
comp.	ADI 5.1	Moto 1.11	NEC 1.0	TI 6.5	ADI 2.0	TI 4.5	Tartan 2.01	Borland 4.5
Δc[%]	775/885	548	457/5	396	219/157	305	290/5	360-22
Δp[%]	250/283	163	200/17	200	57/43	22	44/0	89-64
Δd[%]	0/0	0	0/0	0	0/0	0	0/0	11-6
Δm[%]	38/44	32	31/9	39	10/8	5	10/0	33-29

$\Delta t = (t_g - t_r)/t_r$ [%] – execution time overhead (equals Δc)
$\Delta c = (c_g - c_r)/c_r$ [%] – clock cycle overhead
$\Delta p = (p_g - p_r)/p_r$ [%] – program memory overhead
$\Delta d = (d_g - d_r)/d_r$ [%] – data memory overhead
$\Delta m = (m_g - m_r)/m_r$ [%] – memory overhead

Table 2: Compiled Code - Overhead Introduced by the Compiler

of the zero-overhead loop. Likewise, the content of the `register` variable y should not be stored in the loop. This would decrease the content of the loop body and increase code performance. We could not observe any improvement by explicitly allocating x and h to different memory banks by using C language extensions.

The Motorola *DSP56000* compiler generates parallelized code with 10 instructions in the loop body. The filtering itself is performed with a `mac` instruction in a zero-overhead `do` loop. Each C pointer is translated into an appropriate register at assembly level. Higher code compaction could be reached by putting the register postdecrementing operations after the last use of the register itself within the loop. The compiler places also an entirely useless `nop` instruction as last loop operation, consuming extra unnecessary cycles. The relative high code parallelization results in a low overhead in memory consumption. No language extensions are supported.

The NEC *μPD77016* compiler translates the loop within a zero-overhead instruction. The loop contains 7 instructions, where 2 of them perform parallel operations. No `mac` instruction is generated by the compiler, but two separate multiply and add operations. The compiler sets also a on bit-left shift in order to keep data representation right.

The NEC compiler offers DSP-specific C language extensions. Among others these extensions include the explicit assignment of variables to memory banks as well as a modulo addressing. Using these two features for programming the FIR-filter, the compiler is able to generate highly optimized code, with almost no overhead. The compiled code makes use of the `mac` instruction, with two parallel moves using modulo-addressing. Therefore each tap of the filter can be processed within one clock cycle. This allows to use the `rep` instruction instead of the `loop`. The complete filtering operation is executed within 22 clock cycles, using 21 instructions.

It has to be mentioned that the improvements due to extensions heavily depend on the benchmarked kernel, although for all kernels improvements can be encountered. Reductions in execution time range from 19% (FIR benchmark) to 68% (n_real_update benchmark of DS-Pstone) relative to the results achieved using ANSI C.

The TI *TMS320C51* compiler translates the critical part of the C code, the loop, into a zero-overhead compact structure of 7 instructions. The filtering is performed via separated multiplication, addition and storing operations. The compiler makes an extensive use of the indirect addressing capabilities of the processor, providing a compact assembly translation. The translation of the critical C code part in 7 instructions allows the ANSI C code to be performed faster than at the other targets. The compiler reaches the lowest compiler overhead on ANSI C among all fixed-point compilers. No language

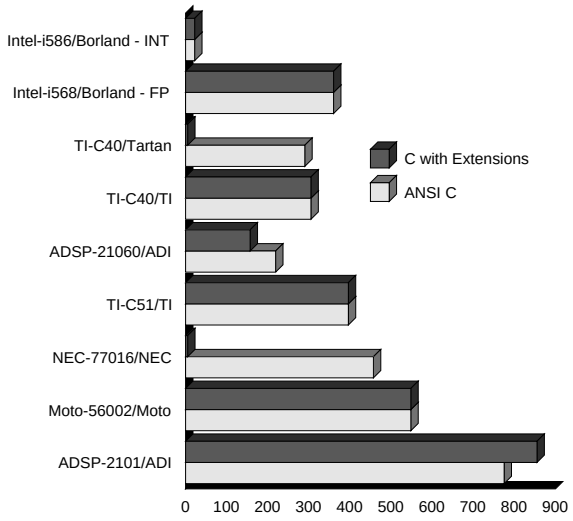

Figure 2: Compiled Code - Overhead in Execution Time [%]

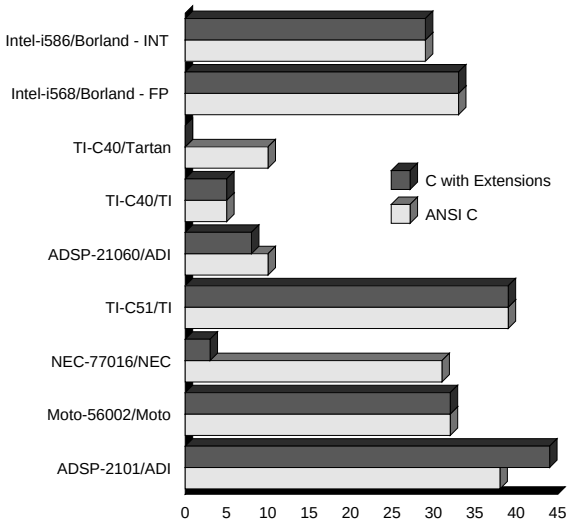

Figure 3: Compiled Code - Overhead in Memory Utilization [%]

extensions are supported.

With an overhead from 400 to almost 800 percent the performance of the fixed-point compilers on ANSI C code is very poor and the code cannot be used without manual tuning. However, if C language extensions are supported and implemented in a proper fashion, like in the case of the NEC *μPD77016* compiler, almost zero percent overhead can be reached.

V. Floating-Point DSP Processors and Compilers

Just like their fixed-point counter parts, the performance of the floating-point DSPs relies heavily on the use of `mac` instructions which enable the processors to compute one filter tap each instruction cycle once the pipeline is set. Assembly reference codes for the floating-point targets need about as many instruction cycles as the fixed-point targets whereas the generated code is in general quite shorter for the floating-point processors. Note that the C40 consumes two clock cycles to perform one instruction cycle.

The Analog Devices *ADSP-21060* compiler places all variables in data memory. The loop is translated into a zero overhead loop which contains four instructions. The two operands x_i and h_{N-i} are fetched in two separate instructions. Instead of using a `mac` instruction the compiler implements a multiply followed by an accumulate operation. The shifting of the delay line is done in parallel to the computation. The limiting factor within this loop body is given by the four data memory accesses. Thus, with all variables in data memory, the shortest possible loop body is generated. Language extensions for this compiler include dual memory support, Numerical C, and circular buffering. For this FIR benchmark, using the keyword `pm` to place the coefficients into program memory proved to be the only helpful extension while the Numerical C iterator and the circular buffering are convenient for C programming but slowed down the performance of the generated code. Nevertheless, Numerical C has improved the compiler efficiency for many other benchmarks.

The Texas Instruments *TMS320C40* compiler generates a non-delayed loop. The loop body contains four instructions. No `mac` instruction is used by the compiler. Multiplication, accumulation, and shifting of the delay line are done sequentially. All C level pointers are transformed into addresses in the auxiliary registers `ARn` for indirect addressing with postdecrementation. Although the assembly code generated by the compiler is quite efficient, use of the `mac` instruction could still improve the code. No language extensions are supported by the compiler.

The Tartan *TMS320C40* compiler was invoked with the `-o` command line option which gives the compiler the directive to balance time and memory optimization. Although we are concentrating on run-time efficiency rather than on memory consumption the `-ot` switch, which optimizes highly in time, was not applied because it causes the compiler to unfold the loop thus creating an unproportional memory overhead. Results show that the loop body stays unchanged compared to the TI compiler. The loop itself is implemented using a delayed repeat instruction. This loop implementation obtains a gain of three instruction cycles compared to the non-delayed solution. The Tartan compiler also supports C level circular buffering. Using these extensions improved the compiler efficiency dramatically. Except for one instruction cycle the generated code is as fast as the reference code. The compiler implements the loop with a delayed repeat instruction. The loop body consists of a `mac` instruction. Therefore, the Tartan compiler and the NEC compiler in

conjunction with their specific language extensions are the only ones which are able to create code near the optimum.

Our results show that compilers for floating-point DSPs, although more efficient than compilers for fixed-point DSPs, are still far from optimum. All compilers lack the ability to implement `mac` instructions from ANSI C code. With an overhead ranging from 219 to 305 percent, the performance of the compilers is not suited for straightforward code design with ANSI C. As especially the Tartan compiler shows, language extensions can improve compiler performance and, therefore, code efficiency decisively.

VI. Intel Pentium GPP processor and Borland C Compiler

Two versions of the FIR filter have been evaluated on the Intel i586 — a floating point version with 32 bit precision using the numeric coprocessor, and a 16 bit fixed point version. Both versions have been compiled with Borland C/C++ compiler with all optimizations (fastest code, i386 instructions) turned on.

The floating point code generated by the compiler consists of assembler statements that reflect the sequential execution order of the C source. Thus inefficient code is produced, because it doesn't utilize the potential parallelism between CPU and floating point unit (FPU), the latter being known as numeric coprocessor for former Intel processors up to i386. While one floating point instruction is started on the FPU, the CPU is free for further commands. Only when synchronization between them is necessary, an FWAIT instruction waits for the coprocessor to finish calculation. The reference assembly code has the FPU commands embedded within CPU code, so a highly parallel execution results. In the code generated by the compiler the utilization of registers for pointers (index registers) is not optimal. In the assembly reference code all variables except the arrays are held in registers.

The reference code holds the accumulated value after multiplication $x_i \cdot h_{N-i}$ on the coprocessors stack instead of storing and retrieving it to/from memory, like the compiler does. Data memory optimization comes along with register use automatically, but a new loop structure saves program space, too. The compiler implements the `for` loop unnecessarily with a test before the first loop instruction. This is not done in the reference code. The epilog section is put into the loop, too, thus saving further program memory space.

The fixed-point versions of the reference and compiled code differ much less than in the floating-point case. However reorganizing the loop and using registers instead of memory, has improved the code for additional 22% percent.

VII. Conclusions

The relatively low efficiency of the fixed-point C compilers on ANSI C code is a result which is not a surprise for DSP programmers. The fixed-point processor architectures cannot be programmed properly using a language (ANSI C) and compiler techniques which are introduced for a quite different type of processing. From the measurement results it is obvious that these problems can be solved. The NEC *μPD77016* compiler and its support for language extensions is a good example.

More surprising is the relatively high overhead of the floating-point compilers. With an overhead in execution time between 219 and 305 percent on ANSI C code, they are more efficient than the fixed-point compilers, but it is hard to expect that they can be directly used for any production-quality code development. Also, in this case language extensions can enormously improve efficiency.

The relatively low efficiency of the GPP compilers is surely something what we have not expected. On a floating-point FIR filter benchmark the Borland compiler was not able to detect the possibility for concurrent operation with the FPU unit. As a result an overhead of 360 percent resulted. This is higher than for any of the floating-point DSP compilers. It can be concluded that even from the compiler point of view, the GPP processors cannot be a serious competitor for the DSP processors as far as typical DSP-type code is concerned. In the integer case the GPP compiler was able to deliver 22 percent overhead in execution time, which is the best result on ANSI C code among all compilers. However, it is still higher than the results obtained by using C language extensions of DSP compilers.

It can be concluded that for DSP-type code DSP processors cannot be replaced by their powerful GPP relatives. Typical application domains, like baseband processing in modems, will still need DSP-like architectures despite of the relative inefficiency of DSP compilers.

We are aware of the fact that FIR filtering cannot be used as the only performance indicator for compilers. However, the concentration on one benchmark in this paper enabled us to discuss the results in more details. The complete DSPstone suite contains over 20 benchmarks and should provide a complete picture about the state-of-the-art DSP and GPP compilers. Our efforts concentrate at this moment on benchmarking of control-type code. It can be expected that in this case the GPP processors and compilers will show all their power. We hope to report these results very soon.

VIII. References

[1] V. Živojnović, J. Martínez, C. Schläger, and H. Meyr, "DSPstone: A DSP-oriented benchmarking methodology," in *Proc. of ICSPAT'94 - Dallas*, Oct. 1994.

[2] M. Levy and J. Leonard, "EDN's DSP-chip directory," *EDN Magazine*, May 1995.