DSPSTONE: A DSP-ORIENTED BENCHMARKING METHODOLOGY

Vojin Živojnović, Juan Martínez Velarde, Christian Schläger and Heinrich Meyr

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

ABSTRACT

A new, DSP-oriented benchmarking methodology is presented. Contrary to the existing, assembly-based DSP benchmarks, the proposed methodology is able to evaluate the performance of DSP compilers and joint compiler/processor systems. In the paper the reference code method is introduced in order to deliver reliable and comparable benchmarking results. As an example, the DSPstone methodology is applied on a set of five state-of-the-art fixed-point digital signal processors with appropriate C compilers. The Analog Devices 2101, AT&T 1610, Motorola 56001, NEC 77016 and TI 320C51 DSP C compilers are benchmarked under the DSPstone, and the results are reported.

I. Introduction

In the last couple of years a large number of DSP hardware and software products flooded the electronic OEM/VEU market. A great deal of users, and especially newcomers to the DSP field, are faced up with serious problems in selecting the appropriate processor and/or tool for their application.

Numerous parameters influence the decision. Although parameters like stable product line and available technical support have to be treated with great respect, the primary parameter is the cost/efficiency trade-off in selecting a particular processor. Every DSP user tries to implement the most efficient algorithm on the least expensive hardware within given time.

Time-to-market constraints and high development costs have raised the demand for DSP development tools and specially for high-level language compilers. However, after more than 8 years from the appearance of the first DSP compiler, assembly programming is still an inevitable part of the DSP software development. Is it possible that in the era when the technology changes every two years, the DSP software development technology looks almost the same for almost a decade? Obviously, it is.

During the discussions with numerous DSP users we had the opportunity to hear that all the shortcomings lie on the inefficiency of the DSP high-level language (HLL), mostly C, compilers. Especially the compilers for fixed-point processors have been declared as almost useless for product development. However, nobody was able to give us some quantitative data about the overhead introduced by the high-level language. Therefore, the primary goal of the DSPstone project was to help providing an answer to this question.

DSPstone is not a point and shoot benchmark with one program and one measure for the overall performance which delivers a rating of compilers or compiler/processor systems. Complex problems, like the evaluation of a DSP system, cannot be treated in this way. DSPstone is a methodology which permits the user to make his own picture about the DSP system he intends to use for his application.

The paper is organized as follows. After the introduction, in Section II some background information is given. In Section III a DSP-oriented benchmarking methodology is introduced. The organization of the DSPstone benchmark suites and the selection of the benchmark programs is presented and the benchmarking procedure is explained. In order to verify the methodology, in Section IV five state-of-the-art fixed-point C compilers (Analog Devices ADSP2101, AT&T DSP1610, Motorola DSP56001, NEC μPD77016 and TI TMS320C51) are benchmarked under DSPstone. Finally, Section V presents the conclusions.

This paper presents only a part of the results and comparisons from the DSPstone final report. For more details refer to [1].

II. Benchmarking of DSP Hardware and Software

In the past benchmarking of digital signal processing hardware was conducted almost only by the chip vendors themselves [2,3,4,5]. Standard DSP algorithms, like FFTs and FIR/IIR filters, have been benchmarked on a particular processor. The processing time was used as the only performance measure. Recently, a DSP hardware benchmarking report coming from an independent source [6] appeared. Speed, memory resources and power dissipation of almost all state-of-the-art digital signal processors have been measured by the authors themselves and the results are reported.

As far as the authors knowledge concerns, there are no references regarding benchmarking of DSP compilers published by independent sources. In [7] the necessity for C-based DSP benchmarking was recognized, but later on no actions followed. DSP hardware/software suppliers have benchmarked mostly own products and reported the results in internal, confidential reports.

III. A DSP-Oriented Benchmarking Methodology

The existing computer benchmarks are not suited for benchmarking of DSPs. Code kernels with lots of string manipulations and file I/O are never or rarely run on DSPs. Even in cases where the standard computer benchmarks could make sense (eg. Dhrystone or Sieve), the results are of little value for the DSP

user. In order to supply meaningful, DSP-relevant benchmarking results, we proposed a new, DSP-oriented methodology, the DSPstone. Although DSPstone differs from standard computer benchmarking, the existing benchmarking know-how is used as its base [8,9,10,11].

III.1. The C Compiler as a Transfer Function

The C compiler is a processing unit converting input information (source code, definition files, compiler flags, etc.) to output information (assembly code, mapping information, etc.) according to some conversion mechanism. For the user the C compiler is a black-box (Fig.1). He controls the input and can judge about the behavior of the unit by observing the corresponding output - primarily the generated assembly code.

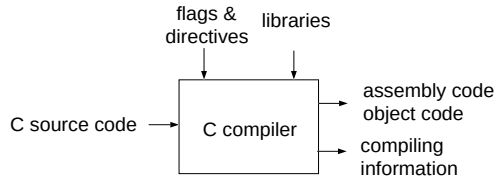

Figure 1: The C Compiler as a Black-Box Processing Unit.

The C compiler is a highly nonlinear system and only one specific input set cannot provide all the information about its behavior. Also, it is not possible to generate a set of orthogonal input programs which test one and only one feature of the compiler. As a consequence, it is not possible to compute analytically the performance of the whole program knowing the performance of a finite set of smaller programs or program fragments. This is the reason why various approaches to benchmark programs selection and measurement evaluation exist.

III.2. Benchmark Program Selection and Classification

The best benchmark is the application itself. However, in most cases we want a performance estimate of the end product at the initial phase of the project. The only way is to choose an application which represents a similar workload for the processor. If it is not available, we can choose computationally intensive program fragments of the application which do some standard processing, like filter, convolution, etc.. The well known 10-90 rule-of-the-thumb tells that 10% of the computation time is spent in 90% of the code. Code fragments where most of the computation is performed can be identified. If the compiler/processor performance for these fragments is given a priori, we can estimate the performance of the whole application in advance.

The DSPstone benchmark consists of the following three suites:

- **Application benchmarks** are complete programs widely employed by the DSP user community. In our case complex DSP applications, like the ADPCM transcoder are used.
- **DSP - kernel benchmarks** consist of code fragments or functions which cover the most often used DSP algorithms (FIR/IIR filters, FFTs, etc.).
- **C - kernel benchmarks** consist of typical C statements (loops, function calls, etc.).

DSPstone is not one program which reflects all the features of a DSP system consisting of compiler and processor, like e.g. Dhrystone. It is a collection of programs with three different levels of granularity corresponding to the three benchmark suites. The user can estimate the worst case overall performance by combining benchmark results of the functional parts forming his application.

III.3. Programming the Benchmarks

The C code of the benchmarks is written without any specific architecture or compiler in mind and in the same way most DSP users and C programmers would do. However, it is hard to decide whether it is better e.g. to use explicit array indexing or pointers with some specific compiler. We are aware of the fact that the benchmark results are influenced by some amount of subjective decision about the question what is actually generic C. We tried to keep this effect on minimum.

Compiler flags and directives are additional information inputs for the compiler. Their proper use can improve drastically the quality of the output for some specific C program. We have applied all those flags and directives of a particular compiler which shorten the execution time of the compiled program.

The memory in most DSPs is heterogeneous. Depending on the distribution of the program code and data on memory, large differences in computation time can be obtain. In order to guarantee fair comparison, in all benchmarks we distributed code and data to minimize the execution time. For some application benchmarks we even used external memory. Thereby, we assumed that the fastest possible external memory is attached (zero wait-state).

The functional equivalence of the C or assembly programs is checked on test sequences which are part of the benchmark. For the C-kernel benchmarks and in cases where the equivalence is obvious no test sequences are specified.

III.4. Metric definition

Instruction Count Metric

Measuring the speed performance of a computer system is mostly done using the program execution time t. The drawback of this metric is that it measures compiler and hardware efficiency in a joint fashion. According to [11] the execution time can be expanded into:

$$t = \frac{instructions}{program} \frac{average\ clock\ cycles}{instruction} \frac{seconds}{clock\ cycle} \qquad (1)$$

The overall performance is a product of three factors:

- **clock rate** - technology and hardware organization dependent;
- **average clock cycles per instruction** - hardware organization and instruction set architecture dependent;
- **instructions per program** - instruction set architecture and compiler dependent;

Unfortunately, these parts are interdependent and do not permit decoupling of the compiler and hardware technology influence on speed performance. Comparing different compilers using only the instruction count per program produces unreliable results. Compilers for LIW (large instruction word) architectures are privileged when compared to RISC (reduced instruction set computer) ones. Also, the memory utilization metric suffers for the same reasons. [1] Despite of all these drawbacks, the instruction count metric has been often used for compiler benchmarking. Simply, there was no alternative.

Reference Code Distance

In order to obtain more reliable compiler benchmarking results, we observed an important difference between general computer and DSP benchmarking. Choosing DSP benchmarks which have functionally equivalent assembly counterparts, we have the opportunity to measure the metric distance between the code generated by the compiler and the *reference assembly code*. In the case of mainframes this is not possible. It is very hard to find a functionally equivalent hand-written assembly version of some standard benchmarking program, like e.g. Dhrystone or SPICE.

We suppose that the reference assembly code is the best or almost the best one which can be written with the given instruction set. How to guarantee this? We suppose that the chip vendors are highly motivated to supply the best possible code for some function in the accompanying libraries or on their bulletin board services (BBS). Our task is to pick up the largest common subset of these programs, check the functional characteristics of the supplied code for equivalence and do the profiling.

The existence of the assembly reference code enables us to benchmark the DSP compiler-processor system in a decoupled fashion. We measure separately the joint performance of compiler and processor using compiled C programs and the performance of the processor alone using the assembly reference code. According to this data, separate evaluation of compiler and processor is possible. This is the basic feature of the DSPstone benchmark. The suggested approach is depicted in Fig. 2. It is obvious that the reference code method can be applied

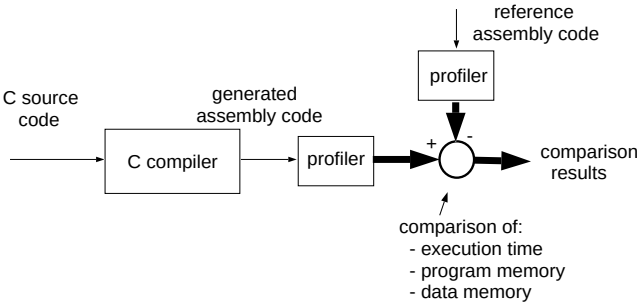

Figure 2: C Compiler Performance Evaluation

independent of the programming language (C, ADA, etc.), tool (compiler, code generator, etc.) or processor type (floating- or fixed-point).

The well known approach for comparing two functionally equivalent programs is to measure their execution time t, number of clock cycles c, program memory utilization p, data memory utilization d and overall memory utilization $m = p+d$. For some given processor clock-cycle period τ the execution time t can be easily converted into number of clock cycles $c = t/\tau$ and *vice versa*.

Every code is a point in the 3D space spanned by c, p and d. If the generated code G is described by (c_g, p_g, d_g) and the reference code R by (c_r, p_r, d_r), then we define:

- execution time overhead

$$\Delta t_g = (t_g - t_r)/t_r \,[\%] \tag{2}$$

- clock-cycles overhead (equals the execution time overhead)

$$\Delta c_g = (c_g - c_r)/c_r \,[\%] \tag{3}$$

- program memory overhead

$$\Delta p_g = (p_g - p_r)/p_r \,[\%] \tag{4}$$

- data memory overhead

$$\Delta d_g = (d_g - d_r)/d_r \,[\%] \tag{5}$$

- memory overhead

$$\Delta m_g = [(p_g + d_g) - (p_r + d_r)]/(p_r + d_r) \,[\%] \tag{6}$$

The introduced overhead measures are the basic metrics which are used in the DSPstone for measuring compiler efficiency. For those users which need the information about the efficiency of the joint software/hardware system consisting of compiler and processor, we also report the absolute measurements for the fastest processor supported by the compiler.

The advantages of the reference code methodology are its simplicity, clarity for the user, unbiased results and simple profiling. The metric gives the answer to the most common question: How large will be the overhead if I decide to program my application in C? The main disadvantage lies in the rigor of the measure. The DSP compiler is mostly *a priori* limited to use some subset of the instruction set. Some features, like bit-reversed or modulo addressing, are excluded from the instruction set seen by the compiler. The reference code distance cannot count for this. The problem can be bypassed in two ways. Implement the instruction, which cannot be reached by compiler, in assembly, or rewrite the reference code in order to exclude unreachable instructions.

III.5. Metric Estimation

The program code can be mostly divided into three parts: initialization, actual processing and post-processing. In the DSPstone the execution time of the actual processing is measured and reported. The problem is how to determine the start and end instructions and how to obtain the necessary resolution. In practice this problem was solved by executing the actual processing in a large number of iterations. In this way the effects of the initialization and post-processing are canceled and the resolution of the measurements is improved.

Using a simulator for time measurements is the only reasonable alternative when the DSP hardware is not available. Even

[1] Improvements can be obtained by introducing the normalization factor which accounts for the differences in the architectures.

on high-performance workstations, the simulation is a rather slow process. If you have to process 1000 samples in the ADPCM transcoder, you have to wait for days. However, the execution time is obtained in processor clock-cycles, which under the assumption that the simulator does his job bug free, guarantees the best possible accuracy. The drawback is the necessity to determine the start and end instructions of the actual processing. These points are labeled by START_PROFILING and END_PROFILING in the assembly code. The C compiler mostly rearranges the code, so positioning the labels already in the C code can yield to inaccurate measurements. In those cases the label positions in the generated assembly code are adjusted manually.

One of the features of DSP C compilers is their ability to do constant propagation as a part of the machine independent optimizations. In order to protect the benchmarked programs, especially the short ones in the HLL-kernels suite, against this optimization we have introduced a mechanism based on the `pin_down()` procedure. It represents the border for the constant propagation. If the compiler is even able to explore the contents of the `pin_down()` procedure, relocation to a separate file will help.

Every benchmark program is accompanied by a *measurement report* which is the basis for analysis and comparison. In the DSPstone methodology the format and contents of this report are specified. It entails the measurements, the listings of the measured code and all the facts regarding compilation and profiling which enable an exact reproduction of the measurements.

III.6. Evaluation

After collecting all the measurements of the benchmark programs, the next step is the evaluation of the results. The goal of the evaluation is to provide the user with informations about strong and weak points of the compilers and the joint compiler/processor system. Using the benchmark results he can decide whether the C compiler is the appropriate tool for his design process. Also, he is enabled to determine which part of the code has to rewritten in assembly or replaced by highly-optimized library functions.

Also, using the DSPstone results, the reasons for the low performance of the compilers can be identified and comparisons can be made. This could be of special interest for DSP compiler specialists. In the DSPstone project comparisons and ratings of various compilers and compiler/processor systems are of secondary importance only.

IV. Example

The DSPstone methodology has been applied on a set of five state-of-the-art DSP C compilers for fixed-point processors (Analog Devices 2101, AT&T 1610, Motorola 56001, NEC 77016 and TI 320C51). The decision to select this test suite was motivated by the fact that the fixed-point compilers are mostly newcomers to the market and that no clear answers about their usefulness exist. Up to the TI and NEC compilers, all others are ports of the GNU gcc compiler [12].

Although we tried to evaluate all compilers under the same benchmarks, different development states of the compilers have partitioned the test set into two subsets. In the first subset are the ADI, Motorola and TI compiler. These companies started releasing their compilers quite early, so the products are stable and well supported. Also, the underlying processors are for some time on the market which caused the assembly reference code for the most standard applications and DSP functions to be available.

In order to gain an insight into the ability and limitations of the compilers to support specific processor and language features an overview is presented on Table 1 [2].

feature compiler	AT&T 1610	AD 2101	Motorola 56001	NEC 77016	TI C51
first release in	1994	?	1990	1994	1988
benchmarked version	beta	5.1	1.11	beta	6.24
multiply-add	-	√	√	-	-
parallel-move/single	-	√	√	√	-
parallel-move/double	-	-	-	-	†
repeat-loop	-	†	√	√	-
do-loop	-	√	√	√	√
nested-loop	-	√	√	√	-
modulo addressing	-	-	-	√	-
bit-reversed addr.	†	-	-	-	-
pre/post inc./dec.	√	√	√	√	√
static frame alloc.	-	√	-	√	-
fractional arithmetic	-	-	-	√	-
inline assembly	√	√	√	√	√
"-" to C interface	√	√	√	√	√
function inlining	√	√	-	-	√
interrupts in C	-	-	-	√	√
√ supported; - not supported; † no hardware support;					

Table 1: Compiler Characteristics.

In the sequel we shall present some measurement results in order to verify the introduced methodology. For more details refer to the DSPstone final report [1]. The results and comments presented in the next subsection express only the views of the authors.

IV.1. Application Benchmarks

ADPCM Transcoder - CCITT Recommendation G.721

The ADPCM standard is one of the oldest speech coding standards which plays an important role even in newest designs (DECT). It is specified up to bit-accurate test sequences provided by the International Telecommunication Union (ITU), so differences in functionality are easily checked and removed. Because of the data dependent execution time we measured the performance of all programs on the first 32 samples of the ITU-CCITT test sequence `nrm.m`.

One of the reasons to include the ADPCM transcoder as a benchmark lies in the fact that it is the largest DSP application for which standard-complying, assembly versions for most targets exist. Unfortunately, not for all. We could not obtain the assembly versions for the NEC and AT&T processors, so the results are missing.

The ADPCM benchmark is characterized by a lot of bit-oriented computation which is a heavy task for a DSP C compilers. The ADPCM C code is written in a way which guarantees

[2] All the compilers undergo permanent revisions. For the features of the actual version contact the vendor.

high efficiency for all compilers in the same time retaining a readable and maintainable form. The reference programs for the ADI and the Motorola compiler can be freely obtained from their BBS and the TI reference code is licensed. Table 2 presents the compiler overhead of the ADPCM benchmark. It

	AD 2101	Motorola 56001	TI C51
$\Delta c[\%]$	698	510	555
$\Delta p[\%]$	284	51	7
$\Delta d[\%]$	383	175	301
$\Delta m[\%]$	302	70	30

Table 2: ADPCM Benchmark: Compiler Overhead.

is evident that the generated code has a very high overhead in execution time (over 500%) and as such is useful only for rapid prototyping and as a template for the development of the assembly code. The memory overhead is lower but still cannot be described as acceptable. The relatively low program memory overhead for some compilers is the consequence of full inlining in the assembly code and the use of procedures in the C code.

The introduced reference code methodology and the overhead measures show how well the compiler understands, matches and uses the underlying architecture. The overhead measures do not produce the information about the performance of the processor running the C code. In order to verify this important difference Table 3 presents the absolute performance of the compiled and the reference code (given in braces). The processor clock periods are taken from [13]. It is obvious that

	AD 2101-50ns	Motorola 56001-30ns	TI C51-25ns
$t_q(t_r)[\mu s]$	356(44)	521(85)	224(34)
$load[\%]$	285(36)	417(68)	179(27)
$c_q(c_r)$	7122(892)	20854(3414)	8947(1365)
$p_q(p_r)$	2410(628)	1852(1230)	2957(2934)
$d_q(d_r)$	662(137)	610(222)	1235(308)
$m_q(m_r)$	3072(765)	2462(1452)	4192(3242)

Table 3: ADPCM Benchmark: Absolute Performance.

between compiler overhead and the performance of the compiled code large differences exist. Both measures are useful for the user. In the DSPstone the overhead as well as absolute performance are reported.

What are the reasons for such a high overhead? The inability to use the specific hardware features of the processor is surely a very important factor, but not the only one. In the ADPCM benchmark the bit-manipulations have been a much heavier problem for the compiler than the parallel instructions. According to our observations the compilers waste a lot of time in highly inefficient data moves in the glue code between obviously independently compiled code fragments.

The ADPCM is an application with almost no standard processing blocks, so the time-critical code fragments cannot be simply taken from a library - they have to be coded manually. This happens very often in the domain of fixed-point algorithms and especially in standards. We have coded the time-critical MSB routine (computation of the most significant bit in the FMULT routine) in assembly for each processor and calculated the number of clock cycles needed. The results are presented in Table 4.

	AD 2101	Motorola 56001	TI C51
c_g (MSB in Assembly)	6322	19094	8211
improvement [%]	11	9	8
$\Delta c[\%]$	609	459	502

Table 4: Compiler Overhead with MSB in Assembly.

Although in our case only a small part of the code was rewritten in assembly and the performance improvement is not so high, mixed assembly-C coding is without any doubt the right way to obtain a trade-off between desk-time and run-time efficiency of the design. By our opinion, the need for mixed assembly-C programming will persist for a long time. This fact should not demoralize the compiler designers. They should further try to reduce the percentage of the hand-written assembly code in the program. The developments in the hardware technology are going to help them in their efforts.

Some suppliers of DSP equipment have recognized all the importance of the assembly libraries. However, the problem is solved only for standard coarse grain functions (FFT, DCT, LMS, etc.). In those cases the assembly to C context switching overhead is low compared to the gain obtained by assembly programming. For fine grain functions (e.g. bit-manipulation) inline-assembly macros are the best alternative. However, most compilers are not able to optimize beyond the inline-assembly delimiters and the overall improvement is low.

In the future the application suite of the DSPstone should be extended on a number of other standard applications in order to equally cover all DSP application domains.

IV.2. DSP-Kernel Benchmarks

The benchmarks of the DSP-kernels suite are:

- real updates
- complex updates
- matrix product
- convolution
- complex product
- IIR biquad section
- IIR filter
- FIR filter
- LMS filter
- FFT
- 2-D FIR filter

The reference assembly programs for the DSP-kernels suite belong in the mean-time to the standard equipment of every development package, so we had to write the equivalent C programs only. The functional equivalence was tested on test inputs which together with the appropriate outputs form a part of the benchmark suite. As an example in Table 5 the results for the convolution DSP-kernel ($y = \sum_{i=0}^{i=N-1} x_i * h_{N-i-1}$) are given. Almost

# taps = 16	AT&T 1610	AD 2101	Motorola 56001	NEC 77016	TI C51
$\Delta c[\%]$	1240	426	480	419	260
$\Delta p[\%]$	400	100	100	40	75
$\Delta d[\%]$	0	0	0	0	0
$\Delta m[\%]$	72	14	13	5	15

Table 5: Convolution Benchmark: Compiler Overhead.

all compilers have used the zero-overhead loop. Unfortunately, they are unable to use the parallel multiply-add instruction with parallel moves. The problem is in the explicit control over the memory banks. But even on those compilers which support this

C extension, the generated code stays the same.

IV.3. HLL-Kernel Benchmarks

The performance of the compiler on standard C constructs was benchmarked with the HLL-kernels suite consisting of the following benchmarks:

- CALL - C function call overhead
- FOR - `for` loop analysis
- NESTED-FOR - nested `for` loop analysis
- DOWHILE - `do while` loop analysis
- WHILE - `while` loop analysis
- FLOAT - `float` arithmetic performance
- INT - `int` arithmetic performance
- FRACT - fractional arithmetic performance
- LONGINT - `long int` arithmetic performance
- MADD - usage of multiply-add instruction
- PARALLEL - instruction parallelism analysis
- INDEXING - indexing vs. pointer addressing
- COMPACTION - effects of source code compaction

The motivation to include the suite of HLL kernels was twofold: to estimate the performance and to find out the forms of the C code which are best suited for the particular compiler. So, e.g. the FOR benchmark has 8 programs implementing various `for` differing in the automatic pre/post increment/decrement of the iteration variable. The goal was to find out those forms which are compiled to zero-overhead loops.

As an example the results for the CALL benchmark are presented in Table 6.

	AT&T 1610	AD 2101	Motorola 56001	NEC 77016	TI C51
Δc[%]	734	880	680	400	367
c_g	50	49	78	25	28

Table 6: CALL Benchmark: Compiler Overhead.

The CALL benchmark gives the context switching overhead which is introduced by C function calls. The benchmarked function has five integer arguments and returns their sum. Only those instructions introduced to perform the context switch have been measured. The results show that the compilers do a lot of housekeeping around the C function calls. Especially the GNU-based compilers have problems determining what is the minimum job which has to be done during a C call.

By our opinion reducing the context switching overhead is one of the points where HLL programming of DSPs differs from the programming of general-purpose computers and where DSP compiler improvements are necessary and feasible. The first step in the right direction is the static frame allocation for non-recursive procedures which is already implemented in some of the compilers.

V. Conclusions

A new, DSP-oriented benchmarking methodology is introduced. The DSPstone metric is based on the distance between the C code and the assembly reference code which is supposed to be the optimal one which can be written for the given functionality. The reference code metric gives the user the opportunity to directly judge about the compiler, processor and joint compiler/processor performance. The benchmarking programs are divided according to granularity into: application, DSP-kernel and HLL-kernel benchmarks. In order to guarantee the reproducibility of the results a detailed description of the measuring and reporting is specified.

As an example, the DSPstone benchmarking methodology is applied on a set of five state-of-the-art fixed-point DSP C compilers and processors. The introduced methodology gave a clear answer about the performance of the compilers under test. The results show that a lot of work has to be invested into fixed-point DSP compiler development in order to make them useful, not only for rapid prototyping, but also for production quality programming.

As the next step the DSPstone shall be applied on floating-point DSP compilers.

Acknowledgements

The DSPstone project was supported by Analog Devices, AT&T, Motorola, NEC and Texas Instruments by means of software and consultations. We would like to thank Manfred Christ (TI), Jeff Enderwick (Motorola), Tom Gentles (AT&T), Berthold Heck (NEC), Kevin Leary (ADI), George Mock (TI), Stephan Reitemeyer (NEC) and Craig Smilovitz (ADI) for their kind support.

VI. References

[1] V. Živojnović, J. Martínez Velarde, and C. Schläger, "DSPstone: A DSP-oriented benchmarking methodology," tech. rep., Aachen University of Technology, August, 1994.

[2] Analog Devices Inc., *Digital Signal Processing Applications Using the ADSP-2100 Family, Vol. I*, 1992.

[3] Motorola Inc., *DSP56000/DSP56001 User's Manual*, 1990.

[4] Motorola Inc., *DSP56156 User's Manual*, 1992.

[5] T. Instruments, "Considerations in choosing a high-performance floating-point dsp," tech. rep., Texas Instruments, Inc., 1994.

[6] P. Lapsley, J. Bier, and E. Lee, *Buyer's Guide to DSP Processors*. Berkeley Design Technology, Inc., 1994. pp. 495-634.

[7] E. Lee, "Programmable DSP architectures: Part I," *IEEE ASSP Magazine*, pp. 4–19, October 1988.

[8] T. Conte and W. Hwu, "Benchmark characterization," *IEEE Computer Magazine*, pp. 48–56, Jan. 1991.

[9] R. Weicker, "An overview of common benchmarks," *IEEE Computer Magazine*, pp. 65–75, Dec. 1990.

[10] W. Price, "A benchmark tutorial," *IEEE Micro Magazine*, pp. 28–43, Oct. 1989.

[11] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[12] R. Stallman, *Using and Porting GNU CC*. Free Software Foundation, Inc., 1990.

[13] "DSP directory," *EDN Magazine*, June 9 1994.