Aachen University of Technology
Integrated Systems for Signal Processing

DSPstone

A DSP-Oriented Benchmarking Methodology

Final Report

Aachen, August 1994

Contents

Abstract

This is the final report on the DSPstone project. The main goal of the project was the efficiency evaluation of the state-of-the-art DSP C compilers. The motivation was twofold. First, we wanted to give an answer to the question coming often from DSP users: *If I decide to use C instead of assembly programming, how large will be the speed/memory overhead?* As second, we have been interested in reasons for the relative inefficiency of the existing compilers, as well as possible ways to improve their performance.

In the DSPstone over 30 benchmark programs organized in three benchmark suites (Application, DSP-kernel and C-kernel) are defined. In order to measure the performance, evaluate and compare the results in a systematic way, a new, DSP-oriented benchmarking methodology is introduced. It is based on the reference-code method where the metric distance between the hand-written assembly code and the assembly code generated by the compiler is measured. This enables a decoupled evaluation of compiler and processor which is not possible using the standard computer benchmarking approaches.

The introduced methodology is applied on a set of five commercial DSP C compilers (Analog Devices 21xx, AT&T 16xx, Motorola 56xxx, NEC 770xx and TI 320C5x). The performance results are compared and presented.

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.

1. Introduction

Comparing performance of computers is rarely a dull event, especially when the designers are involved. Charges and countercharges fly across an electronic network; one is accused of underhanced tactics and the other of misleading statements. Since careers sometimes depend on the results of such performance comparisons, it is understandable that the truth is occasionally stretched. But more frequently discrepancies can be explained by differing assumptions or lack of information.

From "Computer Architecture: A Quantitative Approach"
by John Henessy and D. Patterson

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.

In the DSP community the predominant opinion is that the exclusive reason for the inefficiency of the compilers is their inability to use specific architectural features of the DSPs [1,2]. Already the first measurements we made have shown that for fixed-point DSP compilers the problems is much more complex. Detecting and describing the real reasons for DSP compiler inefficiency and suggesting improvements was the second task of this project.

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 report 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.

2. Background

A Computer Benchmarking

In the computer literature benchmarking is a well treated subject [3,4,5,6]. Benchmarks like SPECS, Dhrystone, Whetstone and Linpack are a widely accepted mean for comparison of computer systems. The comparison is based on the execution speed of HLL programs, so the results depend not only on the computational speed of the hardware, but also on the efficiency of the compiler. In the past the benchmarking specialists treated this effect only marginally. The benchmarking reports entailed the name and version of the compiler, as well as the applied compiler flags. The differences in code efficiency between various compilers have been relatively small and therefore regarded as measurement noise. The introduction of optimizing compilers has made the benchmarking more difficult. The results obtained by applying various compilers or by using compiler flags can be very different, so the assumption that the compiler is a negliglible disturbing factor is not valid anymore.

Suprisingly, benchmarking of compilers on the basis of the efficiency of the generated code has been treated only occasionally. In [7] the ratio between execution speed of optimized and non-optimized code as a measure of compiler quality was used. The drawback of this approach is the absence of a reliable reference point. Producing inefficient code when optimization is off improves the benchmarking results.

B Benchmarking of DSP Hardware and Software

In the past benchmarking of digital signal processing hardware was conducted almost only by the chip vendors themselves [8,9,10,11]. 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 [12] 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 [1] 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.

3. 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 [3,4,5,6].

A 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).

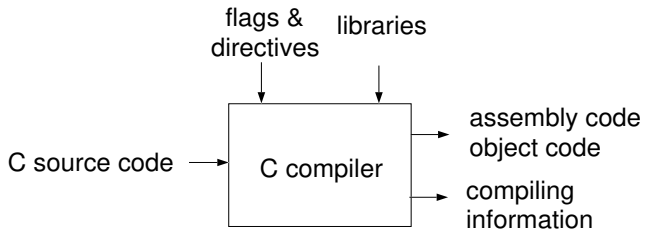

Figure 1: The C Compiler as a Black-Box Processing Unit.

He controls the input and can judge about the behavior of the unit by observing the corresponding output - primarily the generated assembly code.

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.

B 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.

C 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.

D 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 [6] the execution time can be expanded into:

$$t = \frac{instructions}{program} \frac{average\,clock\,cycles}{instruction} \frac{seconds}{clock\,cycle} \tag{1}$$

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.

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

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

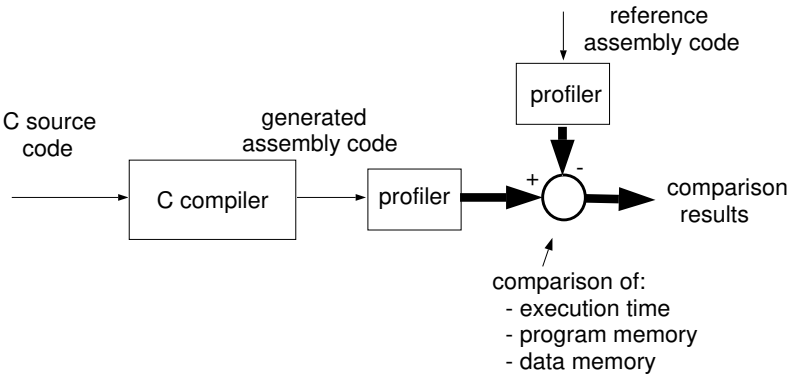

Figure 2: C Compiler Performance Evaluation

can be applied 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.

Additionally, we report the processor load l which is the ratio between the number of clock-cycles $c(T)$ needed by the program in some given time period T (mostly sampling period) and the number of clock-cycles which a processor with cycle period τ can produce during the period T.

$$l = \frac{c(T)}{T/\tau}\ [\%] \tag{7}$$

E.g. for the ADPCM transcoder the sampling period is $T = 0.125ms$. If the transcoder needs 892 cycles per sample and the clock-cycle period is 50ns, the load will be 892/(125000/50) which gives 36%.

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.

E 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 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.

On the next page the outlook of the cover page for an example benchmark program is given. The description of the items is given in italic. It has to be stressed that the benchmark measurement report contains only raw measurements data. The complete benchmark software in a computer readable form will be delivered to interested users.

Benchmark: int *(name of the benchmark)*
Version: 0.1 *(version of the benchmark)*
Benchmark suite: C - kernel *(the benchmark belongs to the suite)*
Description: evaluation of the integer arithmetic *(short description of the benchmark)*
Target: Motorola DSP56001 *(description of the target processor)*
Memory: program - internal, no WS
data - internal, no WS
(description of the memory layout used)
Compiler: g56k (v1.11) *(version of the compiler)*
Compiling command: *(how was the file compiled and linked)*

```
g56k -alo -S -O -D__DSP56000__ $*.c
g56k -alo -asm '-occ -L' $*.asm -o $*.cld
```

Reference code: none *(which assembly reference code was used - origin and functionality)*
Profiling procedure: from the list file *(how was the profiling done)*
History: 24-1-94 - profiling (Meyer) *(history log of the benchmarking)*

Benchmarking results: *(time and memory profiling data)*

code	#clock-cycles	#instructions	#operations	time@33MHz [μs]
int.c	120	60	80	3.64

code - *name of the file*
#clock-cycles - *number of clock cycles needed for the execution of the program*
#instructions - *number of instructions executed in the program*
#operations - *number of operations executed in the program*
#time@33MHz - *time in μs needed to execute the program on a processor with given clock*

code	#words(prog)	#instructions	#words(data)	#words(p+d)
int.c	100	80	40	140

code - *name of the file*
#words(prog) - *number of processor words needed for the program*
#instructions - *number of instructions put in program memory*
#words(data) - *number of words needed for data*
#words(program+data) - *number of words needed for program and data together*

Remarks: *(Here are given the remarks regarding the code and the profiling procedure.)*

F 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.

4. An Example: Benchmarking of Fixed-Point C Compilers

The DSPstone methodology has been applied to 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 [13].

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.

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

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 [14]. The results and comments presented in the next subsection express only the views of the authors.

A Application Benchmarks

1. 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 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 is evident that the generated code has a very high overhead in

	AD 2101	Motorola 56001	TI C51
Δc[%]	698	510	555
Δp[%]	284	51	7
Δd[%]	383	175	301
Δm[%]	302	70	30

Table 2: ADPCM Benchmark: Compiler Overhead.

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 [15]. It is obvious that 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

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

Table 3: ADPCM Benchmark: Absolute Performance.

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.

B DSP-Kernel Benchmarks

The benchmarks of the DSP-kernels suite are:

- real updates (REAL_UPDATES, N_REAL_UPDATES)
- complex updates (COMPLEX_UPDATES, N_COMPLEX_UPDATES)
- matrix product (DOT_PRODUCT, MATRIX_1X3, MATRIX)
- convolution (CONVOLUTION)
- finite impulse response filter (FIR, FIR2D)
- infinite impulse response filter (IIR_BIQUAD_1, IIR_BIQUAD_N)
- least mean square filter (LMS)
- fast Fourier transform (FFT_INPUT_SCALED, FFT_STAGE_SCALED)

1. REAL_UPDATE Benchmark

Functional Description:

The REAL_UPDATE benchmark implements the operation $d = c + a * b$, where a,b,c and d are real numbers.

Assembly Reference Code:

The assembly reference codes of most targets use the dual memory architecture, so a and b are read in parallel. After multiplication and addition of the result to c, the final result d is stored in memory. The assembly references could be programmed using 4 to 6 instructions.

The *AT&T-1610* assembly code makes an efficient use of the 3 stage DAU pipeline. In four consecutive instructions, the processor reads the data from memory, performs the multiplication and adds the product to the accumulator. Multiply and/or accumulate statements are permitted in an instruction with x,y fetches, but multiply and/or accumulate operations will be performed on data loaded in the previous instruction [16]. The last instruction stores the updated value d in memory. The benchmark is completed in five instructions.

The *ADI-2101* target first loads the variable data c into the register `mr0`. The values for a and b are loaded from program and data memory in parallel `mx1` and `my1`. Next the processor performs the `mac` operation in one cycle. The fourth and last instruction stores the result d in memory.

The *Motorola-56001* reference code needs one bit left shift of the summand c before the `mac` instruction is performed. Parallel to the shift operation, the values for a and b are loaded from memory into registers `x1` and `y1`. After the multiplication and addition, the result d is one bit right shifted and stored in memory. Five instructions are needed to perform the reference code. The reason for the necessary shifts is the arithmetic representations used by the processor during the `mac` operation [17].

The *NEC-77016* reference code consists of 6 instructions. Data is loaded from X and Y memory in the first two instructions. The register containing the value c is shifted one bit left before and one bit right after the `mac` operation is performed. The reason lies also on the different data representation of the processor. The last instruction stores the value d in memory.

The *TI-C51* reference code implements the real update with a separate multiplication and addition operation. First the value for the summand c is loaded into the accumulator. The multiplication is performed in two instructions. The first loads the multiplicand in the `TREG` register, the second performs the multiplication, pointing to the multiplier and stores the product in the `PREG` register [18]. The addition to the accumulator is performed in the fourth instruction. Last instruction stores the updated value d in memory.

Generated Code:

The developed C program performs the real update through pointer operations declared as `register` variables. The update is programmed in one C statement. The pointer declaration as `register` advises the compiler that the variable in question will be heavily used [**?**]. The compiler should place the pointers in machine registers, which result in smaller and faster programs. The compilers generate code that vary from 6 to 14 instructions.

The *AT&T-1610* compiler translates the C code into a non-parallelized assembly code. The variables are pointed by the Y data space pointers `r1` and `r3`, and the auxiliary BMU registers `ar0 = reg58` and `ar1 = reg59`. Extra instructions are needed to get the content of the auxiliary BMU registers into the `y` register, at the multiplication and at the addition operation. Code expands significantly to 14 instructions, consuming 22 cycles. The AT&T generated code needs more than twice as much cycles as the presented optimal solution.

The *ADI-2101* compiler produces a compact code, consisting of three data reads, one multiplication, one addition and one store instruction. No instruction is performed parallel. Only one memory bank is used. The compiler separates the optimal `mac` instruction in one multiplication and one addition. The generated code needs 6 clock cycles to perform the benchmark, two cycles more than the optimal assembly reference.

The *Motorola-56001* compiler also uses one memory bank. One parallel instruction (accumulator one bit left shift) is performed parallel to a data read. The `mac` instruction is used, but the processor needs to shift one bit the summand c before and after the `mac` operation, in order to keep a consistent data representation. In this case, the compiler clock cycles overhead is 33.3%, the lowest one at this benchmark.

The *NEC-77016* compiler translates the code using only one memory bank. The compiler separates the multiply-add operation in two instructions. Between these operations, data is left shifted one bit . The compiler places a `nop` operation after writing the data pointer register, in order to avoid bus conflicts. The `nop` operation could be replaced by a more useful operation, eg. the `r0` register assignment of value c, which appears at the begin of the benchmark. The translated program needs twice cycles as the reference program to be performed.

The *TI-C51* compiler environment recognizes the update operation and generates six assembly instructions, separating the multiplication from the addition operation, as in the reference program. The result is loaded into the accumulator and added to the summand c. The result is stored finally in memory. The C code is generated compactly in only one instruction more than the presented assembly reference, consuming in all 7 clock cycles and resulting in the lowest compiler overhead in program memory requirements.

	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$t_g(t_r)[\mu s]$	0.55(0.175)	0.3(0.2)	0.48(0.36)	0.36(0.18)	0.175(0.125)
$c_g(c_r)$	22(7)	6(4)	16(12)	12(6)	7(5)
$p_g(p_r)$	14(5)	6(4)	8(6)	12(6)	6(5)
$d_g(d_r)$	4(4)	4(4)	4(4)	4(4)	4(4)
$m_g(m_r)$	18(9)	10(8)	12(10)	16(10)	10(9)

Table 5: REAL_UPDATE Benchmark: Absolute Performance

	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$\Delta c[\%]$	214	50	33	100	40
$\Delta p[\%]$	180	50	33	100	20
$\Delta d[\%]$	0	0	0	0	0
$\Delta m[\%]$	100	25	20	60	11

Table 6: REAL_UPDATE Benchmark: Compiler Overhead

2. N_REAL_UPDATES Benchmark

Functional Description:

The N_REAL_UPDATES benchmark updates an array of real data with the operation $d_i = c_i + a_i * b_i, 1 \leq i \leq N$.

Assembly Reference Code:

All assembly reference codes make use of zero overhead loops, in order to get optimal speed results. References are highly parallelized and most improved with `mac` instructions.

The *AT&T-1610* reference occupies 10 instructions. The reference is programmed to be optimized for execution speed, making use if the 3 stage DAU pipeline. The first two instructions load the values for the variables a_1, b_1 and c_1.

The next instructions are included in the zero-overhead loop and show the update kernel. The instructions within the loop body,

```
000c:720f     91            do 15{
000d:f845     92                                  p=x*y  y=*r1++      x=*pt++
000e:30a0     93                        a0=a0+p
000f:e0c9     94                        *r2++=a0l
0010:3ccd     95                        a0l=*r3++
              96             }
```

perform first the multiplication operation and parallel loads the `x` and `y` registers with the next a_i and b_i values. Addition, storing of the result and loading of the next c_i value are done in the next instructions. The loop is repeated $N - 1$ times, where N is the length of the updated array. Once the loop is finished, last multiplication, addition and storing are performed. The update is performed in 85 cycles.

The *ADI-2101* reference code is programmed in the analog way. The data for the first update a_1, b_1 and c_1 is loaded before the zero-overhead loop begins.Within the `do` loop and parallel to the `mac` instruction, the next data for a_i and b_i is loaded, all in one instruction. Pointers are automatically incremented by one. The result is next stored in memory. The loop is performed $N - 1$ times. Last `mac` and storing is done once the loop has finished. The loop body contents only three instructions so the assembly reference code is performed in only 51 clock cycles.

The *Motorola-56001* reference code reads the a_i and b_i data parallel to the shifting of the accumulator, prior to the `mac` instruction. The loop is programmed overhead-free with a `do` instruction. One bit shift operations must be placed before and after the `mac` instruction, as in the previous benchmark. The program in coded in 8 instructions, seven of them as the loop body, which is repeated N times.

The *NEC-77016* assembly reference code is structured similarly. The values for a_1 and b_1 are

read from the data and program memory parallel before the zero-overhead `LOOP` begins. The accumulator value is read and shifted one bit left within the loop, before the `mac` operation is performed. Parallel to the multiply-add, the values for the next a_i and b_i are read. Finally the result is right-shifted one bit and stored in memory.

The *TI-C51* assembly code makes use of a `RPTB` zero overhead loop construction. The whole program consists of 9 instructions, the loop body is compactly coded in 5 instructions. It is very much the same code as the previous benchmark REAL_UPDATE, separating the multiplication in two instructions. Here, the auxiliary registers pointing to the data values a_i, b_i, c_i and d_i are incremented at each instruction by one.

Generated Code:

The C program perform the update operation within a `for` loop that includes only one statement. Update management is realized through pointers, declared as `register` variables. Pointers are updated within the same C statement with post-increment operators.

The *AT&T-1610* compiler does not generate a `do` loop instruction. The loop body is controlled via an expensive `if-then` instruction. No instruction is generated parallel. The compiler uses a narrow range of the registers offered by the processor. These facts expand code to 41 words at 913 cycles execution time. The compiler overhead of this target presents the highest values of all, both at the cycles and at the words overhead.

The *ADI-2101* target translates the update kernel in a compact assembly program. One data read is generated parallel to the `mac` operation in the zero overhead do loop. The code within the loop is translated compactly in two more assembler instructions than the assembly reference. The compiler program code overhead is therefore the lowest of all.

The *Motorola-56001* compiler generates also a very compact program from the high-level language, and one parallel instruction is included in the translated code. As in the previous Update Kernel, the compiler places a one bit shift before and after the `mac` operation. The code within the loop body corresponds to the one at the Real Update kernel, with the addition of the register increments for the pointer update, which are performed as post-increment operators to the `move` instructions. The loop body needs two instructions more than the assembly reference, achieving a low compiler overhead towards program code, as in the previous target. The compiler overhead relating to the cycles count is at this target the lowest one.

The *NEC-77016* compiler implements a zero-overhead `LOOP`. The loop body consists of 9 instructions, with two instructions containing parallel operations. The compiler separates here the optimal `mac` instruction in two operations, one multiplication and one addition, shifting the multiplication result one bit before performing the addition.The translated code consumes two instructions more than the optimal reference, so the compiler overhead towards program words is as low as in the previous target. The compiler overhead relating to the cycles count is of same magnitude as in the previous target.

The *TI-C51* compiler uses a zero-overhead do loop, but no `mac` instruction is generated. The compiler uses indirect addressing extensively. The loop body consists of 9 instructions,

because of the separated multiplication and addition operations, 4 more than the assembly reference code. For each multiplication, the processor needs to load first the `TREG` with the multiplicand. The `MPY` instruction multiplies this value with the data memory value it is pointing to. The result is stored in the `PREG` register. The compiler overhead towards program words raises to 78 %, because of the loop body translation.

#of updates = 16	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$t_g(t_r)[\mu s]$	22.825(2.125)	6.55(2.55)	7.939(6)	4.42(3)	4.6(2.1)
$c_g(c_r)$	913(85)	131(51)	262(198)	146(99)	184(84)
$p_g(p_r)$	41(10)	11(9)	10(8)	10(8)	16(9)
$d_g(d_r)$	64(64)	64(64)	64(64)	64(64)	64(64)
$m_g(m_r)$	105(74)	75(73)	74(72)	74(72)	80(73)

Table 7: N_REAL_UPDATES Benchmark: Absolute Performance

#of updates = 16	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$\Delta c[\%]$	974	157	32	47	119
$\Delta p[\%]$	310	22	25	25	78
$\Delta d[\%]$	0	0	0	0	0
$\Delta m[\%]$	42	3	3	3	10

Table 8: N_REAL_UPDATES Benchmark: Compiler Overhead

3. COMPLEX_UPDATE Benchmark

Functional Description:

The COMPLEX_UPDATE benchmark implements the operation $d = c + a * b$, where all variables are complex. The d variable is updated with the result of the addition of summand c and product $a*b$. Four multiplications, three additions, one subtraction, and two assignments are part of the kernel, as

$$d_r = c_r + a_r * b_r - a_i * b_i$$
$$d_i = c_i + a_r * b_i + a_i * b_r$$

Assembly Reference Code:

Most assembly references use the dual memory program and data memory distribution. The values for the variables a and b (real and/or imaginary data) may be then read parallel. Due to the several multiplications, the optimal code reads the data parallel to the multiplication of the previous loaded values.

The *AT&T-1610* assembly code makes use of the 3 stage DAU pipeline, as in the previous kernel. Coefficient a is stored in data memory, coefficient b in program memory. The assembly code concatenates the multiplications and additions in order to fill the pipeline best. Data fetches are performed with pointer addressing. The pointer update is realized at the same operation, at zero cost. The assembler permits the post-increment of the address register `pt` with the value of register `i`. If the register `i` is negative, it yields to a post-decrement operation [16]. The assembler code performs the update in 15 cycles, due to the extensive use of the pipeline.

The *ADI-2101* assembly code performs the complex update operation in only 9 cycles. The Analog Devices assembly code loads the MAC registers `mx0`, `mx1`,`my0` and `my1` with the real and imaginary values of the multipliers. These registers are loaded once and are reused within the benchmark code. The `MAC` instruction may act as a multiply-add-and-accumulate or multiply-subtract-and-accumulate operation.

The *Motorola-56001* reference code needs the left-shifting of the summand c before the correspondent `mac` instruction is performed. After the multiplication and addition, the result is right shifted one bit and stored in memory. The reason is the different arithmetic representations used by the processor [17]. The assembly reference performs data reads parallel to the shift instructions and to the first `mac` instruction. The four data ALU input registers `x0`,`x1`,`y0` and `y1` are loaded with the real and imaginary a and b values, in order to reuse them in the following `mac` instructions. The processor permits to combine the ALU input registers `yi` with `xi` at the `mac` instruction. A "`-`" sign option is used to negate the specified product prior to accumulation, so the multiply-add–and-accumulate becomes a multiply-subtract-and-accumulate. Code is programmed in 12 program words, consuming 24 cycles.

The *NEC-77016* reference code is programmed in 13 instructions. Registers holding the a and b values are reused in order to maximize speed. The processor permits a double read

parallel to a `mac` operation, as well as a multiply-subtract-and accumulate operation. Data is shifted one bit before and after the `mac` instruction.

The *TI-C51* reference code implements a separate multiplication and addition operation. The assembly reference uses the indirect addressing modes of the processor. Most instructions permit to define the next auxiliary register pointer `ARP` without any speed or size costs. Data can be handled efficiently through the `ARP` pointer. Auxiliary register `AR` pointers are incremented or decremented within the same instruction at zero cost. The TI C51 assembly is programmed in 16 instructions, notably more than previous targets. The reason is that this target does not have a dedicated multiply-accumulate hardware when using one memory bank. The C51 requires the use of both the multiplier and the ALU to perform a complete multiply/accumulate operation [18].

Generated Code:

The high-level language version of the COMPLEX_UPDATE benchmark consists of four multiplications, three additions, one subtraction and two assignments. Four arrays hold the necessary data, each of them belonging to the variables a, b, c and d. Arrays are declared as `static`. The arrays are accessed by pointers which are declared as `register` variables, in order to let the compiler create best possible code.

The *AT&T-1610* compiler generates a non-parallelized code. The use of the auxiliary BMU registers `ar0` = `reg58` and `ar1` = `reg59` needs extra instructions to perform data move. The postincrement and postdecrement C operators are translated in extra instructions, expanding the code notably. The simple C instructions extend to 93 assembly instructions, consuming 155 cycles, making this target the one with greatest compiler overheads of all.

The *ADI-2101* environment produces a very compact code, consisting of only 23 assembly instructions. The compiler produces two parallel instructions, but no `mac` operation is performed. The compiler uses the register range offered by the processor broadly. Multiplication and addition are separated in two instructions.

The *Motorola-56001* compiler makes use of the `mac` instruction, producing one parallel instruction. Common to other update kernels, the compiler generates one bit left-shift before and one bit right-shift instruction after each `mac`. Especially remarkable is the unnecessary generation of following consecutive instructions after the `mac` operations.

```
...
166    P:0027 2000D2  [2 -  90]    mac     +x0,y0,a
167    P:0028 200022  [2 -  92]    asr     a
168    P:0029 210E00  [2 -  94]    move              a0,a
169    P:002A 21C800  [2 -  96]    move              a,a0
170    P:002B 5C6132  [2 -  98]    asl     a                       a1,y:(r1)
...
```

The same results werre computed without the `move a0,a` and the `move a,a0` instructions. The same applies to the `asr a` and `asl a` instructions. Compiling the generated assembly

program *without* the named instructions produces the same results. The compiler version used does not include a context sensitive optimizer to enhance the generated code.

The *NEC-77016* compiler needs 38 instructions to translate the COMPLEX_UPDATE benchmark. Five instructions are generated containing parallel operations. No `mac` operation is generated, the compiler always separates multiplication and addition/subtraction operations. Three `nop` operations are generated to avoid bus conflicts. These could be easily omitted by an efficient register scope analysis.

The *TI-C51* compiler does not use any `mac` operation and separates the multiplication in two assembly instructions, as seen in previous programs. The addition or subtraction respectively of the the values c_r and c_i are performed with the instructions `APAC` and `SPAC`. Data is accessed by indirect addressing. The compiler translates all four C statements separately, without checking if the variables calculated are for temporary use or if they must be stored in memory. This fact extends code to 31 program memory words. The memory overhead is the lowest at this target.

	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$t_g(t_r)[\mu s]$	3.875(0.375)	1.15(0.45)	2.04(0.72)	1.1(0.39)	0.95(0.4)
$c_g(c_r)$	155(15)	23(9)	68(24)	38(13)	38(16)
$p_g(p_r)$	93(10)	23(9)	34(12)	38(13)	31(16)
$d_g(d_r)$	8(8)	8(8)	8(8)	8(8)	8(8)
$m_g(m_r)$	101(18)	31(17)	42(20)	44(21)	39(24)

Table 9: COMPLEX_UPDATE Benchmark: Absolute Performance

	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$\Delta c[\%]$	933	156	183	192	138
$\Delta p[\%]$	830	156	183	192	94
$\Delta d[\%]$	0	0	0	0	0
$\Delta m[\%]$	461	82	110	110	63

Table 10: COMPLEX_UPDATE Benchmark: Compiler Overhead

4. N_COMPLEX_UPDATES Benchmark

Description:

The N_COMPLEX_UPDATES benchmark updates an array of data with the operation $d_i = c_i + a_i * b_i, 1 \leq i \leq N$ where all variables are complex.

Assembly Reference Code:

All assembly reference programs implement a zero-overhead loop with highly parallelized instructions.

The *AT&T-1610* reference code has 11 instructions. The reference is programmed to be optimized for execution speed, making extensive use if the 3 stage DAU pipeline. The code kernel is controlled by a zero overhead `do` instruction. First, the values for the real parts of a_1 and b_1 are loaded parallely in the registers `y` and `x`. The next instructions are included in the zero-overhead loop. The value for the real part of c is first loaded into the accumulator. Then, the multiplication of the real parts of a and b is performed parallel to the loading of the imaginary parts of a and b into the registers `y` and `x`. Next, both registers are multiplied and the result is subtracted from the accumulator. Parallel to these instructions, the registers `y` and `x` are further loaded with the appropriate data to calculate the imaginary part of d. The reference is highly parallelized. X and Y-pointers are updated optimally at zero cost.

The *ADI-2101* reference code is programmed compactly. The Analog Devices assembly permits to load the MAC registers `mx0`, `mx1`,`my0` and `my1` with the real and imaginary values of the variables. These registers are loaded once and reused within the loop at the `mac` operations. The `MAC` instruction may act as a multiply-add-and-accumulate or multiply-subtract-and-accumulate. The data for the multiply operator `mx1` and `my1`, corresponding to the real values of the a_i and b_i, are loaded before the `mac` instruction. Parallel to the `mac` instruction, the next data for `mx0` and `my0`, corresponding to the imaginary values of the a_i and b_i are loaded. Pointers are automatically incremented by one. The code consists of 11 instructions and is performed in 146 cycles.

The *Motorola-560001* reference code is a high parallelized and optimized assembly program. The reference performs data reads parallel to the shift instructions and to the first `mac` instruction. The four data ALU input registers `x0`,`x1`,`y0` and `y1` are loaded with the real and imaginary parts of the a and b values, in order to reuse them in the following multiply-add instructions. The assembly permits to combine the ALU input registers `yi` with `xi` at the `mac` instruction. A "`-`" sign option is used to negate the specified product prior to accumulation, so the multiply-add–and-accumulate become a multiply-subtract-and-accumulate. Due to the necessary shifts, the *Motorola-56001* benchmark takes 14 words of memory and 390 cycles to be completed.

The *NEC-77016* reference code puts the real and imaginary parts of the a and b values in several registers to reused them with the `LOOP` structure. The reference code takes 14 words of memory, where 12 instructions form the body of the loop. Right and left shift instructions must be performed to keep the adequate representation of the managed data.

The *TI-C51* reference makes use of the indirect addressing modes of the processor. Most instructions permit defining the next auxiliary register pointer `ARP` without any speed or size costs. Data can be handled efficiently by the `ARP` pointer. Auxiliary register `AR` pointers are incremented or decremented within the same instruction at zero cost. Multiplications are splitted in two separate instructions, as in the previous Update Benchmarks. The benchmark performs four multiplications, three additions, one subtraction and two assignments in 17 instructions, taking 276 cycles to be performed.

Generated Code:

The C program performs the update operation within a `for` loop. The high-level language versions of the COMPLEX_UPDATE benchmark consists of four multiplications, three additions, one subtraction and two assignments. Four arrays hold the necessary data, each of them containing the data for the variables a_i, b_i, c_i and d_i. Arrays are declared as `static`. Pointers are declared as `register` variables, in order to let the compiler create best possible code.

The *AT&T-1610* compiler generates a non-parallelized code. The simple C statements expand to 130 assembly instructions, that states a compiler overhead towards programs words of more than 1000%. No zero-overhead loop is translated. Instead, an expensive `if-then` construct is generated. Analog, the compiler overhead relative to cycle count arrives at this target a maximum value, far distanced from the rest of the benchmarked compilers.

The *ADI-2101* environment produces code consisting of 43 assembly instructions. The compiler produces two parallel instructions, but no `mac` operation is performed. Multiplication and addition are separated. Only the `mx1` and `my1` registers are used. The compiler does not use the registers `mx0` or `my0` at any time. A better register handling within the loop would improve code performance considerably.

The *Motorola-560001* compiler makes use of the `mac` instruction, producing three parallel instructions. The loop is translated optimally in to a zero-overhead `do` loop. Common to other update benchmarks, the compiler generates a one bit left-shift before and a one bit right-shift instruction after each `mac`. Especially remarkable is the unnecessary generation of `move` operations, as in the COMPLEX_UPDATE benchmark. Nevertheless, the *Motorola-56001* compiler creates a compact code in 35 instructions. The compiler sets also an unnecessary `nop` instruction at the end of the loop. This instruction could be easily omitted by an more efficient code distribution, saving two cycles per loop stage.

The *NEC-77016* compiler translates the benchmark in 43 instructions. Each C statement is generated separately at assembly level. No zero-overhead `LOOP` is generated at this target. The loop structure is controlled by a `if-jmp` instruction. Only one parallel instruction is generated. Like the previous kernels, the compiler does not include any `mac` operation. Pointer updates are performed as postmodification operations at zero-cost. The compiler uses also several `nop` operations, that could be omitted, as we stated in the previously UPDATE benchmarks.

The *TI-C51* compiler creates a compact code in 38 instructions. The compiler translates the loop in a zero-overhead `RPTB` instruction. The loop body takes 31 instructions. As in the

other update kernels, the C51 compiler needs two instructions to complete a multiply operation. Like in the COMPLEX_UPDATE benchmark, the compiler translated all C statements separately and does not check if the variables could be interpreted as temporary ones. The generated code could be optimized if the real and imaginary values of d were stored only twice per loop, instead of four times after each assignment. The results could be kept for most of the time in registers. The *TI-C51* environment delivers best results for the compiler overhead relative to memory usage and cycle count.

#of updates = 16	AT&T 1610-25ns	ADI 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$t_g(t_r)[\mu s]$	83.225(4.575)	32.15(7.3)	32.86(11.7)	20.64(6.33)	15(6.9)
$c_g(c_r)$	3329(183)	643(146)	1062(390)	688(211)	600(276)
$p_g(p_r)$	130(11)	43(11)	35(14)	43(15)	38(21)
$d_g(d_r)$	128(128)	128(128)	128(128)	128(128)	128(128)
$m_g(m_r)$	139(258)	171(139)	163(142)	171(143)	166(149)

Table 11: N_COMPLEX_UPDATES Benchmark: Absolute Performance

#of updates = 16	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$\Delta c[\%]$	1719	340	172	226	117
$\Delta p[\%]$	1082	291	150	187	81
$\Delta d[\%]$	0	0	0	0	0
$\Delta m[\%]$	86	23	15	20	11

Table 12: N_COMPLEX_UPDATES Benchmark: Compiler Overhead

5. DOT_PRODUCT Benchmark

Functional Description:

The DOT_PRODUCT benchmark computes the product of two vectors a and b, as

$$c = a * b = \begin{pmatrix} a_1 & a_2 \end{pmatrix} * \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = a_1 * b_1 + a_2 * b_2$$

The output c is a scalar.

Assembly Reference Code:

All reference assembly programs use parallelized instructions to read and compute the dot product. Vector a is stored in data memory and vector b in program memory in order to permit parallel data reading from both memory areas.

The *AT&T-1610* reference reads the first elements a_1 and b_1 of vector a and b. Multiplication is performed in the next step, while next elements a_2 and b_2 are loaded from memory. Next follows the second multiplication and the assignment of the first product to the accumulator. These operations are performed parallel in the one instruction. Last instructions add the second product to the accumulator and store the result in memory. The DOT_PRODUCT benchmark is realized optimal in 5 assembly language instructions.

The *ADI-2101* assembly program also reads first the vector elements a_1 and b_1. In the next operation, the accumulator is directly filled with the multiplication result. Parallel, the next vector data a_2 and b_2 is loaded into the registers `mx1` and `my1`. Next instruction performs the multiplication of `mx1` and `my1` and its addition to the accumulator. Last instruction stores the result c in memory.

The *Motorola-56001* reference makes use of the long memory data move operation [17]. The operation `L:` moves one 48-bit long word operand from/to X and Y memory. Two data ALU registers are concatenated to form the 48-bit long-word operand. This allows efficient moving of both double-precision (high:low) and e.g. complex(real:imaginary) data from/to one effective address in $L(X:Y)$ memory. The same effective is used for both the X and Y memory spaces; thus, only one effective address is required. The registers `x0` and `x1` are loaded with the first vector elements a_1 and b_1 in the first instruction. The second instruction performs the multiplication and loads the next elements a_2 and b_2 from memory with the `L:` operator. These are multiplied in the next instruction and added to the accumulator with a `mac` operation. Data should be right-shifted before being stored in memory.

The *NEC-77016* assembly reference code is not available.

The *TI-C51* reference performs the dot product operation within a zero-overhead loop. In the loop, the `TREG` register is first loaded with the a_1 element, and the `MPY` operand points to the b_1 element. Registers are updated at zero cost and point then to the next element at second loop stage. Data is multiplied and added to the accumulator. Last instructions outside the loop store the result c.

Generated Code:

The C code consists of a `for` loop, with one C program statement in its body. Data is accessed through pointers. The variable c accumulates the result of the multiplication of the vectors a and b. Pointers are updated in the instruction with postmodification operations.

The *AT&T-1610* generated code shows several deficits. No parallel instruction is created. As in the previous kernels, no zero-overhead loop is generated, but an expensive `if-then` construction. The compiler does not place the pointer update operations parallel to the ALU operations, slowing the performance. The translated C code takes 33 instructions in assembly level. The compiler overhead reaches the maximum values at the *AT&T-1610* target for memory and cycles consumption.

The *ADI-2101* generates a very compact code in only 8 instructions. Within the zero-overhead `do` loop, data is read from memory, first multiplied and then added to the accumulator. The accumulator value is stored within the loop. A code improvement strategy could place the variable storing outside the loop, saving one instruction.

The *Motorola-56001* creates also a compact code within a `do` loop. The compiler places two operations parallel, namely accumulator one bit shift-left and data read from memory. The code is generated in 10 instructions, 9 of them belonging to the loop.

The *NEC-77016* compiler results are not available.

The *TI-C51* compacts the C code in 11 instructions. The compiler sets an optimal `RPTB` structure with 6 instructions. The C translation is very close to the presented assembly reference program, because of the simple kernel structure and the compiler possibilities to compact C code. Therefore, the *TI-C51* compiler overhead arrives minimal values for this target.

	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$t_g(t_r)[\mu s]$	2.275(0.2)	0.788(0.242)	0.667(0.303)	n/a	0.475(0.325)
$c_g(c_r)$	91(8)	13(4)	22(10)	n/a	19(13)
$p_g(p_r)$	33(5)	8(4)	10(5)	n/a	11(10)
$d_g(d_r)$	5(5)	5(5)	5(5)	n/a	5(5)
$m_g(m_r)$	38(10)	13(9)	15(10)	n/a	16(15)

Table 13: DOT_PRODUCT Benchmark: Absolute Performance

	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$\Delta c[\%]$	1038	225	183	n/a	46
$\Delta p[\%]$	560	100	183	n/a	10
$\Delta d[\%]$	0	0	0	n/a	0
$\Delta m[\%]$	280	44	110	n/a	7

Table 14: DOT_PRODUCT: Compiler Overhead

6. MATRIX_1X3 Benchmark

Functional Description:

The MATRIX_1X3 benchmark computes the matrix product of a 3×3 matrix H and a 3×1 vector x

$$y = H * x = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} h_{11} * x_1 + h_{12} * x_2 + h_{13} * x_3 \\ h_{21} * x_1 + h_{22} * x_2 + h_{23} * x_3 \\ h_{31} * x_1 + h_{32} * x_2 + h_{33} * x_3 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Assembly Reference Code:

The assembly references implement high parallel instructions. An effective use of the Harvard architecture permits double data reading and parallel `mpy` or `mac` instructions.

The *AT&T-1610* reference code makes use of the zero-overhead `do` loops and the cache `redo` instructions. Cache instructions implement zero-overhead loops. The use of cache loops conserve program memory and speed execution time. The `do` instruction treats the specified NI instructions as a structure to be repeated n times. The `redo` instruction treats the previous NI instructions as another loop to be executed n times. The `redo` instruction performs the last structure repeated by `do` and stored at the cache memory, only by coding the instruction `redo n`. Up to 15 instructions can be performed in the `do` or `redo` loops. Both cache instructions use one program memory location [16]. Code is performed in 34 cycles and occupies 13 program words locations.

The *ADI-2101* assembly reference program is structured in one zero-overhead loop, containing one multiplication and two further multiply-add instructions. Data is read parallel to the arithmetic operations. The `i4` pointer is defined at each loop stage to the beginning of the x matrix. The presented reference code performs the kernel in 20 cycles, taking 8 program words of memory.

The *Motorola-56001* reference performs the matrix operation without implementing a loop structure. Instructions are coded straight-forward. The extensive use of registers and accumulators and the implementation of a circular buffer for the x coefficients lead to broad parallel instruction use. The accumulator result is only right-shifted before it is stored, saving unnecessary logical operations. The code is simulated in 28 cycles.

The *NEC-77016* reference code is not available.

The *TI-C51* reference program is build with two zero-overhead nested loops. The inner loop (row counter) is controlled by a `RPTB` instruction, repeating only the next operation 3 times. The calculation of the vector elements y_i is perform in the only instruction of the inner loop. It is based on the instruction `MADS`, *Multiply and Accumulate with Dynamic Addressing*, which multiplies a data memory value by a program memory value [18]. It also adds the previous product to the accumulator. Dynamic Addressing is performed through the register `BMAR`, which points at program beginning to the first x vector element. The outer loop acts as column counter for the matrix H. The reference code shows the wide possibilities of an

efficient assembly programming for the C5x environment. The reference code is performed in 40 cycles.

Generated Code:

The C program computes the result matrix with two nested loops. The first loop acts as column counter for the matrix H, the second loop as row counter for the vector x. The vector elements y_i are computed in one C statement. Data is accessed through `register` pointers ; these are updated between the loops. The inner loop contains the same instruction as in the DOT_PRODUCT benchmark. This benchmark shows how compilers translate a more complicated, nested structure with assignments and arithmetic operations.

The *AT&T-1610* does not generate any zero-overhead loops. The compiler does not set any parallel operations. Even the pointer update operations require one separate instruction, expanding the code. The multiply-add-update operation at the inner loop of the C program is translated in 33 instructions. The benchmark processing time is more than 15 times slower than the presented assembly reference.

The *ADI-2101* compiler translates the C code into two zero-overhead `do` loops. The inner loop is translated in 5 instructions, as in the previous benchmark. The compiler creates a very compact code which can be performed in only 63 cycles. Pointer update could be easily improved. The compiler defines the register `m0` after the inner loop. It is used to update the pointer. The register definition could take place before entering the loops. The value for `m0` does not change within the benchmark. Therefore, the *ADI-2101* compiler has the lowest compiler overhead towards memory consumption.

The *Motorola-56001* translation contains also two zero-overhead `do` loops. The compiler delivers a compact code of 14 instructions. The generated code is performed in 186 cycles. The inner loop control itself, the `do` instruction, takes 6 cycles to be performed. The inner loop is perform 9 times and consists of 8 instructions, because of the necessary shift operations, increasing the cycles consumption to a compiler overhead greater than 500 %.

The *NEC-77016* compiler results are not available.

The *TI-C51* compiler cannot create a nested loop with two zero-overhead loop structures. Only the inner loop is generated with a zero-overhead `RPTB` instruction. There is only one set of block repeat registers, so multiple block repeats cannot be nested without saving the content of the outside block or using `if-then` structures. The compiler uses a schematic strategy for executing nested loops, placing a `RPTB` for only the innermost loop and using a `BGEZ` instruction for the outer one [18]. The inner loop is translated in 5 instructions, the whole program occupies 27 words, because of the nested loop translation inefficiency. Still, the compiler overhead relative to cycles consumption arrives at the *TI-C51* target the minimal value, with 173 %.

	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$t_g(t_r)[\mu s]$	13.825(0.85)	3.15(1)	5.58(0.84)	n/a	2.725(1)
$c_g(c_r)$	553(34)	63(20)	186(28)	n/a	109(40)
$p_g(p_r)$	58(13)	15(8)	15(14)	n/a	27(15)
$d_g(d_r)$	15(15)	15(15)	15(15)	n/a	15(15)
$m_g(m_r)$	73(28)	30(23)	30(29)	n/a	42(30)

Table 15: MATRIX_1X3 Benchmark: Absolute Performance

	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$\Delta c[\%]$	1526	215	564	n/a	173
$\Delta p[\%]$	346	63	88	n/a	118
$\Delta d[\%]$	0	0	0	n/a	0
$\Delta m[\%]$	161	22	30	n/a	40

Table 16: MATRIX_1X3 Benchmark: Compiler Overhead

7. MATRIX Benchmark

Functional Description:

The MATRIX benchmark computes the product of two matrices $A_{k\times l}$ and $B_{l\times m}$, as

$$C = A * B = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1l} \\ a_{21} & a_{22} & \dots & a_{2l} \\ \vdots & \vdots & \ddots & \vdots \\ a_{k1} & a_{k2} & \dots & a_{kl} \end{pmatrix} * \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{l1} & b_{l2} & \dots & b_{lm} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1m} \\ c_{21} & c_{22} & \dots & c_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ c_{k1} & c_{k2} & \dots & c_{km} \end{pmatrix}$$

The matrices A and B are matrices of arbitrary dimension. The only restriction is that the l dimension of the matrices must be greater than 1. The result of the kernel operation is a matrix C of dimension $k \times m$.

Assembly Reference Code:

The assembly reference code for all targets consist of three nested loops. The outermost loop controls the m counter, the middle loop the k counter and the innermost loop calculates the output matrix elements c_{km}.

The *AT&T-1610* assembly reference code is not available.

The *ADI-2101* assembler reference code is written in 13 instructions. The innermost loop consists of only one pipelined instruction to get optimal results. It performs a `mac` operation and reads the data for the next multiply-add. Last multiplication is performed outside the loop.

The *Motorola-56001* reference code is also build up in three nested zero-overhead loops. The outer ones are `do` loops, the innermost is a cheaper `rep` loop, because its body loop consists of only one instruction. The structure is similar as in the *AD-2101* assembler reference, with addition of a the one-bit shift-right operation of the accumulator before storing the calculated matrix element c_{km}. Code is written in 15 instructions.

The *NEC-77016* assembly reference code is not available.

The *TI-C51* reference program include two zero-overhead loops. The innermost loop has only one instruction, using the operation `MADS`, as in the MATRIX_1X3 benchmark. The middle loop is controlled by a `RPTB` structure. The third loop, the external one, is controlled by a `BANZD` delayed branch instruction. This outer loop is optimized with the use of *delayed*

conditional branches [18]. In the delayed operation of branches, the two-instruction words following the delayed instruction are executed while the instructions at and following the branch address are being fetched - therefore , giving an effective two-cycle branch instead of flushing the pipeline. If the instruction following the delayed branch is two-word instruction, only that instruction is executed before the branch is taken.

Generated Code:

The profiled C program implement a speed optimized code. The program consists of three nested loops that act as m, k and l counters. The inner loop contains the arithmetic operations as a sum-of-products. This statement is common to the DOT_PRODUCT and MATRIX_1X3 benchmarks. The first multiply-add is performed before the loop, as an accumulator assignment of the multiplication of the first two matrix elements, so that no clear operation on the accumulator has to be done. The inner loop performs then $l-2$ multiply-adds. The last multiply-add is performed after the inner loop, and it includes the pointer increment to the next output element. At all, l multiply-add operations are performed in the innermost loop. Results are given for $m = k = l = 10$.

The *AT&T-1610* compiler results are not available.

The *ADI-2101* compiler generates three zero-overhead `do` loop structures. The innermost loop is translated in 6 instructions, one of them with 2 parallel operations. But this code part performance could be enhanced. The *ADI-2101* compiler defines in the innermost loop the value for the `m0` register, even though it is not changed within the whole code. These unnecessary instructions in nested loops increase execution time. Saving the named instruction in the innermost loop would spare $m*k*(l-2)$ cycles, in our example, 800 cycles, more than 12 % of the clock cycle time, and its compiler overhead values would improve.

The *Motorola-56001* compiler delivers an assembly program with three nested `do` loops. The innermost loop is translated in 8 instructions as in the previous benchmark MATRIX_1X3. The consequences of the `asl` and `asr` shifts and `move` instructions before and after the `mac` operation in the innermost loop increase the execution time dramatically. Each instruction is performed at this stage $m*k*(l-2)$ times, with at least 2 clock cycles per instruction.

The *NEC-77016* compiler results are not available.

The *TI-C51* compiler cannot generate more than one zero-overhead loop at nested loop structures. The compiler places the zero-overhead loop at the innermost stage, because it will be the most processed part of the nested code. The remaining loop are translated with suboptimal `if-then` constructions. The innermost code, as in the previous benchmark, is translated in only 5 instructions. The compiler uses indirect addressing extensively, applying four different auxiliary registers in the translated program. The compiler generates a compact code, whose memory and clock cycle overheads arrives at the *TI-C51* target the best results.

$k = l = m = 10$	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$t_g(t_r)[\mu s]$	n/a	324.8(77.1)	518.82(105.78)	n/a	165.95(48.075)
$c_g(c_r)$	n/a	6496(1542)	17294(3526)	n/a	6638(1923)
$p_g(p_r)$	n/a	37(13)	53(15)	n/a	48(22)
$d_g(d_r)$	n/a	300(300)	300(300)	n/a	300(300)
$m_g(m_r)$	n/a	337(313)	353(315)	n/a	348(322)

Table 17: MATRIX Benchmark: Absolute Performance

$k = l = m = 10$	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$\Delta c[\%]$	n/a	321	390	n/a	245
$\Delta p[\%]$	n/a	185	253	n/a	118
$\Delta d[\%]$	n/a	0	0	n/a	0
$\Delta m[\%]$	n/a	8	12	n/a	8

Table 18: MATRIX Benchmark: Compiler Overhead

8. CONVOLUTION Benchmark

Functional Description:

The CONVOLUTION function implements the general purpose and often required operation in digital filters,

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

as a sum-of-products *without* a state update.

Assembly Reference Code:

All assembly reference codes implement a convolution with optimized parallel operations. Multiplication and addition to the previous accumulate are performed parallel to the data reads.

All targets allow programming highly-optimized assembly compact programs which can take advantage of the chip resources, like double memory management (data and program memory), single-instructions multiply-add operations and parallelized data reads.

The *AT&T-1610* reference consists primarily of a `do` loop with one `mac` instruction.. Before entering the loop, the variables `x` and `y` hold the values for $x[0]$ and $h[N-1]$ and the `p` register is equal zero. Within the loop, one instruction permit increase the accumulator by the last value of `p`,calculate a new `p` value by multiplication of the `x` and `y` register, and load the next x and h values into the `x` and `y` registers. The convolution is realized within the `do` loop, which is performed N times. Finally, the result y is stored in memory.

The *ADI-2101*, the *Motorola-56001* and the *NEC-77016* assembly code also concentrate the convolution operation in only one instruction as the content of a zero-overhead loop structure. These targets perform the double data read parallel to the `mac` operation. The multiplication is performed with the values read in the previous period, but the addition to the accumulator is performed with the multiplication result calculated in the same instruction.

The *TI-C51* reference code takes advantage of the pipelined `MAC` instruction. The `MAC` instruction multiplies a data memory value by a program memory value. It also adds the previous product to the accumulator. The actual multiplication result is stored in the `PREG` register [18]. When the `MAC` instruction is repeated, the program memory address is incremented automatically by one during its operation. In this manner, automatic pointer update is performed. The `MAC` instruction needs without being pipelined at least 3 cycles to perform the operation. Once the RPT pipeline is started, it becomes a single-cycle instruction.

Generated Code:

The C code consists of a loop construct containing a sum-of-products operation. The data array x and the coefficient array h are accessed by `register` pointers, being updated in the

same C statement. Most targets compile the code into a zero-overhead do-loop block. Loop body includes a multiplication, an addition and an assignment operation. Only one memory bank is used by the compilers, so no double parallel data read can be performed. All values are given for a convolution filter length of 16.

The *AT&T-1610* target does not build a zero-overhead loop, and uses an expensive `if-then` conditional construction. The compiler does not place any parallel multiplication to a data read or an ALU operation. Likewise, the compiler could easily place two of the four register increment operations parallel to the multiplication operation and to the addition operation, saving three cycles per tap. A poor use of the `YAAU` (Y Space Address Arithmetic Unit) registers increases the instruction and cycle count within the loop. The generated code in the loop body is 27 assembly instructions long and the compiler overhead reaches at the *AT&T-1610* target the lowest values.

The *ADI-2101* compiler does not implement the `mac` operation, but generates a compact code. Multiplication and addition are resolved in two separated assembly instructions. State value and the coefficient are read consecutively, since only one memory bank is used. Even the code version supporting C Language Extensions [20], like separate program and data memory specifications at the High-Language level, performed no double parallel reads. The loop body is here 6 instructions long.

Due to the signed multiplication of state and coefficient values and the simultaneous addition to the cumulative output with the `mac` instruction at the *Motorola-56001*, the compiler shifts the cumulative value *each* time before and after the multiply-add operation [21]. Only one memory bank is used by the compiler, so no double parallel reads can be performed. The compiler generates two parallel operations. It places also a unnecessary `nop` instruction as last operation in the loop body. This fact extends the loop body to 7 instructions, reaching same compiler overhead values as for the previous target.

The *NEC-77016* target generates a zero-overhead loop instruction. The loop body is translates in 5 assembly instructions, making use of one memory bank. The addition and multiplication operations are separated in two instructions. The result of the multiplication is left-shifted in a separate instruction in order to preserve a correct addition result. The *NEC-77016* compiler produces a compact code, using only 7 words of program memory for the whole convolution function, even though no `mac` operation is generated. The compiler overhead towards memory consumption reach the lowest value at this target, together with the *TI-C51* environment.

The convolution operation is performed with indirectly indexed arithmetic and move operations at the *TI-C51* target. The loop body is compiled very compactly in 5 instructions. Since the `mac` instruction needs one operand in data memory and one in program memory, the TI compiler uses here the data memory and the separates the operation in a multiply and a addition assembly instruction. The linker loads the kernel program into ON_CHIP memory and data into internal ram memory in order to provide best results. These facts state the *TI-C51* compiler the lowest overhead relative to the cycle count and the same compiler overhead relative to program memory consumption, as at the previous target.

# of taps = 16	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$t_g(t_r)[\mu s]$	18.08(1.35)	5(0.95)	6.96(1.2)	2.49(0.6)	2.25(0.625)
$c_g(c_r)$	723(54)	100(19)	232(40)	83(20)	90(25)
$p_g(p_r)$	35(7)	10(5)	10(5)	7(4)	14(8)
$d_g(d_r)$	32(32)	32(32)	32(32)	32(32)	32(32)
$m_g(m_r)$	67(39)	42(37)	42(37)	39(36)	46(40)

Table 19: CONVOLUTION Benchmark: Absolute Performance

# of taps = 16	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$\Delta c[\%]$	2577	426	426	315	260
$\Delta p[\%]$	400	100	100	75	75
$\Delta d[\%]$	0	0	0	0	0
$\Delta m[\%]$	72	14	14	8.33	15

Table 20: CONVOLUTION Benchmark: Compiler Overhead

9. FIR Benchmark

Functional Description:

Digital Filtering is one of the central applications using DSPs. The presented linear-phase FIR Kernel represents a typical digital filter structure. The FIR benchmark multiplies an array of state variables x by an array of coefficients h and accumulates the result in the output variable y, as

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

State variables x_i are kept in a tapped delay line of length N that holds the input values from $x(k)$ to $x(k-N+1)$. Each time a value inputs the filter, the last $N-1$ values are shifted within the delay line x. The result y is the weighted sum of the multiplications of the delay line content and the coefficients h.

Assembly Reference Code:

All targets deliver high performance assembly codes. Most reference codes use modulo arithmetic programming techniques. The modulo arithmetic addressing mode causes the address register value remain within an address range. The modulo arithmetic permits simulate the state variable delay line with a circular buffer, without shifting data throughout memory.

The Harvard architecture allows coefficients to be available in both program and data memory. Simultaneous fetching of two operands is necessary to make efficient use of the architecture. Reference programs are quite similar structured as those for the convolution benchmark. Most facts we determined at the previous benchmark are found also here. High performance is also reached applying parallelized instructions. Therefore, data is stored in both the program (for h_i coefficients) and data memory banks (for x_i state variables) and can be read parallel to a third operation.

The *AT&T-1610* target offers one circular buffer. Only +1 is allowed as postmodify value. The `rb` and `re` registers contain the begin and end address of the buffer [16]. When the pointer address register equals the address in `re`, the address in `rb` is placed in the pointer register at next clock. The presented assembly reference code is speed optimized. The code takes full advantage of the 3 stage DAU pipeline within the zero-overhead `do` loop, and extends to 6 words.

The *ADI-2101* target can define up to four circular buffers in each of both independent Data Address Generators. The reference code realizes the fir filtering within a zero-overhead `do` loop. The body loop contents one instruction, that performs the `mac` operation and the coefficient and data reads. Code occupies 6 words in program memory.

The *Motorola-56001* processor also permits define up to eight circular buffers, four for each of two Address Generation Units [17]. The presented assembly code uses a low cost zero-overhead instruction named `rep` that repeats only the next instruction N times, instead of

defining a more expensive loop block with the instruction `do`. Data is read from program and data memory parallel to a `mac` operation. Code extends to 7 instructions.

The *NEC-77016* assembly reference code uses a zero-overhead `REP` loop and a `mac` instruction to perform the filtering. Parallel to the `mac`, the data for the next multiply-add is loaded into the registers `r1` and `r3`. The code implements a circular buffer in $XRAM$ to perform the FIR state delay line. The coefficients of the filter are stored in $YRAM$. Code is implemented in 6 instructions.

The *TI-C51* reference code uses a `MACD` instruction. The `MACD` instruction multiplies a data memory value (holding the state value) by a program memory value (holding the coefficient value). It also adds the previous product to the accumulator [18]. `MACD` functions in the same manner as the `MAC` at the convolution kernel, with the addition of the data move for on-chip RAM blocks. When the `MACD` instruction is repeated, the program memory address is incremented by one during its operation. When used with zero-overhead repeat instructions, `MACD` becomes a single-cycle instruction one the RPT pipeline is started. Although no circular buffer is used in the assembly reference code, the 'C5x supports two concurrent circular buffers [18]. The reference assembly code is realized in 6 instructions.

Generated Code:

The C version of the fir digital filter defines two `static` arrays, filled with the state and coefficient values. Convolution and data shifting is performed within a loop. The loop itself and its content is the critical C code of the benchmark, and therefore we take special attention at the translation of these code parts. All values are given for a fir filter length of 16.

The *AT&T-1610* compiler does not place any parallel instruction in the generated code. The loop is controlled via an `if-then` structure. Code is translated in 60 words. Loop body is performed in 59 clock-cycles. The compiler overhead takes at the *AT&T-1610* target the highest values. The distance from assembly to C is more than 3000% relative to clock cycles and 500% relative to words program consumption.

The *ADI-2101* target puts in the translated code a zero-overhead loop and 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 in the convolution computation the product $x_i * h_{N-i}$. This modify register value should be defined out 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 the code performance.

The *Motorola-56001* generates a relative high 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. A higher code compact level can be easily reached, putting the register postdecrementing operations after the last use of the register itself within the loop. The compiler places also an entirely unless `nop` instruction as last loop operation, consuming extra unnecessary cycles. Nevertheless, the relative high code parallelization states in consequence a low compiler overhead values for memory consumption.

The *NEC-77016* 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 *TI-C51* compiler translates the critical part of the C code, the loop, into a zero-overhead compact structure of 7 instructions. The fir digital filtering is performed via separated multiplication, addition and storing operation. The compiler makes an extensive use of the indirect addressing possibilities of the processor, providing a compact assembly translation. The translation of the critical C code part in 7 instructions allows the code to be performed faster than at the other targets, specially if this part should be performed N times. Therefore, the *TI-C51* environment reaches the lowest compiler overhead value relative to the cycle count.

# of taps = 16	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$t_g(t_r)$	22.85(0.775)	8.75(1)	9.72(1.5)	3.51(0.63)	3.1(0.625)
$c_g(c_r)$	914(29)	175(20)	324(50)	117(21)	124(25)
$p_g(p_r)$	60(10)	21(6)	21(8)	18(6)	24(8)
$d_g(d_r)$	32(32)	32(32)	32(32)	32(32)	32(32)
$m_g(m_r)$	42(92)	54(38)	53(40)	50(38)	56(40)

Table 21: FIR Benchmark: Absolute Performance

# of taps = 16	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$\Delta c[\%]$	3052	775	548	457	396
$\Delta p[\%]$	500	250	163	200	200
$\Delta d[\%]$	0	0	0	0	0
$\Delta m[\%]$	119	42	33	32	40

Table 22: FIR Benchmark: Compiler Overhead

10. FIR2DIM Benchmark

Functional Description:

The FIR2DIM benchmark perform the convolution of an input matrix $A_{k\times k}$ by a coefficient matrix $C_{3\times 3}$. To provide boundary conditions for the filtering, the input matrix is surrounded by a set of zeros, such that the matrix to filter is actually of dimension $A_{(k+2)\times(k+2)}$. This is necessary in order to preserve the boundary array values. The output filtered matrix is of dimension $Y_{k\times k}$. As a realistic benchmark, the input array may be an image object to be filtered with a coefficient mask $C_{3\times 3}$.

$$Y = A * C = \begin{pmatrix} 0 & 0 & \dots & 0 & 0 \\ 0 & a_{11} & \dots & a_{1k} & 0 \\ 0 & a_{21} & \dots & a_{2k} & 0 \\ 0 & \vdots & \ddots & \vdots & 0 \\ 0 & a_{k1} & \dots & a_{kk} & 0 \\ 0 & 0 & \dots & 0 & 0 \end{pmatrix} * \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix} = \begin{pmatrix} y_{11} & \dots & y_{1k} \\ y_{21} & \dots & y_{2k} \\ \vdots & \ddots & \vdots \\ y_{k1} & \dots & y_{kk} \end{pmatrix}$$

No values updates are performed within the benchmark.

Assembly Reference Code:

The reference code provide high parallelized instructions. The input array is stored in data memory and the coefficients are stored in program memory, to permit double data reads to the arithmetic operations. Input and output matrices are stored in row major storage. Two nested loops act as row and column counters for the convolution calculation.

The *AT&T-1610* reference code provides two nested loops. Only the inner loop is zero-overhead, because the processor cannot nest `do` loops. The outer loop is controlled by the counter `c0`. The *AT&T-1610* offers two 8-bit signed counters. Each time one of the counter flags is tested by a control conditional instruction, like an `if-then instruction`, the counter is incremented, saving the increment operation. In the inner loop are performed the multiply-accumulate operations and the storing of the result in memory. X-memory pointer adjustment requires at this benchmark several instructions, because it can be only performed with the register `i`, which is used within the inner loop.

The *ADI-2101* assembly reference code is structured in two nested `do` loops. The outer loop acts as row counter i, the inner loop acts as column counter j. The inner loop contains the convolution operation itself. It consists of nine multiply-add operations. Each of the `mac` is performed parallel to two parallel reads. At the end of the loop, the calculated matrix element y_{ij} is stored in memory.

The *Motorola-56001* reference code is constructed on two nested loops as row and column counters, as in the previous target. The inner loop contains also nine multiply-add operations, each of them with a double read performed parallel. The calculated output matrix element

y_{ij} is right-shifted one bit before it is stored in memory. Pointers adjustment is performed between the inner and the outer loop.

The *NEC-77016* reference code is build up on two nested loops, as in the previous targets, the outer acts as the row counter, the inner as the column counter. The content of the inner loop performs the two dimensional convolution. It consists of one `mpy` and eight `mac` instructions, with two data reads parallel to the arithmetic operation. The mask coefficients are stored in X memory, the input and output data in Y memory. The calculated result must right-shifted one bit, in order to maintain the consistent data representation. The last instruction within the inner loop stores the result in memory. Pointer updates are performed with post-decrement operations and with the addition of index registers `dpN##` [**?**].

The *TI-C51* reference code is structured in three loop levels. The outer loop acts as row counter. The middle loop acts as a column counter. This level contains three zero-overhead `RPT` loops, and each of them calculate the partial multiplications of the input matrix elements with the elements of a coefficient column. The reference code makes use of the instruction `MAC` to perform the convolution operation. This instruction is pipelined within a `RPT` loop, to get a single-cycle instruction and get optimal results at assembly level [18].

Generated Code:

The C program performs the Two-Dimensional FIR2DIM Convolution as a three-loop level construction. The matrix data is stored in `static` arrays and it is accessed by `register` pointers. The outer loop acts as a row counter i, the middle loop acts as a column counter j. Within this level, three loops perform the convolution operation itself. These loops contain a `mac` operation and each of them are performed 3 times. The pointers are updated with the actual values for i and j. Special attention will be paid to the translation of the inner loops, as the critical part of the code, which is performed at the magnitude $3 * k^2$, where $k \times k$ is the dimension if the input array. All values are given for a 4×4 input matrix A.

The *AT&T-1610* compiler generates code without any `do` loop. The critical C code at the inner loops is translated in *each* 46 instructions, though they only perform a simple `mac` operation. No instruction contains more than one operation, that is no parallel operations are performed in the code. Therefore, the clock cycles compiler overhead is more than 4600 % in relation to the assembly reference code.

The *ADI-2101* compiler translates the code and generates all loops to zero-overhead `do` loops. The critical C code in the innermost loops is translated suboptimal. The first loop is generated in 5 instructions at assembly level, but the second and third, though all loops contain the equivalent C statement, are translated each in 12 instructions. The compiler overhead relative to clock cycles increases near to 900% ; the generated code is distanced from the assembly reference code.

The *Motorola-56001* compiler generates also three zero-overhead `do` loop levels. The content of each inner loop is translated in 8 instructions. The compiler always places a `mac` operation and the necessary one bit shift-left and shift-right. The environment reaches low compiler overhead values, in the same magnitude as for the *TI-C51* target.

The *NEC-77016* compiler provides a nested three level `do` loop. The compiler places shift operations to keep the consistent arithmetic representation, and these take extra instructions. No `mac` instructions are generated. Instead, the compiler separated the multiply and the accumulate operation in two instructions. The three `mpy` instructions of the code contain parallel assignment operations. The three inner loops contain therefore each 8 instructions.

The *TI-C51* compiler generates no nested zero-overhead loops. The innermost loop is in this case translated with a zero-overhead instruction. Here, the compiler places a `RPTB` loop for the three inner loops. The loop content is translated in 7 instructions for the both first loops and in 5 instructions for the third one. The adequate translation of the critical C code parts, let the *TI-C51* environment reach the lowest compiler overhead values for cycle and program memory consumption.

$k = 4$	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$t_g(t_r)[\mu s]$	334.475(7)	94.4(9.55)	54.84(15.09)	47.76(6.09)	42.275(12.125)
$c_g(c_r)$	13379(280)	1898(191)	1828(503)	1592(203)	1691(485)
$p_g(p_r)$	245 (24)	73(20)	76(26)	62(18)	97(35)
$d_g(d_r)$	61(61)	61(61)	61(61)	61(61)	61(61)
$m_g(m_r)$	306(85)	134(81)	137(84)	123(79)	158(96)

Table 23: FIR2DIM Benchmark: Absolute Performance

$k = 4$	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$\Delta c[\%]$	4678	894	263	684	249
$\Delta p[\%]$	921	265	192	244	177
$\Delta d[\%]$	0	0	0	0	0
$\Delta m[\%]$	260	65	63	56	65

Table 24: FIR2DIM Benchmark: Compiler Overhead

11. IIR_ONE_BIQUAD Benchmark

Functional Description:

The IIR_ONE_BIQUAD Benchmark performs the filtering of input values through a biquad IIR section. The benchmark implements a second oreder filer with a minimum of delay elements. The difference equation relating the output to the input is :

$$y(n) = a1 * y(n-1) + a2 * y(n-2) + b0 * x(n) + b1 * x(n-1) + b2 * x(n-2),$$

where $a1$, $a2$, $b0$, $b1$ and $b2$ are the filter biquad coefficients.

Assembly Reference Code:

The *AT&T-1610* assembly reference code distributes the states values in Y data memory and the coefficients in X program memory. A wide use of the 3 stage DAU pipeline permits code the benchmark in 11 instructions.

The *ADI-2101* assembly reference code performs the biquad filtering in 12 instructions. Data is stored in data and program memory, in order to permit double data reads. The `mpy` and `mac` operations are performed parallel to the double data reads. A circular buffer in program memory contains the biquad coefficients. A wide use of parallel instructions minimizes code requirements and speed up the assembly reference code.

The *Motorola-56001* assembly reference code makes also use of the dual memory distribution, locatinf the coefficients in Y memory and the filter states values in X memory. The code realizes double data reads parallel to the `mac` operations In order to preserve the consistent data representation, several one bit shift operations are included within the code. The assembly reference code occupies 13 instructions.

The *NEC-77016* assembly reference code is not available.

The *TI-C51* assembly reference code includes for the IIR_ONE_BIQUAD benchmark in 15 instructions. The reference code places states and coefficient values in the data memory bank. Note the use of `LTD` (*load TREG, accumulate previous product and move data*), `MPY` (*multiply*) and `MPYA` (*multiply and accumulate previous product*) instructions. This reference code makes use of one memory bank.

Generated Code:

The C program declares the state and coefficient variables as `static`. It consists of 7 C statements that perform the biquad calculation, five multiplications, two additions and two subtractions, as well as the state shifts, in order to perform the upon given difference equation.

The *AT&T-1610* compiler translates the benchmark in 88 instructions that occupy 93 program words. As in the previous benchmarks, no parallel instruction is generated. For each of multiplication operation, the compiler loads first the x and y registers separately. Each x

and y load needs up to six instructions. These loads are performed through the accumulator `a1` and one Y register, and increase the translated code so that the compiler overhead reach the highest values.

The *ADI-2101* compiler generates a compact code in 19 instructions, only 7 more than the reference code. The compiler can set 5 instructions with each two parallel operations. Therefore, the overheads towards clock cycle and program memory consumption reach a low value with 58%.

The *Motorola-56001* generates a 23 instruction long code that occupies 33 words of program memory. Six instructions include parallel operations and each multiplication is translated into a `mpy` or `mac` instruction. The compiler sets also, as in previous benchmarks, the necessary shift operations to preserve the consistent data representation. The compiler overhead values rise therefore to 177% and 154% towards clock cycles and program memory consumption respectively.

The *NEC-77016* assembly reference code is not available.

The *TI-C51* compiler translates the code into 26 instructions. The compiler is able to recognize the temporary and non-temporary variables of the C code. The compiler overhead remain in relative low values and reach the lowest overhead towards program memory usage, as consequence of the compactness of the code generation.

	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$t_g(t_r)[\mu s]$	4.175(0.45)	0.95(0.6)	2.16(0.78)	n/a	0.7 (0.375)
$c_g(c_r)$	167 (18)	19(12)	72(26)	n/a	28(15)
$p_g(p_r)$	93(11)	19(12)	33(13)	n/a	22(15)
$d_g(d_r)$	9(9)	9(9)	9(9)	n/a	9(9)
$m_g(m_r)$	102(20)	28(21)	42 (22)	n/a	31(24)

Table 25: IIR_ONE_BIQUAD Benchmark: Absolute Performance

	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$\Delta c[\%]$	828	58	177	n/a	87
$\Delta p[\%]$	745	58	154	n/a	47
$\Delta d[\%]$	0	0	0	n/a	0
$\Delta m[\%]$	410	33	91	n/a	29

Table 26: IIR_ONE_BIQUAD Benchmark: Compiler Overhead

12. IIR_N_BIQUADS Benchmark

Functional Description:

Assembly Reference Code:

Generated Code:

$k = 4$	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$t_g(t_r)[\mu s]$	23.425(1.075)	7.6(1.85)	9.36(3.12)	n/a	4.725(1.425)
$c_g(c_r)$	937(43)	152(37)	312(104)	n/a	189(57)
$p_g(p_r)$	90(10)	41(13)	42(15)	n/a	50(21)
$d_g(d_r)$	30(30)	30(30)	30(30)	n/a	30(30)
$m_g(m_r)$	118(38)	71(43)	72(45)	n/a	80(51)

Table 27: IIR_N_BIQUADS Benchmark: Absolute Performance

$k = 4$	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$\Delta c[\%]$	2072	311	200	n/a	231
$\Delta p[\%]$	800	215	180	n/a	138
$\Delta d[\%]$	0	0	0	n/a	0
$\Delta m[\%]$	211	65	60	n/a	57

Table 28: IIR_N_BIQUADS Benchmark: Compiler Overhead

13. LMS Benchmark

Functional Description:

The Least-Mean-Square (LMS) adaptive filter performs a finite-impulse-response filtering, where the tap coefficients are adapted iteratively, with the update equation

$$w_i(k+1) = w_i(k) + \mu * err(k) * x_i(n)$$

where $w_i(k+1)$ is the value of the i-th coefficient at sample time $k+1$, $w_i(k)$ is the value of the i-th coefficient at sample time k, $x_i(k)$ is the value of the i-th filter state at time k, μ is the gain of the error loop, $err(k)$ is the error, as difference between the filter output and the desired signal at time k and n is the length of the adaptive filter.

Assembly Reference Code:

The target assembly references implement a lms filtering with optimized parallel operations, use of hardware loops and circular buffer addressing. As in the convolution and fir kernel, multiplication and addition to the previous accumulate are performed parallel to the data reads. Special attention is paid to the parameter dependent parts, two loop structures to calculate the fir output and to update the coefficients.

The *AT&T-1610* assembly reference code stores coefficients and state values in Y memory. No data to be updated (states or coefficients) can be stored in X memory, because no instruction can alter the X data values. Optimal pipelining is thereby disturbed. State variable update is performed by compound addressing. Compound addressing is a memory read/write operation using only one memory register [16]. Both loops, the fir loop and the coefficient update loop, are performed with zero-overhead `do` instructions. The fir loop contains two instructions in the loop body, the coefficient loop contains five instructions to perform the update operation. The assembly reference code uses 22 words of program memory.

The *ADI-2101* reference code implements zero-overhead `do` loops for calculating the fir output and the coefficient update. The assembly reference makes use of modulo arithmetic to handle the state values and the filter coefficient optimal. The code occupies 20 program memory words.

The *Motorola-56001* assembly code implements two loop structures at zero-overhead. The first loop structure realizes the fir filtering as in the previous kernel. The second loop realizes the coefficient update. The code can be placed in 24 program memory words.

The *NEC-77016* assembly reference code is not available.

The *TI-C51* reference code is organized as for the previous environments. The fir filtering is realized via the `RPTB` and the `MACD` instruction, as in the fir kernel. The coefficient update is performed within three instructions. The kernel occupies 38 words of program memory for 27 instructions.

Generated Code:

Critical C code is the one within the two loop blocks. This segments increase linearly with the filter tap number. The first loop realizes the fir filtering. The second one adapts the filter coefficients. All benchmarked targets translate both loops zero-overheaded.

The *AT&T-1610* compiler does not generate any `do` loop, and controls the loops with expensive `if-then` instructions. The fir loop contains 39 instructions to perform the fir filtering. The coefficient update is performed in 35 instructions. No instruction with parallel operations are generated. The compiler overhead values for both clock cycles and program memory have the highest values of all benchmarked targets, far distanced from the rest.

Since no `mac` instruction is used by the compiler, the *ADI-2101* needs 10 instructions to perform the complete fir filtering. The adaption of the filter coefficients is realized in 4 instructions. In both loops the compiler sets a parallel data read to the multiplication. The benchmarked program is simulated in 248 clock cycles.

At the *Motorola-56001*, double parallel data accesses are not performed since only one memory bank is used. Values are accessed through register addressing, updating the register pointers parallel to arithmetic and bitshift instructions. Register offsets values are defined before and within the first loop, although the defined offset does not change its value, expanding the code. The compiler also places an accumulator shift before and after all `mac` operations. The compiler translates the fir loop in 12 instructions, the updating loop in 7 instructions.

The *NEC-77016* compiler results are not available.

The translated *TI-C51* C code attains data through indirect addressing. Multiplication, value assignment and vector updating are translated compactly. Due to the indirect pointer addressing, the fir filtering is realized within 8 instructions, coefficient adaption within 5 instructions. The benchmarked program takes 232 cycles to perform the lms operation for 16 taps. The compiler generates for this target for most compact code, placing the assembly in 44 words of program memory.

# of taps = 16	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$t_g(t_r)[\mu s]$	50.225(4.05)	12.4(3.2)	21.76(6.12)	n/a	5.8(2.25)
$c_g(c_r)$	2009(162)	248(64)	718(202)	n/a	232(90)
$p_g(p_r)$	146(22)	48(20)	49(24)	n/a	44(38)
$d_g(d_r)$	36(36)	36(36)	36(36)	n/a	36(36)
$m_g(m_r)$	182(58)	84(56)	85(60)	n/a	80(74)

Table 29: LMS Benchmark: Absolute Performance

# of taps = 16	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
$\Delta c[\%]$	1140	288	255	n/a	158
$\Delta p[\%]$	564	140	104	n/a	16
$\Delta d[\%]$	0	0	0	n/a	0
$\Delta m[\%]$	214	48	42	n/a	8

Table 30: LMS Benchmark: Compiler Overhead

14. FFT_INPUT_SCALED Benchmark

Functional Description of the C program:

The benchmarked FFT_INPUT_SCALED C function, implements a radix-2, in-place, N complex input (16 and 1024 point), decimation-in-time FFT. In terms of speed optimization, twiddle factors are precalculated and provided in an array.

To avoid errors caused by overflow and bit growth, the input data is scaled. Bit growth occurs potentially at butterfly operations, which involve a complex multiplication, a complex addition and a complex subtraction. Maximal bit growth from butterfly input to butterfly output is two bits.

The input data includes enough extra sign bits, called guard bits, to ensure that bit growth never results in overflow [22]. Data can grow by a maximum factor of 2.4 from butterfly input to output (two bits of grow). However, a data value cannot grow by maximum amount in two consecutive stages. The number of guard bits necessary to compensate the maximum bit growth in an N-point FFT is $(log_2(N)) + 1)$.

In a 16-point FFT (requires 4 stages), each of the input samples must contain 5 guard bits. Indeed, the input data is restricted to 10 bits, one sign bit and nine magnitude bits, in order to prevent an overflow from the multiplication with the precalculated twiddle coefficients.

The FFT function assumes that the user has scaled input values adequately, in order to ensure that bit growth never results in overflow.

	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
16 point					
$t_g(t_r)[\mu s]$	538.78(19.88)	167.9(31.6)	343.152(21.7)	n/a	109.93(11.58)
$c_g(c_r)$	21551(795)	3358(632)	11324(716)	n/a	4367(463)
$p_g(p_r)$	621(124)	183(73)	241(101)	n/a	251(158)
$d_g(d_r)$	30(28)	30(16)	30(16)	n/a	30(11)
$m_g(m_r)$	651(152)	213(89)	271(117)	n/a	281(169)

Table 31: FFT_INPUT_SCALED Benchmark: Absolute Performance (N=16)

Generated Code:

The critical code of the FFT C implementation is its innermost loop, consisting of a butterfly kernel (four multiplications, three additions and three subtractions), one variable update, two pointer updates and two shifting and assignment operations. We will take special attention at this code part.

	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns
1024 point			
$t_g(t_r)[\mu s]$	67164.03(2412.8)	21694.2(3083.8)	40525.27(3069.45)
$c_g(c_r)$	2686561(96512)	433884(61676)	1337334(101292)
$p_g(p_r)$	621(1674)[3]	183(73)	241(101)
$d_g(d_r)$	2046(2017)	1517(1024)	2046(1024)
$m_g(m_r)$	2667(3691)	1700(1097)	2287(1125)

	NEC 77016-30ns	TI C51-25ns
1024 point		
$t_g(t_r)[\mu s]$	n/a	13419.13(2017.4)
$c_g(c_r)$	n/a	536765(80696)
$p_g(p_r)$	n/a	251(236)
$d_g(d_r)$	n/a	2046(1517)
$m_g(m_r)$	n/a	2297(1753)

Table 32: FFT_INPUT_SCALED Benchmark: Absolute Performance (N=1024)

The *AT&T-1610* compiler does not place any `do` loop, as we had seen in the rest of the benchmarks. Loops are managed by `if-then` structure. No parallel operation is generated. As we had noticed at the UPDATE benchmarks, the assignment and update operations take extra instructions to manage data. The inner loop takes 235 program words and the generated code is more than 26 times slower than the reference code.

The use of temporary register variables and code parallelization techniques in the inner FFT loop improve the code for the *ADI- 2101*. The compiler overhead for the 16 point FFT arrives the lowest value towards clock cycle consumption.

The *Motorola-56001* target provides a highly optimized and parallelized assembler reference code with a 6 instructions butterfly kernel [23]. In the compiled C code, the shift operations are done through zero-overhead `repeat` loops, shifting only one bit at a time. As in the previous benchmarked programs, only single parallel moves are performed by the compiler. The use of direct register addressing takes four clock cycles when used with address offsets. Due to instruction pipelining, if an address register is changed with a `move` instruction, the new contents will not be available for use as a pointer until the second following instruction [17]. This fact affects code length and instruction scheduling, as well as parallelization possibilities and has consequences especially at the compiler overhead towards clock cycles. This compiler overhead is particularly high , more than 1200% for the 16 point FFT and more than 1400% for the 1024 point FFT.

[3] The *AT&T-1610* assembler reference code for the 1024 point fft is programmed straight forward, that is *without* a loop structure.

The *NEC-77016* compiler results are not available.

The *TI-C51* compiler performs the shifting operations within load operations, taking advantage of the Scaling Shifter at the end of the ALU [18]. The logical shifts are realized in a single-cycle operation, as a postscale operation, shifting the data coming from the accumulator. The compiler distinguishes between temporary and non-temporary data and assigns the auxiliary registers and auto-storage variables adequately. The *TI-C51* target states a compact code, as shown at the especially low compiler overhead values for memory consumption. Alike, a relative low compiler overhead is reached relative to the clock cycle consumption.

	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
16 point					
Δc[%]	2611	431	1482	n/a	843
Δp[%]	401	151	139	n/a	59
Δd[%]	7	88	88	n/a	173
Δm[%]	328	139	132	n/a	66
1024 point					
Δc[%]	2684	603	1220	n/a	565
Δp[%]	n/a	151	139	n/a	59
Δd[%]	1	48	100	n/a	35
Δm[%]	n/a [4]	55	103	n/a	31

Table 33: FFT_INPUT_SCALED Benchmark: Compiler Overhead (N=16,1024)

[4] It is not reasonable to calculate the memory overhead of assembly and generated code, because the presented reference for *AT&T-1610* has no loop structure and it is therefore longer than the C code.

15. FFT_STAGE_SCALED Benchmark

Functional Description of the C program:

The benchmarked FFT_STAGE_SCALED C function, implements a radix-2, in-place, N complex input (16 and 1024 point), decimation-in-time FFT. In terms of speed optimization, twiddle factors are precalculated and provided in an array.

The FFT_STAGE_SCALED benchmark compensates bit growth by scaling the outputs down by a factor of two unconditionally after each stage. This approach is called unconditional scaling. Input data should not be scaled as in the previous benchmark.

Initially, 2 guard bits are included in the input data to accommodate the maximum overflow in the first stage. In each butterfly of a stage calculation, the data can grow into the guard bits. To prevent overflow in the next stage, the guard bits are replaced before the next stage is executed by shifting the entire block of data one bit to the right.

In the FFT calculation, the data loses a total of (lg2 N)-1 bits because of shifting. Unconditional scaling results in the same number of bits lost as in the input data scaling. However, it produces more precise results because the FFT starts with more precise input data. The tradeoff is a slower FFT calculation because of the extra cycles needed to shift the data of each stage.

	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
16 point					
$t_g(t_r)[\mu s]$	702.45(19.88)	213.35(31.6)	421.7(21.7))	n/a	119.78(11.58)
$c_g(c_r)$	28098(795)	4267(632)	13916(716)	n/a	4791(463)
$p_g(p_r)$	647(124)	194(73)	250(101)	n/a	263(158)
$d_g(d_r)$	30(28)	30(16)	30(16)	n/a	30(11)
$m_g(m_r)$	677(152)	224(89)	280(117)	n/a	293(169)

Table 34: FFT_STAGE_SCALED Benchmark: Absolute Performance (N=16)

Generated Code:

The benchmarked C program is based on the FFT C code for input scaled data, with the addition, that at each stage data is right-shifted unconditionally. This C code consists of a `for` loop that right-shifts the content of a $2 * N\_FFT$ array by one bit. Only this difference with the FFT_INPUT_SCALED benchmark will be commented.

	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns
1024 point			
$t_g(t_r)[\mu s]$	93277.25(2412.8)	27863.75(3083.8)	52939.82(3069.45)
$c_g(c_r)$	3731090 (96512)	557275(61676)	1747014(101292)
$p_g(p_r)$	647(1674)[5]	194(73)	250 (101)
$d_g(d_r)$	2046(2017)	1517(1024)	2046 (1024)
$m_g(m_r)$	2693(3691)	1711(1097)	2296(1125)

	NEC 77016-30ns	TI C51-25ns
1024 point		
$t_g(t_r)[\mu s]$	n/a	14957.88(2017.4)
$c_g(c_r)$	n/a	598315 (80696)
$p_g(p_r)$	n/a	263 (236)
$d_g(d_r)$	n/a	2046(1517)
$m_g(m_r)$	n/a	2309(1753)

Table 35: FFT_STAGE_SCALED Benchmark: Absolute Performance (N=1024)

The *AT&T-1610* generates an expensive `if-then structure` at the commented C code part. This loop consumes 37 words of program memory. Pointer management could be sensibly optimized : e.g. pointer incrementations and decrementations are performed as solely instructions, instead of being generated parallel to other operations. The *AT&T-1610* target states the highest compiler overhead values, far distanced from the rest of the targets.

The *ADI-2101* and the *Motorola-56001* generate a `do` structure to reduct the stage data. The *ADI-2101* translates the loop in 9 program words. The loop body consists of only 7 instructions. Per stage corresponds it with 226 extra clock cycles, in comparison with the Input Scaled version. The loop body of the *Motorola-56001* is performed in only 5 instructions, optimizing data reads and writes with the pointer update.

The *NEC-77016* compiler results are not available.

The *TI-C51* compiler places also a zero-overhead `RPTB` structure. The compiler is able to reduct the loop body only to 2 instructions. The first instruction reads the data and shifts it automatically. Data is passed through the scaling shifter. The C5x provides a scaling shifter that has a 16-bit input connected to the data bus and a 32-bit output connected to the ALU. The scaling shifter produces a left shift of 0 to 16 bits on the input data. The shift can be performed then at zero-cost [18]. The *TI-C51* target states the highest compiler overhead results towards cycle consumption at the 1024 point FFT. Remarkable is the very low program memory overhead at the 1024 point FFT, with only 11% overhead relative to

[5] The *AT&T-1610* assembler reference code for the 1024 point FFT is programmed straightforward, that is *without* a loop structure.

the reference code.

	AT&T 1610-25ns	AD 2101-50ns	Motorola 56001-30ns	NEC 77016-30ns	TI C51-25ns
16 point					
$\Delta c[\%]$	3434	575	1844	n/a	934
$\Delta p[\%]$	422	165	148	n/a	66
$\Delta d[\%]$	7	88	88	n/a	173
$\Delta m[\%]$	345	152	139	n/a	73
1024 point					
$\Delta c[\%]$	3766	804	1625	n/a	641
$\Delta p[\%]$	n/a	166	148	n/a	11
$\Delta d[\%]$	1	48	100	n/a	35
$\Delta m[\%]$	n/a [6]	56	104	n/a	32

Table 36: FFT_STAGE_SCALED Benchmark: Compiler Overhead (N= 16 and 1024)

[6] It is not reasonable to calculate the memory overhead of assembly and generated code, because the presented reference for *AT&T-1610* has no loop structure and it is therefore longer than the C code.

C HLL-Kernel Benchmarks

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

- C function call overhead (CALL)
- `for` loop analysis (FOR)
- nested `for` loop analysis (NESTED_FOR)
- `do while` loop analysis (DO_WHILE)
- `while` loop analysis (WHILE)
- `float` arithmetic performance (FLOAT)
- `int` arithmetic performance (INT)
- fractional arithmetic performance (FRACT)
- `long int` arithmetic performance (LONG_INT)
- usage of multiply-add instruction (MADD)
- instruction parallelism analysis (PARALLEL)
- indexing vs. pointer addressing (INDEXING)
- effects of source code compaction (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 37: 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.

5. Questions and Answers about DSPstone

Q: The C compiler is in the design process mostly used as a glue code between different library or assembly routines. Does the DSPstone account for this?

A: Yes. The spectrum analysis benchmark from the applications suite is introduced in order to account for this.

Q: Suppose one processor has a very efficient instruction set enabling a much more efficient implementation of some algorithmic kernel, than the other one. Suppose that their compilers are of approximately same efficiency. Using the reference-code method of the DSPstone methodology, the first compiler with the more efficient instruction set will be the unterdog. Why?

A: The reference-code method does not measure the absolute quality of a compiler, but how well the compiler uses the underlying instruction set. Nevertheless, in the ADPCM we give also the results with the instruction set sensitive MSB (calculation of the most significant bit) function excluded.

Q: How do you guarantee that the reference code is the best possible one which can be written using the instruction set given? What is with the speed-memory trade-off?

A: Our assumption is that the DSP hardware vendors or their third parties are highly motivated to deliver the best possible code for a particular functionality. Also, according to our experience the offered code is mostly optimized in speed performnce.

Q: How does the methodology differ between 16-bit and 24-bit processors?

A: The primary goal of the methodology is to evaluate the efficiency of the compiler, so the difference in word length mostly does not influence the efficiency. Also, the only 24-bit processor under the test was the Motorola 56000. At the time we have started the benchmarking there was only a limited support available for the Motorola 16-bit processor 56156.

6. 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 to floating-point DSP compilers.

References

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

[2] E. Lee, "Programmable DSP architectures: Part II," *IEEE ASSP Magazine*, pp. 4–14, January 1989.

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

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

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

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

[7] M. Klerer and H. Liu, "A new benchmark test to estimate optimization quality of compilers," *ACM Sigplan Notices*, vol. 23, Mar. 1988.

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

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

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

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

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

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

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

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

[16] AT & T, *DSP1610 Digital Signal Processor*, 1992.

[17] Motorola, *DSP56000/DSP56001 Digital Signal Processor User's Manual*, 1990.

[18] Digital Signal Processing Products, Texas Instruments, *TMS320C5x User's Guide*, 1993.

[19] Analog Devices, *ADSP-2100 Family User's Manual*, 1993.

[20] Analog Devices, *ADSP-2100 Family C Tools Manual*, 1993.

[21] Motorola, *Fractional and Integer Arithmetic Using the DSP56000 Family of General-Purpose Digital Signal Processors*, 1988.

[22] L. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing.* Prentice-Hall, Inc., 1975.

[23] Motorola, *Implementation of Fast Fourier Transforms on Motorola's DSP 56000 / DSP 56001 and DSP 96002 Digital Signal Processors*, 1991.