DSP & MULTIMEDIA TECHNOLOGY Magazine, vol. 4, no. 5, July/August, 1995, pp. 27 – 45

COMPILERS FOR DIGITAL SIGNAL PROCESSORS
The Hard Way From Marketing- to Production-Tool

Vojin Živojnović

Aachen University of Technology
Aachen, Germany

I. Introduction

High-level language compilers for DSP processors are one of the rare products of the DSP industry for which a controversy regarding their use and usefulness exists. Although DSP compilers are available for almost all fixed- and floating-point DSP processors found on the market, their present role in the software design process is far less significant than expected by users, compiler designers, and sales personnel. Contrary to standard programming practice and despite all efforts, assembly programming remains an inevitable part of the DSP software design process. As a consequence, a provocative question can be asked: *Are state-of-the-art DSP compilers marketing or production tools?* Facts presented in this paper should help the reader in forming his own conclusion about this question. We present below an overview of DSP compiler technology, their characteristics, and their role in DSP code development.

This paper should help the reader to form a complete picture of the main features of DSP compilers and perhaps help in predicting future developments in the highly interesting field of DSP applications. The paper is organized as follows. Following the introduction, Section II discusses the low efficiency of state-of-the-art DSP compilers and its consequence for development of multimedia applications. Section III briefly reviews the history of DSP compilers. Section IV discusses the role of DSP compilers in DSP code development. An overview of DSP processor architectures is presented in Section V. This section discusses the influence of DSP processor architecture on compiler design and compiler efficiency. Section VI gives an overview of high-level programming languages for programming DSP processors. Language extensions and compilation flags of DSP compilers are discussed in Section VII. Section VIII discusses the role of fixed-point arithmetic in fixed-point DSP compilers. Compiler optimization issues are reviewed in Section IX. Section X presents some results from a quantitative evaluation of DSP compilers. Finally, Section XI discusses future developments in the DSP compiler field.

II. DSP Compilers, Processors, and Multimedia Applications

DSP compilers are software tools which enable programming of DSP processors in a high-level language (HLL). Instead of using cumbersome and time-comsuming assembly language, the programmer expresses the computation in a language which is more easily understood and used. As a consequence, the programmer's productivity is higher, and the code is easily portable to various platforms and can be reused for other purposes. All these factors shorten the time to market of the final product. As in standard, general-purpose processing, there is a strong motivation for using high-level programming languages and compilers for programming DSP processors.

DSP processors are special-purpose processors. They were introduced in the early eighties to enable cost-efficient real-time implementations of DSP algorithms. As successors to microcontrollers, they inherited many microcontroller characteristics - on-chip memory and I/O, small size, and low cost. However, DSP processors gained a number of additional, highly specific architectural features which increased their processing power. At the same time, these features made them highly unsuitable as compiler targets and hardly controllable from standard programming languages. As a consequence, most state-of-the-art DSP compilers show serious deficiencies in efficient utilization of the target architecture.

Recent efforts in DSP processor design try to make the processor architecture more compiler friendly. Although risky for its position on the market - chip price and power consumption will probably rise - this could be a step in the right direction. However, even the existing, architecture-related efficiency problems of DSP compilers are solvable. Despite the differences in the design goals between general-purpose and digital signal processing, and between respective processor architectures, the problems of DSP compiler design could be successfully solved by using and extending the existing know-how of compiler and programming language specialists. The reason why this has not been done yet lies in the financial investments necessary to support compiler design teams attacking the problem. Such specialized teams are expensive, and the DSP processor and compiler manufacturers could not recoup their development cost by selling the compilers. As a consequence, the DSP industry has largely decided to use existing compilers for general-purpose computers and to adapt them modestly (and more or less successfully) to DSP processors. Such an approach guarantees a compiler which fulfills the main goal of every compiler - to generate correct code. Also, for DSP architectures which are similar to general-purpose architectures and for code fragments that are not signal-processing intensive, the results can be satisfactory. However, for most DSP processors and applications, the generated code is far from optimum and cannot be used in the final product without additional, hand-crafted modifications.

New factors are influencing developments in the DSP compiler market. The DSP market is no longer a small niche of the overall computer market. Especially in multimedia and mobile communications, the market for DSP processors is constantly growing. Orders to the DSP chip industry have grown from quantities of tens of thousands to millions, or even tens of millions. The number of new companies that are using DSP technology is also constantly increasing. Such developments could be highly beneficial for the DSP compiler market.

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

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

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

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

III. Short History of DSP Compilers

Although the first DSP processors appeared in the early eighties, the first commercial DSP compilers became available in 1988. This delay stemmed from the compiler-unfriendly architectural characteristics of the first DSP processors, such as a small number of registers and limited addressing capabilities.

One of the first commercially available DSP compilers was the C compiler for the AT&T DSP32 floating-point processor. AT&T re-targeted the UNIX portable C compiler to the DSP32 [1]. At the same time TI released their C compiler for the TMS320C25. This was also based on an existing C compiler for general-purpose processors. This compiler became the basis for the development of their fixed-point C compiler and a later C compiler for their floating-point processors.

Motorola and Analog Devices [2] started with their own C compiler solutions for the DSP56000 and ADSP-2100 fixed-point families, respectively. However, both companies later decided to switch to the GNU C compiler – `gcc` – from the Free Software Foundation [3] as the basis for their new fixed- and floating-point compilers.

Since that time the GNU C compiler has been used by numerous companies wishing to develop a bug-free DSP compiler without large investments in compiler design. The `gcc` compiler by the Free Software Foundation is available free of charge under the terms of the GNU Public License (GPL). This license requires that companies incorporating GPL code into their products must make available the source code for the products at no charge. Attempts to use the *standard* `gcc` retargeting procedure mostly failed. To obtain acceptable efficiency of the code, complex changes to the `gcc` have been necessary. Also, additional assembly level optimizers have often been added as a post-compiling procedure (e.g., in compilers for the DSP56000 by Motorola).

In 1989 an effort was started to standardize DSP-oriented extensions of the ANSI C language [4,5,6]. Although a lot of very useful extensions have been proposed by the Numerical C Extensions Group (NCEG), the different interests of the NCEG members coming from the DSP and general numerical processing fields have made agreeing on a common standard impossible.

The growing DSP market has become attractive for companies with experience in compilers for general-purpose processors. In 1990 Tartan released the first Ada DSP compiler which targeted TI's TMS320C30 processor [7]. In 1993 the same company released the first C++ compiler specifically designed for DSP processors [8].

Intermetrics released their C compiler for the Motorola DSP96000 floating-point DSP processor family in 1990, the NEC 77240 processor in 1992 [9] and the Mwave processor in 1993. Intermetrics' newest design is the C compiler for the NEC 77016 fixed-point family [10,11].

IV. Role of DSP Compilers in DSP Code Development

Devices with embedded DSP functionality obtain their processing capabilities through use of programmable and/or hardwired processing elements. If the predominant processing power comes from programmable integrated circuits we speak about DSP processor-based designs.

The main advantages of DSP processors over ASIC solutions are lower development costs, shorter development time, and a high degree of flexibility in the design. Development of devices with embedded DSP processors can be divided into two main tasks: code development and processor/hardware integration. In this paper we are interested in the former.

IV.1. Assembly-Based Design

The classical, assembly-based approach to DSP code development is to convert the algorithm directly into the processor's assembly code (Fig. 1). The main problem with this approach is that programming, debugging, and functional verification all take place at the assembly level. The resulting productivity is mostly low, and the design sometimes becomes a real nightmare. Although most complaints are about programming in assembly language, experience shows that the main problem is actually in assembly-level debugging and functional verification. Even though experienced DSP programmers learn every new assembly language and obtain the necessary programming skill very quickly, fixing coding bugs - especially conceptual bugs - in assembly can cost huge amounts of time.

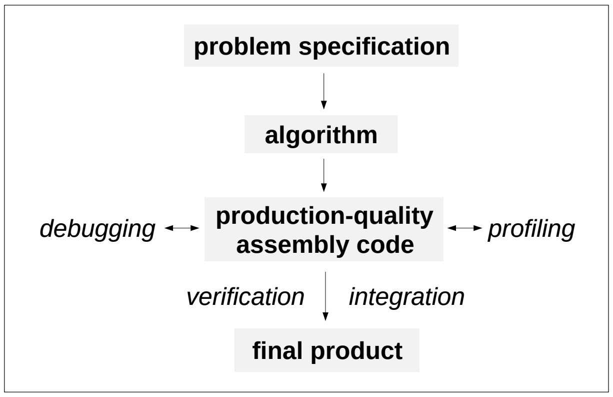

Figure 1: Classical Approach to DSP Code Design.

A well-known and proven approach to improving code design is to program in a high-level language and use the compiler as the translator to assembly code. High-level language programming offers a number of advantages over assembly programming. The most important are:

- programming comfort - the high expressive power of the high-level language makes programming more comfortable and thus more productive;
- testing and debugging - high-level description of the program enables efficient testing and debugging;
- re-usability - the same code can be reused in other applications;
- portability - the same code can be ported to other targets supporting the same language; and
- maintainability - less effort is necessary if changes or extensions of the code have to be made.

All these advantages are valid for DSP code development, too. Unfortunately, the relatively low efficiency of state-of-the-art DSP compilers is a serious limitation. The overhead in execution time and memory utilization introduced by the compiler is very often unacceptably high (see Section X). In general, the code generated by the DSP compiler cannot be used directly in the final product, and the *compile-link-run* procedure cannot be applied. Exceptions are situations in which the DSP processor subsystem of the device is not cost-sensitive or in which the processing speed is not critical or can be scaled by faster processors or additional processing units, as in rapid prototyping. In these cases the relative inefficiency of DSP compilers can be tolerated. In the remainder of this section we concentrate on implementations where this is not the case.

IV.2. Heterogeneity of DSP Programs

Programs found in typical DSP applications are not homogeneous. In addition to their functionality, different fragments of the program have different run-time and compile-time characteristics. For example, in the program of a V.32 modem, the instructions implementing the filtering operation are executed more frequently than the instructions doing startup synchronization. Such behavior is known as locality of reference. The well known general rule of thumb is that 90% of the execution time is spent in 10% of the code [12].

Another form of heterogeneity is introduced by the type of the code. We shall distinguish two code types: DSP-type and GPP-type code. DSP-type code is made up of instructions which implement DSP-specific algorithms (e.g., FIR or FFT). Features of the DSP processor, like multiply-add instructions or bit-reversed addressing, are specially tailored to speed up this type of code.

GPP-type code is found in general-purpose applications where string manipulations and general numeric computations dominate. In DSP applications, GPP-type code is found in the form of glue logic between DSP-type code fragments and is characterized by heavy use of general arithmetic and control statements of the language, like `for`, `if-then-else`, and `switch`. This is the reason why GPP-type code is often also referred to as control-type code.

Existing DSP compilers handle GPP-type code in a much more efficient manner than DSP-type code. Almost all available DSP compilers are made by re-targeting and modifying existing GPP compilers which are tailored according to the general-purpose code model. Also, if the underlying architecture is a GPP processor, the compiler has a much easier task of generating efficient code. Fig. 2 shows the compilation problem in dependence on the code and architecture type. If the code is of DSP type and the architecture is tailored to DSP-type applications, current compilers have difficulty generating efficient code. In cases where the code is of GPP type or the architecture is more similar to the GPP processor architectures, the problem becomes easier. The reason for such behavior lies in the model mismatch. The standard languages, like C, and the related compiler technology are developed to match the general-purpose application and architecture model. If the model is not appropriate, only suboptimal results can be obtained.

Execution frequency is related to code-type characteristics. The time-critical parts of the code are very often of DSP type, so most of the execution time is spent in the code which is hard to compile efficiently. This forces reliance on the manual recoding process.

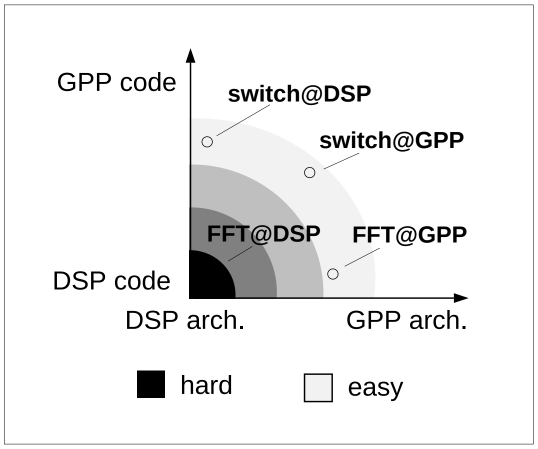

Figure 2: Compilation Problem in the Code-Architecture Space. (switch@DSP means `switch` code running on a DSP processor)

IV.3. High-Level Language-Based Design

As of today, the design principle of using DSP compilers relies on the use of the heterogeneity of typical DSP code. A DSP compiler-based design flow is presented on Fig. 3. Its main characteristic is the introduction

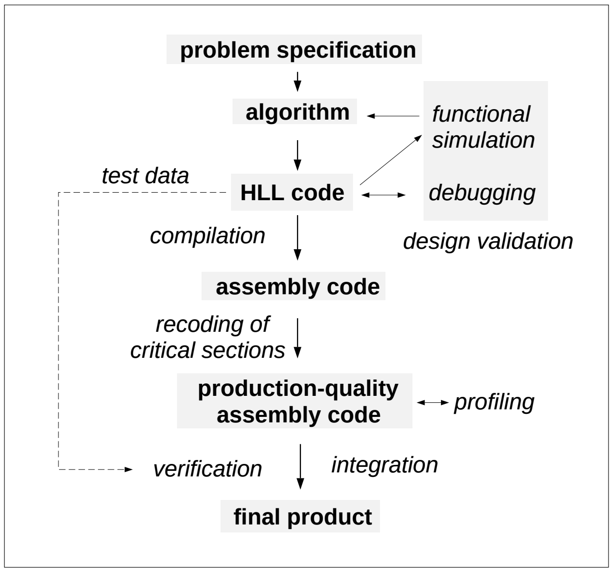

Figure 3: HLL-Based Approach to DSP Code Design.

of HLL programming and use of the DSP compiler. The intention is to move the validation (debugging and functional verification) from the assembly level to the HLL level in order to make the design less error prone and thereby faster.

The role of the HLL program is twofold. First, the HLL code serves as a bit-accurate prototype in which debugging and functional verification is much more comfortable than on the assembly code. Working on a high-level description hides the unnecessary details of the implementation and enables the user to fully concentrate on the functional behavior of the program. Also, by feeding the HLL program with user-defined test input sequences, the signal at predefined internal points can be determined. Even if the input and corresponding output verification sequences are provided a priori, as in the case of standardized algorithms (e.g., CCITT-ITU standardization), they determine only the input and output of the program and not the internal signals. In this way no details about the sources of incorrect behavior are available. Using the bit-accurate HLL code, the designer can generate all internal signals as the response to the test input. These signals are highly useful for the verification of the final assembly code.

Second, the HLL program is used as the input to the DSP compiler, which then generates the assembly program. For the most part, the generated code is of unsatisfactory run-time efficiency, both in execution time and program/data memory utilization. The procedure which has proved to be the most efficient is to locate the time-critical parts by profiling and recode them manually or use fast library routines. An example for this procedure is presented in [13].

Recoding is the most cumbersome and error-prone part of the HLL-based design process. Modifying the time-critical parts in the HLL program by using functions or assembly in-lining is advantageous. Also, the memory map of the HLL program should be laid out with the final memory configuration in mind.

In general the designer should try to keep the correspondence between the HLL and the assembly program as close as possible. Changes during recoding of the assembly program should be mirrored in the HLL program. Although it is simpler to put aside the HLL program and concentrate only on the assembly code, the additional work in synchronizing the two programs will be paid back quickly. During debugging and verification of the recoded assembly code all the advantages become clear. For example, if the variables in the HLL and assembly program represent the same signal, and the program is fed with the same input data, the values of the variables can be compared during run-time. In this way bugs can be detected and located quickly (Fig. 4). Also, by maintaining the correspondence between the HLL and assembly code future maintenance is much easier and the code is more reusable.

V. DSP Architectures and Compilers

The architectures of state-of-the-art DSP processors are tailored to be highly efficient on DSP-type programs. For example, almost all DSP processors are able to compute an FIR filter with a speed of one instruction per filter tap. It is obvious that more than one operation per instruction has to be executed. In the case of a FIR filter, six operations (one multiplication, one addition, two memory moves, and two pointer updates) have to be accomplished within one instruction cycle. Additionally, looping has to be executed without any additional instructions.

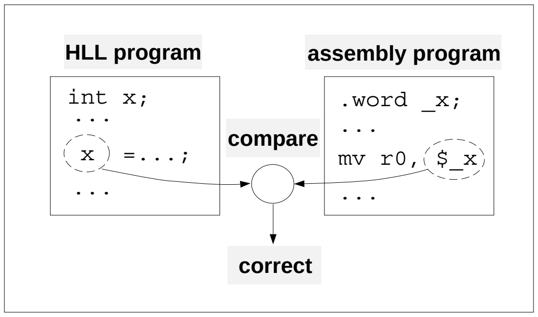

Figure 4: Comparative Debugging.

To reach this goal and at the same time keep the price and power consumption of the chip low, DSP processor architectures developed a number of characteristics that make them very unfriendly to standard compilers. These characteristics and their influence on DSP compiler design are explained in more detail below.

small number of general-purpose registers

The first fixed-point processors had a small number of general-purpose registers. For those architectures it was almost impossible to apply standard compilation techniques. The compiler was not able to allocate registers to variables since all the registers had to be used as temporary variables during expression evaluation.

Later on, the number of general-purpose registers grew in every new architecture appearing on the market, thereby making the compiler's register allocation task easier and register spilling (freeing a register by saving its content in memory) less frequent. Modern floating-point processors offer enough general-purpose registers to enable efficient code generation.

heterogeneous register set

Registers of a fixed-point DSP processor are very often of different lengths and therefore cannot deliver the same accuracy during arithmetic operations. For example, some DSP processors have only one or two registers of full length - accumulators. Full accuracy can be guaranteed only if all intermediate values of a computation are stored in one of these registers.

DSP compilers work with data types which are of constant width during the lifetime of a variable. If this width is the width of the accumulators, the computation becomes very slow because additional memory accesses are necessary (e.g., if `long` type is used). On the other hand, if the data type is the width of the data memory, the computation is inaccurate. Only some compilers permit explicit control over the accumulators [10].

non-orthogonal instruction set

An instruction set is said to be orthogonal if the operation, operand type, and addressing mode of the instructions are mutually independent [14]. For example, if an instruction has two register operands, it should be possible to execute the same operation on any combination of general-purpose registers of the processor register file.

This is mostly not the case for DSP processors. DSP processors are actually half-way between accumulator and general-purpose register architectures. Instructions have a predetermined subset of registers which can be used as valid operands. For example, the multiply-accumulate instruction in most DSP processors expects the two multiplicands to be located in different, memory-space dependent register banks. Obviously, the non-orthogonality of the instruction set introduces differences among registers and makes the register set heterogeneous.

In the case of non-orthogonal instruction sets, the instruction scheduling and register allocation optimizations have to work with a reduced effective number of registers. This decreases efficiency because many register-to-register moves are necessary to meet the requirements of the instructions.

multiple function units

In DSP processors multiple data and address processing units can operate in parallel and can be used by a single instruction. In this way DSP processor architectures are similar to VLIW architectures [14].

To use the whole processing potential of the processor, DSP compilers have to detect operations which can be done in parallel and schedule them adequately at compile-time. Current compilers can detect simple combinations of arithmetic operations that can be done in one instruction (e.g. multiply-add with shift). Parallel operations in address processing units seem to be a more complex problem.

multiple data memory spaces

The Harvard architecture of DSP processors is characterized by different memory spaces for the program and data. The extended Harvard architecture is additionally characterized by multiple data memory spaces - combined program/data and data-only memory spaces. Full speed can be obtained only if the data is properly distributed among different memory spaces.

Current DSP compilers are not able to make any decisions about the allocation of data, as well as program/data and data spaces. Architectural descriptions, which are the source of knowledge about the processor's architecture, do not contain this type of information. As a consequence, the generated code is suboptimal. Only some newer DSP compilers offer language extensions or pragmas for user-directed allocation of multiple memory spaces.

heterogeneous memory space

DSP processor data and program memory can be on-chip or off-chip. Off-chip memory usually cannot be accessed at the same speed as on-chip memory - additional wait states have to be inserted. Also, access to multiple data spaces in some processors cannot be done in a single instruction if the memory is off-chip. Proper memory allocation can significantly increase processing speed in these situations.

The standard approach during code generation is to postpone the decision about final memory mapping of variables until the linking process. However, to optimally deal with heterogeneous memories the compiler itself should be able do the memory mapping. State-of-the-art DSP compilers are missing this capability.

pipelined architecture

To speed up computation, most modern DSPs use pipelined architectures. Pipelining means that the processor works on different portions of different instructions simultaneously. For example, while the processor is executing instruction N, it might be fetching the operands of instruction N+1, and fetching the opcode for instruction N+2. Pipelining allows faster processor clock rates but can have side effects that the programmer must avoid. Pipeline effects visible to the DSP programmer on many DSPs are delayed branches, pipeline flushes, and delayed register updates.

Pipelining effects increase the complexity of the compiler's instruction scheduler. Some compilers, like the `gcc` provide an internal mechanism for dealing with pipelining effects, but still cannot deliver optimum results. More aggressive optimizations in new DSP compilers should improve the use of pipelining and thereby speed execution.

modulo and bit-reversed addressing

Non-linear addressing modes permit non-sequential access to data memory locations. Modulo addressing and bit-reversed addressing are typical non-linear addressing modes found in DSP processors and are used for circular buffers and FFT computation, respectively.

Without explicit information from the high-level language, the compiler is unable to detect addressing which can be done more efficiently directly by the address generator. The main limitation comes from the language side.

hardware (zero-overhead) loops

Hardware support for looping is provided in most DSP processors. The loop is executed with little or no time spent managing the iterators.

Most DSP compilers can use hardware looping very efficiently.

no support for software stack operations

Most DSP processors have a simple and relatively small hardware stack that is accessed implicitly by interrupts and subroutine call/return instructions. The hardware stack cannot be used by compilers for argument passing or allocation of automatic variables.

DSP compilers have to build the program stack in software, which requires additional instructions and registers.

Summarizing this section we can conclude that there are two ways to overcome the architecture-dependent problems of DSP compilers. One way is to improve the interaction between the user and the compiler by using DSP-oriented high-level languages, language extensions, and directives which are better suited for the problem at hand. The other way is to improve the optimization capabilities of the compiler. These issues will be treated in the next sections.

VI. High-Level Languages for DSP Programming

Compared to compilers for general-purpose computers, DSP compilers are available for relatively few high-level languages.

At this writing, C is the most common high-level language for programming DSP processors. It is simple to use, it permits hardware-close programming, and perhaps most important, it is widespread among users. Existing DSP C compilers support the ANSI C language standard [15], and some also support the older K&R standard.

The C language is not the ideal language for programming DSP-type applications. For example, digital filters exhibit a specific behavior which can be best described by a data-flow model and block-diagram (graphical) languages. The main disadvantage of block-diagram languages is that they behave very poorly on control-type code.

Some feel that C++ is the DSP language of the future. Its main strength is the high modeling efficiency which can be easily adapted to user's needs. There is a widespread opinion that the C++ language is not as efficient as pure C. Surely there are some C++-specific constructs which produce a large overhead compared to C; however, the user can always switch back to pure C if this is advantageous [8].

The Ada programming language offers some very interesting features for the DSP programmer, especially if the target is a fixed-point processor. Ada is a more powerful language than C. It is a strongly typed language with a rich set of types, subtypes, and type attributes. It even supports concurrent processing. Additionally, for applications in the military field there is sometimes no alternative. However, Ada's main disadvantage is its complexity (e.g., twice as many keywords as C) and low modeling efficiency for hardware-related programming (e.g., no instructions for bit manipulation).

Applicative languages, like Silage [16], Signal [17], or Data-Flow Language (DFL, a language based on Silage) also have a number of advantages over functional languages like C or Ada. In applicative languages instruction precedence is defined by pure data precedence, thus making it easy to implement a data-flow graph model of the computation. Despite this (and other) advantages, applicative languages have not found acceptance among the DSP programmer community.

Although existing languages are not optimal for DSP programming, new languages for DSP programming will probably not be developed. Introducing language extensions for existing high-level languages seems to be a more reasonable step toward improving the design process.

VII. Language Extensions and Compilation Directives

The role of programming language standards is to specify a common platform for software development of a set of common applications on a set of platforms with common characteristics. For example, the ANSI C standard was developed with a particular application and hardware model in mind. If the application or hardware model do not match the standard model, only suboptimal results can be obtained.

In the case of using the C language to program DSP processors, the application and hardware model do not match. The best examples of this mismatch are the absence of support for fixed-point arithmetic and the assumption of a homogeneous memory architecture. Fixed-point DSP applications need specialized arithmetic which is not covered by existing C data types and arithmetic rules. Also, the standard approach used in C compilers incorporates a single homogeneous memory bank. As mentioned in previous sections, memory in the typical DSP processor is highly heterogeneous.

One way to overcome this model mismatch is to introduce extensions to the language. This necessity was recognized quite early. The role of language extensions is twofold. First, extensions improve the expressive power of a language. This has a direct impact on the modeling efficiency and thereby on the productivity of the programmer. Second, extensions improve the run-time efficiency of the compiler.

Standardization of C extensions necessary for numerical computing started in 1988 as a subcommittee of the ANSI C X3J11 standard committee, known also as the Numerical C Extensions Group (NCEG). The main intention of the NCEG was to standardize math libraries and suggest changes to the C language [6]. The committee was formed by members from companies interested in general numerical processing and digital signal processing. Unfortunately, no Numerical C standard has been issued to date. Although numerical processing and digital signal processing have a similar application model, the differences in the hardware model between general numerical processing and digital signal processing is too large. It is easy to justify the necessity for language extensions, but it is not at all easy to standardize them. Numerical C extensions can be found in the Analog Devices' floating-point C compilers (see [5,6]).

Despite the lack of an NCEG standard, many C compilers support language extensions to varying degrees. For example, almost every DSP C compiler supports some target-specific features. Even in the ANSI C standard using the `pragma` keyword implementation-dependent action can be performed. Also, all the GNU-based compilers support at least the `gcc` extensions.

The extensions to the ANSI C language standard that are supported in DSP compilers can be divided into three groups:

VII.1. General Extensions

General extensions are extensions which enable more efficient programming in C without the restriction on a specific type of application or architecture. The following extensions are part of the GNU compiler [3] and therefore are in all of its DSP-targeted derivates:

- in-line assembly;
  In-line assembly permits intermixing C and assembly instructions in the same source code. It is used to exploit processor features which are unreachable through C, like reading status flags or setting some specific non-memory-mapped registers.
- variable-length arrays;
  Variable-length arrays permit run-time specification of the array length.
- in-lining;
  In-lining eliminates the context switching overhead introduced by the function call. The core of the function is inserted at the position of the call. In-lining represents a typical tradeoff between faster/larger and slower/smaller code.

VII.2. Numerical Extensions

Numerical extensions are tailored to meet the needs of applications dominated by numerical computations. Although their main goal is to simplify programming, i.e., raise the expressive power of the language, they also enable more efficient code generation for DSP processors. Some examples are:

- iterators;
  Iterators help in specifying operations which are performed repeatedly on a large amount of data. For example, `A[I]=0;` stands for `for(i=0;i<N;i++) A[i]=0;`, where `I` is an iterator. This extension is part of the Numerical C proposal and can be found in the compiler described in [6].
- complex data type;
  The complex data type is used to specify pairs of variables describing the real and the imaginary part of the value. This extension can be found in [18].
- fractional data type;
  The fractional data type specifies fixed-point numbers with values between -1 and 1, and is highly useful for programming fixed-point processors. This extension is implemented in [19] and reported in [10].

VII.3. DSP-Oriented Extensions

DSP-oriented extensions are introduced to improve compiler efficiency in use of specific architectural features of DSP processors.

- multiple-memory spaces;
  Multiple memory spaces is one of the main characteristics of DSP architectures. To take advantage of this feature variables have to be properly distributed between the memory spaces. Therefore, memory space qualifiers (that is, keywords that tell the compiler to use a particular memory space) for variables are necessary.
- bit-reversed and circular addressing;
  Another specific feature of DSPs is non-linear addressing modes, like modulo or bit-reversed addressing. To be controllable from the C code, the compiler has to provide appropriate language extensions or compiler directives.

In addition to language statements, most compilers use compilation directives and flags for control of the compilation process. The standard process in which the program is fed to the compiler and the compiler delivers the output is unsuitable for DSP code design. Enabling a closer interaction between programmer and compiler using an *interactive compilation tool* would be highly advantageous.

VIII. DSP Compilers and Fixed-Point Arithmetic

In this section we concentrate on one of the language extensions which is of great importance if the target is a fixed-point DSP processor - support for fixed-point arithmetic.

VIII.1. Fixed-Point vs. Floating-Point Processing

Fixed-point DSP processors have a number of advantages over floating-point processors. Most important are lower costs and lower power consumption of the hardware, as well as higher computation accuracy for the same word length. The price paid lies in decreased dynamic range and extra programming effort that must be expended to implement manual scaling.

To avoid nonlinear effects introduced by overflow, underflow, saturation, and wrapping during computation, control of each variable's range has to be done by appropriate scaling of the operands. In the floating-point case the exponent of a variable is part of the run-time representation and is computed and updated automatically by the floating-point ALU, but in the fixed-point case the exponent of each variable is implicit and determined by the programmer off-line using the predicted dynamic range of the variable. Therefore the scaling of fixed-point operands has to be done explicitly by the programmer. This is generally a tedious and error-prone process.

VIII.2. Fractional vs. Integer Arithmetic

Fixed-point representation of a number using $N = x + y$ binary numbers is denoted by $Qx.y$, where x and y determine the number of bits to the left and right of the decimal point. The most common fixed-point representations are integer ($y = 0$) and fractional ($x = 1$). If the numbers are represented in two's complement, the fractional representation covers the numbers from $[-1, 1)$.

The fact that multiplication of two fractional numbers produces again a $[-1, 1)$ result and no overflow can occur (except for -1*-1), is recognized by the programmers of DSP processors as advantageous for scaling. This is one of the reasons why most state-of-the-art fixed-point DSP processors support fractional arithmetic and only a few additionally support integer arithmetic. We present below the main differences between fractional and integer arithmetic and show that fractional computation can be even more accurate.

Integer and fractional two's complement number representations on DSP processors differ in the multiplication operation. Fractional and integer multiplication of two N-bit numbers both yield a $(2N - 1)$-bit result (assuming that the cases of -1*-1 (fractional) or MININT*MININT (integer) are excluded). If the result has to be represented by $2N$ bits, the fractional result is right-side and the integer left-side extended. Therefore, fractional multiplication of two N-bit two's complement numbers stored as a $2N$-bit result is equivalent to the integer multiplication with a subsequent one-bit left shift.

Where integer arithmetic uses the additional bit for possible range extension in subsequent operations, fractional arithmetic enables more accurate computation. For example, if `z=a*b+c;`, with `a,b,c` and `z` fractional numbers is computed, a more accurate result is obtained than where the same variables are integers scaled to represent the same values. The difference is a consequence of the fact that after the fractional multiplication an implicit scaling happens via a left shift. So, the subsequent addition can be done with an additional bit of accuracy. Fig. 5 illustrates this fact.

Another very important difference between fractional and integer representations becomes apparent during word-length reduction, e.g., saving a double-word accu-

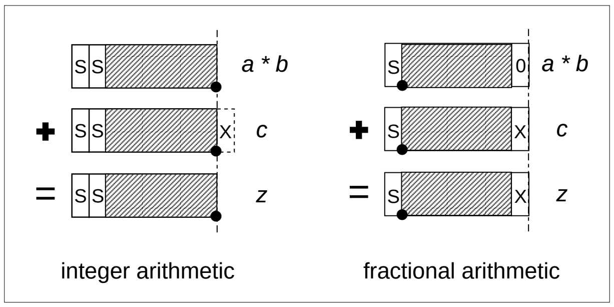

Figure 5: Accuracy of the `z=a*b+c` Calculation.

mulator into a single-word memory location. To keep the implicit exponent of the number unchanged, in the fractional case the upper and in the integer case the lower word is extracted. This difference plays an important role if fractional computation has to be emulated by integer arithmetic.

VIII.3. Fractional Arithmetic and the C Language

Fixed-point arithmetic is an important part of real-time programming languages (see [20]). For the same power consumption and price the fixed-point arithmetic is always faster than the floating-point. The standard C language supports only Qx.0 fixed-point data types and arithmetic. If fractional arithmetic has to be implemented in C it has to be emulated using the provided integer types and operations. This is generally a time-consuming operation. To extract the upper word of the multiplication result, a casting to `long` has to be applied. In most compilers this introduces a call to a function which enormously slows down the computation. Also, the necessary left shift after each multiplication introduces more overhead. As an example, fractional arithmetic is emulated in C and an off-the-shelf fixed-point DSP compiler is tested (Fig. 6).

code	#clock-cycles
`int a, b, c, z;` `z=(((long)a*(long)b)>>23) + c;`	326
optimum assembly code	2

Figure 6: Overhead of the Emulation of Fractional Arithmetic (`int` has 24 bits).

The emulation of the fractional multiply-add operation requires more than 300 cycles, although the same operation can be done in only two cycles in assembly. Our experiments with the arithmetic of a GNU-based DSP compiler show that if the compiler is changed to support fractional arithmetic only six clock cycles are needed.

The introduction of the fractional data type as an extension to the C language standard is necessary if the C compiler is to be used for programming of fixed-point DSP applications. The above discussion shows that the fractional data type is advantageous not only for the scaling operation, but also for the accuracy of the computation. If emulation with integer arithmetic is used, an extremely high overhead is introduced. Our experiments with some GNU-based DSP compilers show that the fractional data type can be added in a simple way, especially if the `float` data type is overloaded. Unfortunately, at the time of this writing only a small number of fixed-point DSP compilers support the fractional data type [21,22].

VIII.4. Compiler Support for Automatic Scaling

If the dynamic ranges of the variables are known a priori, the compiler has all the necessary information to compute the scaling by itself. At compile-time the compiler attaches to each variable an implicit exponent computed according to the range. If an arithmetic operation on variables with different exponents has to be executed, the compiler inserts the necessary scaling directly into the code to adapt the implicit exponents. A prototype of a DSP compiler with automatic scaling was developed in [23]. Similar approaches at the assembly level can be found in [24]. Unfortunately, none of the commercially available compilers supports this feature.

IX. Compiler Optimizations

Processing of state-of-the-art compilers can be generally divided into two main phases: front-end and back-end processing. In the front end the source code is analyzed and translated into an intermediate representation. This representation is neither source language- nor target machine-dependent. In most compilers it has a form of simple three-address instructions (two operands and the result) or of data structures (e.g., in `gcc`). In the next step, during the back-end processing, the intermediate representation is processed and converted into the final assembly or machine code. This process is depicted in Fig. 7.

To generate high-quality output code, compilers apply a series of optimizations to the intermediate and final

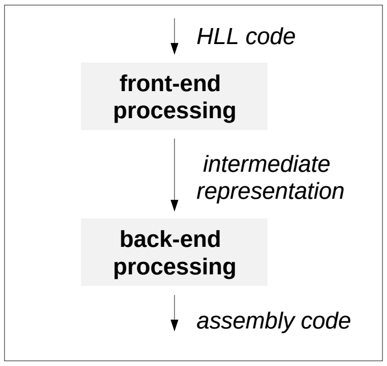

Figure 7: The Compilation Process.

assembly code. The goals of the optimization are to reduce execution time and memory consumption. Most compilation algorithms delivering optimum results belong to the class of NP-complete problems for which only algorithms of exponential complexity are presently known. What does this mean in practice? For example, if the compiler has to optimize a page with $n = 50$ assembly instructions, some optimum algorithm will need $T = ba^n$ time units. If we use typical values of $a = 2$ and $b = 10^{-8}$, we see that the optimum result will be delivered in 130 days. This is a long time to wait.

The only alternative is to use algorithms which are of polynomial complexity and which deliver suboptimal solutions, but do so in a reasonable time. However, even in this case the chunk of code has to be of modest size. What the words "reasonable" and "modest" mean depends on the application domain. Where the compilation of one page of general-purpose code has to be finished in the time it takes for one gulp of coffee, in DSP code development even the duration of an entire lunch could be tolerated.

Compiler optimizations can be divided into machine-independent and machine-dependent optimizations. Most compiler specialists from the general-purpose computer field understand the term "optimizing compilers" to refer to machine-independent optimizations. These optimizations are introduced at the intermediate level, without using any information about the target architecture and instruction set. For completeness the most common machine-independent optimizations are reviewed below. For more details refer to the compiler bible by Aho [12].

- strength reduction
  Replace a more expensive operator by a cheaper one, e.g., $x^2 = x * x$.
- common subexpression elimination
  Avoid recomputing the expression if the previously computed value can be used.
- constant propagation
  Compute expressions involving only constants at compile-time.
- dead code elimination
  Remove code that computes values that are never used.
- loop unrolling
  Write a loop as a sequential stream of repeated statements.
- loop-invariant code motion
  Move to outside the loop expressions whose values do not change as the loop is executed.
- function in-lining
  Insert the body of a function at the point of the function call.

Most machine-independent optimizations can be made superfluous by a proper programming style. Especially for the typical DSP user and typical DSP application, machine-independent optimizations are of minor importance for the efficiency of the final code.

What a DSP user is really interested in are machine-dependent optimizations. These optimizations are target dependent and directly influence the efficiency of the generated code. The most important are as follows:

- register allocation and assignment;
  Allocation of registers for variables and assignment to the machine-specific register set.
- instruction elimination;
  Removal of unnecessary instructions.
- control-flow optimization;
  Elimination of unnecessary conditional and unconditional jumps.
- instruction selection - compaction;
  Multiple operations can be executed in one instruction cycle using parallel execution units. The instruction compactor analyzes the instructions and tries to combine them into a small number of parallelized instructions. For example, a multiply and subsequent add can be combined into a single multiply-add instruction.
- instruction scheduling - software pipelining;
  The execution sequence of instructions is changed to better exploit the characteristics of the underlying architecture. Mostly it is applied to avoid pipeline stalls and prepare the code for better use of multiple processing units.

Most of these optimizations are performed only on fragments of code using a moving window. This procedure is commonly called *peephole optimization.*

In most state-of-the-art DSP compilers machine-dependent optimizations are far from optimal. Why? One reason is the DSP-specific architectural features, which are rarely understood by the compiler without a programmer's help. However, it seems that even a more important reason is the general approach to DSP compilers. Not only are DSP compiler designs heavily influenced by the design of GPP compilers, but they also suffer from constraints which have been introduced by the specific type of general-purpose applications and the way general-purpose code is developed. An example is algorithmic complexity of the optimizations.

DSP code development needs much more complex optimizations than general-purpose code development. Where GPP compilers tend to need only linear or quadratic optimization complexity, DSP compilers need more complex optimizations.

For example, an overhead in code execution time of 20% over the optimum code due to use of a compiler is less costly for a word processing software producer than for a producer of portable phones. The production price of the word processing software will be the same, though performance issues may affect sales. The portable phone, however, would have to be equipped with a 20% faster processor or additional hardware, which could even double its price. This is one of the reasons why different approaches to GPP and DSP compiler design and especially optimization are needed.

X. Quantitative Approach to DSP Compiler Design

In order to explore quantitative characteristics of DSP compilers the Institute for Integrated Systems in Signal Processing of the Aachen University of Technology started the DSPstone project in 1993 [25]. During this project a DSP-related benchmarking methodology was defined which should help in evaluating DSP compilers. The main goal was to get exact quantitative data about the overhead which is introduced if a high-level language and compiler are used for DSP code design. DSPstone also incorporates three suites of benchmarks (application, DSP-kernel, and HLL-kernel suite). More information can be found in [25].

We present below some more detailed results for the ADPCM application benchmark program and the Motorola DSP56000 family C compiler [1]. The analysis was performed to explore quantitative characteristics of DSP compilers and their use in DSP code design.

The ADPCM benchmark is a full, standard-compliant implementation of the ADPCM G.721 transcoder. The C language and the handwritten assembly versions are compared. For the DSP compiler the overhead in execution time was measured to be 510% and the overhead in program and data memory utilization to be 51% and 175% respectively. These results show that the main problem in using the DSP compiler is its high overhead in execution time. Our attempts to speed up the execution by recoding only the time-critical FMULT routine in assembly resulted in only a modest 9% improvement in execution time.

To provide a better insight into the behavior of DSP compilers and their interaction with the architecture we analyzed the dynamic instruction distribution (DID) of the code. These distributions show how frequently the instructions from specified instruction classes are used during program execution. Fig. 8 shows the DID for the ADPCM handwritten assembly code and Fig. 9 for the code obtained by compiling the C program. Five instruction classes were defined - move&transfer, logical, loop&control, jump, and arithmetic.

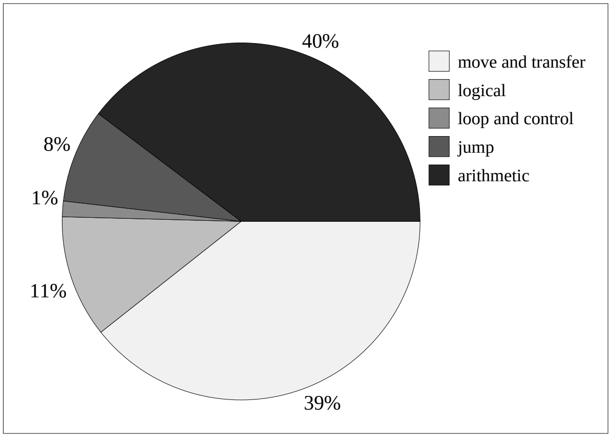

Figure 8: Dynamic Instruction Distribution for the ADPCM Handwritten Assembly Code.

The difference in distributions between the compiled and handwritten code for some instruction classes indicates that the inefficiency of the compiler does not influence all instruction classes equally. The percentage of move and transfer instruction in the overall code is much higher for the compiled than for the handwritten assembly code. We have concluded that one of the reasons for this behavior lies in the the compilation technique itself. The intermediate representation of the C code is translated into fragments of assembly instruc-

[1]Recently we repeated the same analysis for the Analog Devices ADSP2100 family C compiler and obtained very similar results

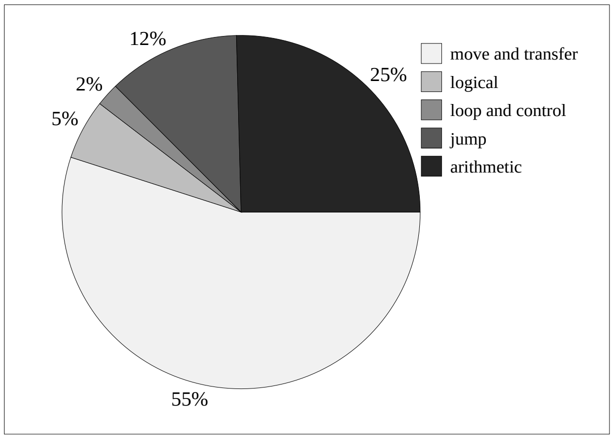

Figure 9: Dynamic Instruction Distribution for the ADPCM Compiled Code.

tions which are glued together using many register-to-register and memory-to-register move instructions.

The ability to execute more than one operation per instruction is one of the main characteristics of DSP processors. The percentage of parallel instructions during execution is computed to get more information about the use of parallel operations in typical applications like the ADPCM transcoder. In the assembly code approximately 25% percent of the executed instructions are of parallel type, where in the compiled code parallel instructions amount to less than 5%. Obviously the compiler has serious problems using parallel instructions. However, this result gives an even more important indication of the compiler's inefficiency. Let's suppose that the processor is able to execute at most six operations per instruction. If the only reason for the inefficiency of the compiler is its inability to use parallel instructions, the overhead in execution time would be at most 125%. Obviously this is much lower than the actual overhead of 510% which we measured. Our conclusion is that the analyzed compiler has additional weak points beyond instruction compaction.

Finally, we wanted to explore how much stack operations influence the efficiency of the code. As discussed earlier, most DSP processors do not provide sufficient support for software stack operations. Our measurements on the DSP compiler under test show that about 9% of the execution time and 11% of the instruction memory are used for stack operations. Although very important, stack manipulation is not the main problem of this compiler.

These examples show that the quantitative approach to compiler evaluation can deliver useful results. To improve the design process, additional quantitative analysis (e.g., register usage) and additional benchmark applications are necessary.

XI. Future of DSP Compilers

Summarizing the paper we can conclude that there are three areas where additional efforts are necessary to improve current DSP compilers:

- programming languages - language extensions, compilation directives, and flags are necessary to provide the compiler with all the information necessary to generate efficient code;
- compiler technology - specific applications and special-purpose architectures cannot be covered by standard, general-purpose compiler technology - new DSP-oriented compiler technology should be developed; and
- architecture - new DSP processor architectures should be developed with compilation problems in mind.

There is no doubt that the future of DSP programming belongs to compilers. However, it is unreasonable to expect that the compilers for existing DSP architectures will ever break the efficiency barrier completely. A more probable scenario is the improvement of new compilers coming with each new generation of DSP processors.

Following the trends in the design of general purpose architectures and compilers [26], we expect that new DSP architectures will be designed with compiler limitations in mind. Also, improvements in DSP compiler technology should help close the gap. Figure 10 presents the design flow and tools needed for a successful joint design of DSP architectures and compilers. The

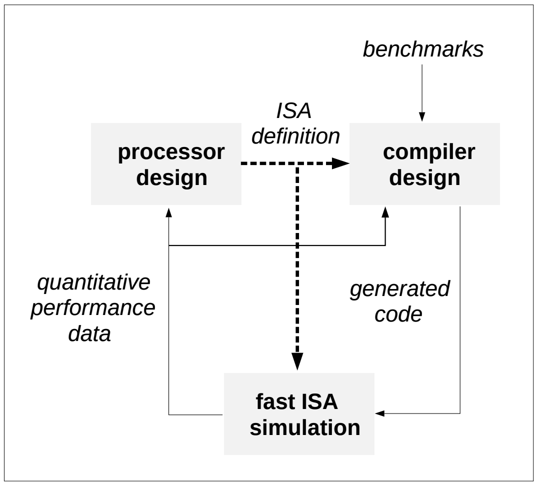

Figure 10: Processor-Compiler Co-Design.

main feature of the proposed design method is that it relies strictly on quantitative performance analysis.

Processor and compiler performance are measured on selected benchmarks using a fast instruction set architecture (ISA) simulator. The benchmarks are selected to represent the future field of application as close as possible. The ISA simulator can be adapted to architectural features of the processor and has an extensive support for statistical profiling and comparison of compiler and architecture versions. Computed performance results are used as feedback information for processor and compiler redesign. At the same time, new processor features are accounted automatically for in the ISA simulator and provided as a model to compiler design.

XII. References

[1] W. Hartung, S. Gay, and S. Haigh, "A practical C language compiler/optimizer for real-time implementations on a family of floating-point DSPs," in *Proc. of the ICASSP*, (New York), IEEE, Apr. 1988.

[2] S. Kafka, "An assembly source level global compacter for digital signal processors," in *Proc. of the ICASSP*, pp. 1061–1064, 1990.

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

[4] K. Leary and C. Cavigioli, "The ADSP-21020: An IEEE floating point and fixed point DSP for HLL programming," in *Proc. of the ICASSP*, pp. 1077–1080, 1991.

[5] K. Leary, "DSP/C: A standard high level language for DSP and numeric processing," in *Int. Conf. on Sig. Proc. Appl. and Tech.*, (Cambridge, MA), pp. 342–345, Nov. 1992.

[6] M. Hoffman, "Numerical C enhances coding of signal processing algorithms," *DSP Applications*, Dec. 1993.

[7] D. Syiek, "Challenging assembly code quality," in *Int. Conf. on Sig. Proc. Appl. and Tech.*, (Berlin, Germany), pp. 178–190, Nov. 1991.

[8] B. Harbison, "Uses and misuses of C++ in DSP application development," in *Proc. of the ICSPAT*, pp. 703–708, 1994.

[9] M. Blower, "Mapping C to DSP," in *Int. Conf. on Sig. Proc. Appl. and Tech.*, (Cambridge, MA), pp. 346–352, Nov. 1992.

[10] B. Krepp, "DSP-oriented extensions to ANSI C," in *Proc. of the ICSPAT*, pp. 695–702, 1994.

[11] B. Krepp, "A better interface to in-line assembly code," in *Proc. of the ICSPAT*, pp. 802–805, 1994.

[12] A. Aho, R. Sethi, and J. Ullman, *Compilers, Principles, Techniques and Tools.* Addison-Wesley, 1986.

[13] P. Papamichalis, J. Reimer, and J. Rowlands, "System and algorithm implementation techniques on the TMS320 family," *DSP & Multimedia Technology*, 1995. this issue.

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

[15] B. Kernighan and D. Ritschie, *The C Programming Language - ANSI C.* Prentice-Hall, 1988.

[16] P. Hilfinger, "A high-level language and silicon compiler for digital signal processing," in *Proc. of the Custom Int. Circ. Conf.*, pp. 213–216, 1985.

[17] P. Guernic, T. Gautier, M. Borgne, and C. Maire, "Programming real-time applications with SIGNAL," *Proc. of the IEEE*, vol. 79, pp. 1321–1336, Sep. 1991.

[18] Analog Devices, Inc., *ADSP-21000 Family: C Tools Manual*, 1993.

[19] Intermetrics, Inc., *77016 Family C Compiler: User's Manual*, 1994.

[20] S. Young, *Real-Time Languages: Design and Development.* John Wiley & Sons, 1982.

[21] R. Lipsett, "The Intertools DSP C compilers," *DSP & Multimedia Technology*, 1995. this issue.

[22] D. Fritz, "The PLC ANSI C compiler for the Zilog Z89C00 DSP," *DSP & Multimedia Technology*, 1995. this issue.

[23] K. Baudendistel, *Compiler Development for Fixed-Point Processors.* PhD thesis, Georgia Institute of Technology, 1992.

[24] S. Kim and W. Sung, "An autoscaling assembler for the TMS320C25," in *Int. Conf. on Sig. Proc. Appl. and Tech.*, (Santa Clara, CA), pp. 543–552, Oct. 1993.

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

[26] M. Tremblay and P. Tirumalai, "Partners in platform design," *IEEE Spectrum*, Apr. 1995.