EURASIP Journal on Applied Signal Processing 2002:9, 908–925
© 2002 Hindawi Publishing Corporation

Design and DSP Implementation of Fixed-Point Systems

Martin Coors

Institute for Integrated Signal Processing Systems, Aachen University of Technology, 52056 Aachen, Germany
Email: coors@iss.rwth-aachen.de

Holger Keding

Institute for Integrated Signal Processing Systems, Aachen University of Technology, 52056 Aachen, Germany
Email: keding@iss.rwth-aachen.de

Olaf Lüthje

Institute for Integrated Signal Processing Systems, Aachen University of Technology, 52056 Aachen, Germany
Email: luethje@iss.rwth-aachen.de

Heinrich Meyr

Institute for Integrated Signal Processing Systems, Aachen University of Technology, 52056 Aachen, Germany
Email: meyr@iss.rwth-aachen.de

Received 31 August 2001

This article is an introduction to the FRIDGE design environment which supports the design and DSP implementation of fixed-point digital signal processing systems. We present the tool-supported transformation of signal processing algorithms coded in floating-point ANSI C to a fixed-point representation in SystemC. We introduce the novel approach to control and data flow analysis, which is necessary for the transformation. The design environment enables fast bit-true simulation by mapping the fixed-point algorithm to integral data types of the host machine. A speedup by a factor of 20 to 400 can be achieved compared to C++-library-based bit-true simulation. FRIDGE also provides a direct link to DSP implementation by processor specific C code generation and advanced code optimization.

Keywords and phrases: fixed-point design, design methodology, data flow analysis, compiled simulation, code optimization.

1. INTRODUCTION

Digital system design is characterized by ever-increasing complexity that has to be implemented within reduced time, resulting in minimum costs and short time-to-market. This requires a seamless design flow that allows the execution of the design steps at the highest suitable level of abstraction.

For most digital systems, the design has to result in a fixed-point implementation, either in HW or SW. This is due to the fact that these systems are sensitive to power consumption, chip size, throughput, and price-per-device. Fixed-point realizations outperform floating-point realizations by far with regard to these criteria.

A typical fixed-point design flow is depicted in Figure 1. Algorithm design starts from a floating-point description that is analyzed by means of simulation without taking the quantization effects into account. This abstraction from all implementation effects allows an exploration of the algorithm space, for example, the evaluation of different digital receiver structures. This exploration is well supported by a variety of commercial block-diagram oriented system level design tools [1, 2, 3]. The modeling efficiency on the floating-point level is high and the floating-point models offer a maximum degree of reusability.

In a next step towards system implementation, a transformation to a bit-true representation of the system is necessary, that is, assigning a fixed word length and a fixed exponent to every operand. This process is quite tedious and error-prone if done manually: often more than 50% of the implementation time is spent on the algorithmic transformation [4] to the fixed-point level for complex designs once the floating-point model has been specified.

The major reasons for this bottleneck are as follows:

(1) There is no unique transformation from floating-point to fixed-point.

(a) Different HW and SW targets put different constraints on the fixed-point specification.

(b) Optimization for different design criteria, like throughput, chip size, memory size, or accuracy are in general mutually exclusive goals and result in a complex design

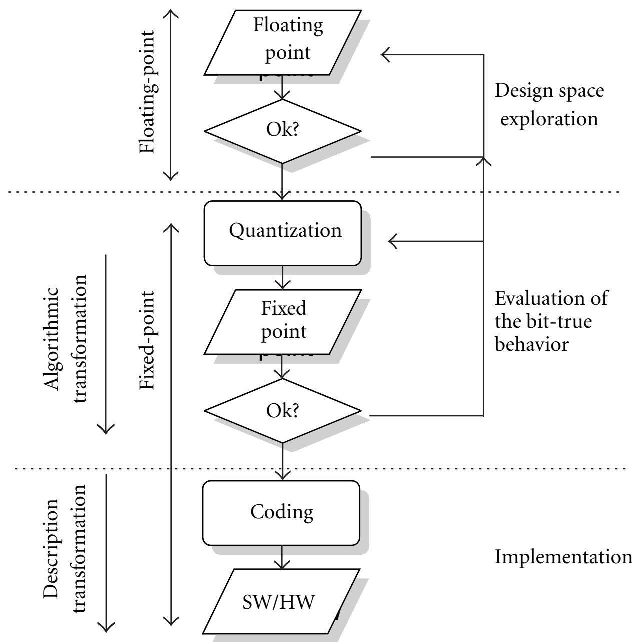


Figure 1: Fixed-point design process.

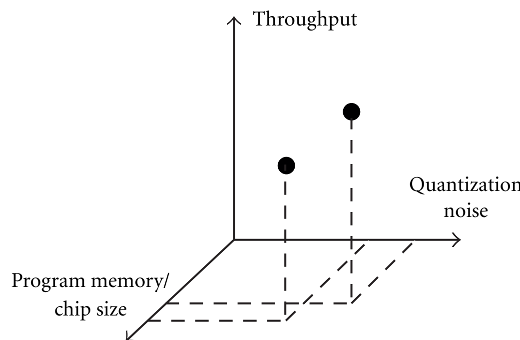


Figure 2: Fixed-point design space.

space as sketched in Figure 2. Furthermore, targets with a given datapath, for example, DSPs put different constraints on the quantization than ASICs where the datapaths are flexible.

(c) The quantization is generally highly dependent on the application, that is, on the applied stimuli.

(2) Quantization is a nonlinear process. Analytical models based on signal theory are only applicable for systems with a low complexity [5]. An exploration of the fixed-point design space with respect to quantization noise, performance, and operand word lengths cannot be done without extensive system simulation.

(3) Some algorithms are difficult to implement in fixed-point due to high signal dynamics or sensitivity to quantization noise. Thus algorithmic alternatives need to be employed.

Finally, the quantized system is implemented, either in hardware or in software on a programmable DSP. The implementation needs to be optimized with respect to chip area, memory consumption, throughput, and power consumption. Here the bit-true system-level model serves as a "golden" reference for the target implementation which yields bit-by-bit the same results.

To increase the designer's efficiency, software tool support for fixed-point design is necessary. Ideally the design environment would have the following features:

(1) A modeling language supporting generic fixed-point data types to model the fixed-point behavior of the system. It will also provide a means of data monitoring of variables and operands during simulation, for example, range, mean, and variance.

(2) A semiautomatic transformation from floating-point to a bit-true representation. The designer can bring in his knowledge about the system and he has full control over the transformation. The tool will accept a set of constraints specified by the designer to model the characteristics of the target hardware.

(3) The ability to perform bit-true simulation with a simulation speed close to floating-point simulation.

(4) A seamless design flow down to system implementation, generating optimized input for DSP compilers.

These requirements have been the motivation for the Fixed-point pRogrammIng and Design Environment (FRIDGE) [6, 7, 8], an interactive design environment for the specification, simulation, and implementation of fixed-point systems.

In this article we describe the principles and elements of FRIDGE and outline the seamless design flow as it becomes possible with this design environment. FRIDGE relies on five main concepts which are briefly introduced in the following.

1.1. Fixed-point modeling language

DSP system design is frequently done on a PC or a workstation utilizing a C/C++-based system-level design environment. For efficient modeling of finite word length effects, language extensions implementing generic fixed-point data types are necessary. ANSI C does not offer such data types and hence fixed-point modeling using pure ANSI C becomes a very tedious and error-prone task.

Fixed-point language extensions implemented as libraries in C++ [9, 10, 11] offer a high modeling efficiency. They supply generic fixed-point data types and various casting modes for overflow and quantization handling. The simulation speed of these libraries on the other hand is rather poor. Some of these libraries also offer data monitoring capabilities during simulation time.

In the FRIDGE design environment, the SystemC fixed-point data types are used for fixed-point modeling and simulation. A more detailed description of the SystemC fixed-point data types is given in Section 3.

1.2. Interpolative transformation

A central component of the FRIDGE design environment is the interpolative transformation from a hybrid description into a fully bit-true representation. The interpolative

transformation, which is presented in detail in Section 4 uses analytical range propagation to determine operand word lengths.

1.3. Data flow analysis

During the development of the FRIDGE design environment, we have identified a need for accurate data flow analysis. The published approaches for static and dynamic program analysis did not match the requirements of the design environment, thus we have developed a novel approach for control and data flow analysis, which is presented in Section 5.

1.4. Fast bit-true simulation

Existing C++-based simulation libraries model the fixed-point operands as objects and make extensive use of operator overloading and container data types. Also, for ease of use, many decisions are made during run time. These mechanisms increase the execution time of fixed-point simulations by one to two orders of magnitude compared to floating-point arithmetic. This makes the simulation run time a major bottleneck during the fixed-point design process.

In Section 7 various approaches for fixed-point simulation are presented and a methodology for fast bit-true simulation by mapping fixed-point algorithms in SystemC to an integer based ANSI C algorithm is introduced.

1.5. DSP target mapping

The final step in a float-to-fixed design flow is the implementation of the DSP system, either in hardware or in software. As a case study for targeting a high performance DSP, we have developed a FRIDGE back end which addresses the Texas Instruments TMS 320C62x fixed-point DSP processor and its C compiler. The back end generates target specific integer C code which exploits the features of the processor and the compiler to achieve a high efficiency of the compiled code. In Section 9 the FRIDGE C62x back end and the optimization strategies are presented.

2. THE FRIDGE DESIGN FLOW

The FRIDGE design flow starts from a floating-point algorithm in ANSI C. As illustrated in Figure 3, the designer then annotates *single* operands with fixed-point attributes. Inserting these *local annotations* results in a *hybrid* description of the algorithm, that is, some of the operands are specified bit-true, while the rest remain floating-point. A comparative simulation of the floating-point and the hybrid code within the same simulation environment shows whether the local annotations are appropriate, or if some annotations have to be modified. The integer word length of the local annotations can be derived from operand range monitoring during simulation runs. Typically, the designer manually annotates function parameters and key variables, for example, accumulator variables, which account for approximately 5% of all operands.

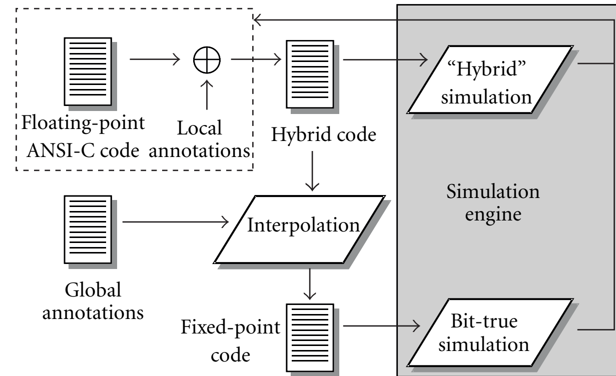


FIGURE 3: Quantization methodology with FRIDGE.

Once the hybrid program matches the design criteria, the remaining floating-point operands are automatically transferred to fixed-point operands by *interpolation*. Interpolation denotes the process of computing the fixed-point parameters of the nonannotated operands from the information that is inherent to the annotated operands and the operations performed on them. Additionally, the interpolator has to observe a set of *global annotations*, that is, default restrictions for the calculation of fixed-point parameters. This can be, for example, a default maximum word length that corresponds to the register length of the target processor.

The interpolation results in a fully annotated program, where each operand and operation is specified bit-true way. Cosimulating this algorithm with the original floating-point code will give an accuracy evaluation—and for changes now only the set of local and/or global annotations have/has to be modified, while the rest is determined and kept consistent by the interpolator.

Described above are the *algorithmic level* transformations as illustrated in Figure 1, that change the behavior or accuracy of an algorithm. The resulting completely bit-true algorithm in *SystemC* is not directly suited for implementation, thus it needs to be mapped to a target, such as, a processor's architecture or to an ASIC. This is an *implementation level* transformation, where the bit-true behavior normally remains unchanged. Within the FRIDGE environment, different *back ends* map the internal bit-true specification to different formats/targets, according to the purpose or goal of the quantization process.

3. FIXED-POINT DATA TYPES AND LOCAL ANNOTATIONS

Since ANSI C offers no efficient support for fixed-point data types [12, 13], we initially developed the fixed-point language *fixed-C* [14] that is a superset of the ANSI C language. It comprises different generic fixed-point data types, cast operators, and interpolator directives. The *fixed-C* language was licensed to Synopsys, Inc., and Synopsys contributed it as a set of additional fixed-point data types to the *Open SystemC*

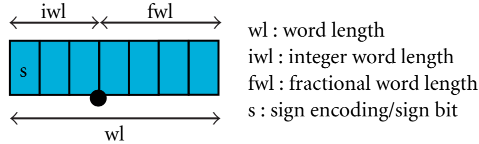


FIGURE 4: Fixed-point attributes of a bit-true description.

Initiative (OSCI) [11]. Together with additional fixed-point language elements from the *A|RT Library* by Frontier Design Inc., [10] *fixed-C* has been the base for the development of the SystemC fixed-point data types that are now used in the FRIDGE project as well.

The SystemC fixed-point data types are utilized for different purposes in the FRIDGE design flow:

- Since ANSI C is a subset of SystemC, the additional fixed-point constructs can be used as bit-true annotations to dedicated operands of the original floating-point ANSI C file, resulting in a *hybrid* specification. This partially fixed-point code can be used for simulation or as input to the interpolator.
- The bit-true output of the interpolator is represented in SystemC as well. This allows a maximum transparency of the results to the designer, since the changes to the code are reduced to a minimum and the effects of the designer's directives, such as local annotations in the hybrid code, become directly visible.

The additional fixed-point types and functions are part of a C++ class library that can be used in any design and simulation environment that are based on or can integrate C or C++ code (see, e.g., [1, 2, 3].)

For a bit-true and implementation independent specification of a fixed-point operand, a three-tuple is necessary: the *word length wl*, the *integer word length iwl*, and the *sign s*, as illustrated in Figure 4.

For every fixed-point format, two of the three parameters *wl*, *iwl*, and *fwl* (fractional word length) are independent; the third parameter can always be calculated from the other two, wl = iwl + fwl.

With a given sign encoding *s*, we can also compute the minimum and maximum value that the fixed-point format `<wl,iwl>` can hold. For example, for a two's complement (tc) signed representation the minimum and maximum compute to

$$\max_{\langle \mathrm{wl,iwl,tc} \rangle} = 2^{\mathrm{iwl}-1} - 2^{\mathrm{fwl}}, \quad \min_{\langle \mathrm{wl,iwl,tc} \rangle} = -2^{\mathrm{iwl}-1}. \tag{1}$$

For an unsigned representation (us), on the other hand, the minimum and maximum are

$$\max_{\langle \mathrm{wl,iwl,us} \rangle} = 2^{\mathrm{iwl}} - 2^{\mathrm{fwl}}, \quad \min_{\langle \mathrm{wl,iwl,us} \rangle} = 0. \tag{2}$$

Note that an *integral* data type is merely a special case of a fixed-point data type with an iwl that always equals wl—hence an integral data type can be described by two parameters only, the word length *wl* and the sign encoding *s*.

In the following sections, we provide a short overview of the most frequently used fixed-point data types and functions in SystemC. A more detailed description can be found in the SystemC users manual [11].

3.1. The data types sc_fixed and sc_ufixed

The two's complement data type `sc_fixed` and the unsigned data type `sc_ufixed` receive their format when they are declared, that is, the fixed-point attributes must be known at compile time (static arguments),

```
sc_fixed<wl,iwl>   d,*e,g[8];
sc_ufixed<wl,iwl>  c;
```

Thus they behave according to these fixed-point parameters throughout their lifetime. This concept is called *declaration time instantiation* (DTI). Similar concepts exist in other fixed-point languages as well [9, 10, 15]. Pointers and arrays, as frequently used in ANSI C, are supported as well.

For every assignment to a DTI variable, a data type check is performed. If the left-hand data type does not match the right-hand data type as illustrated in the code example below, an implicit cast to the left-hand data type becomes necessary,

```
sc_fixed<6,3>      a,b;
sc_ufixed<12,12>    c;
a = b; /* correct, both types match */
c = b;
/* type mismatch -> implicit cast necessary */
```

The data types `sc_fixed` and `sc_ufixed` are the data types of choice, for example, for interfaces to other functionalities or for lookup tables, since they behave like a memory location of a specific length and a known embedding/scaling.

3.2. The data type sc_fxval

Additionally to the DTI data type concept, SystemC provides the *assignment time instantiation* (ATI) data type `sc_fxval`. This type may hold fixed-point numbers of arbitrary format and is especially tailored for the float-to-fixed transformation process. A declaration of a variable of type `sc_fxval` does not specify any fixed-point attributes and if subsequently in the code a fixed-point value is assigned to a `sc_fxval` variable, the variable is (re-)instantiated with all fixed-point attributes of the assigned value.

3.3. The data types sc_fix and sc_ufix

Along with the static attribute types `sc_fixed` and `sc_ufixed`, SystemC also provides the fixed-point types `sc_fix` and `sc_ufix` that may also take nonstatic fixed-point attributes such as variables. The function in the code example below has the word length `wl` and the integer word length `iwl` as formal parameters, that is, `wl` and `iwl` are not known at compile time.

```
sc_fxval cast_func(int wl, int iwl, sc_fxval in)
{
return sc_fix(in,wl,iwl);
}
```

As shown in this example, the constructor for the types `sc_fix` and `sc_ufix` are often used to cast a value to a different fixed-point format.

3.4. Cast modes

For a cast operation to a fixed-point format `<wl,iwl,sign>`, it is also important to specify the overflow and precision reduction in case the target data type cannot hold the original value:

```
a = sc_fix(input,wl,iwl,q_mode,o_mode);
```

The variable `a` holds a two's complement fixed-point format `<wl,iwl>` and the value of `input` is cast to this fixed-point data type according to the quantization mode `q_mode`[1] and the overflow mode `o_mode`.[2] The most important casting modes are listed below. SystemC also specifies many additional cast modes to model target specific behavior.

Quantization modes

Truncation (`SC_TRN`). The bits below the specified LSB are cut off. This quantization mode is the default for SystemC fixed-point types and will be used if no other value is specified.

Rounding (`SC_RND`). Adds LSB/2 first, before cutting off the bits below the LSB.

Overflow modes

Wrap-around (`SC_WRAP`). In case of an overflow the MSB carry bit is ignored. This overflow mode is the default for SystemC fixed-point types and will be used if no other value is specified.

Saturation (`SC_SAT`). In case the minimum or maximum value is exceeded the result is set to the minimum or maximum value, respectively.

With the `sc_fxval` type, every assignment to a variable *overwrites* all prior instantiations, that is, one `sc_fxval` variable may have different context-specific bit-true attributes in the same scope. This concept of ATI is motivated by the specific design flow: transformation starts from a floating-point program, where the designer abstracts from the fixed-point problems and does not think of a variable as finite length register.

The concept of local annotations and ATI is also an effective way to assign context specific information without changing structures or variables when exploring the fixed-point design space.

[1]The quantization handling specifies the behavior in case of a word length reduction at the LSB side.

[2]The overflow handling specifies the behavior in case of a word length reduction at the MSB side.

4. INTERPOLATION

The interpolator with its control and data flow analyzer is the core of the FRIDGE design environment. As depicted in Figure 3 it determines the fixed-point formats for all operands of an algorithm, taking as input a user annotated *hybrid description* of the algorithm and a set of global default rules, the *global annotation* file. Hence *interpolation* describes the computation of the fixed-point parameters of the non-annotated operands from the information that is inherent to the annotated operands.

The interpolative concept is based on three key ideas:

(1) *Attribute propagation*. The method of using the attributes of the bit-true specified operands in the code to calculate bit-true attributes for the remaining operands and operations in the code.

(2) *Global annotations*. The description of default rules and restrictions for attribute propagation.

(3) *Designer support*. The interpolator supplies feedback and reports to assist the designer to debug or improve the interpolation result.

For a better understanding the first two points are explained more detailed in the following.

(1) *Attribute propagation*. Given the information of the fixed-point attributes of some operands, the type and the fixed-point format of other operands can be extracted from this information. For example, if for the inputs to an operation both the range and the relevant fractional word length are specified, the same attributes can be determined for the result.[3]

Consider the following line of code:

```
c = a + b;  d = 1.5;  e = c * d;
```

The corresponding data flow graph is depicted in Figure 5. We assume that the ranges and the precision of the variables `a` and `b` are known, for example, by user annotations:

$$\begin{aligned} a \in [-0.25, 0.75] &\Longrightarrow R_a = [-0.25, 0.75]; \quad \mathrm{fwl}(a) = 2, \\ b \in [-1.25, 0.5] &\Longrightarrow R_b = [-1.25, 0.5]; \quad \mathrm{fwl}(b) = 2. \end{aligned} \tag{3}$$

To receive the range R_c for the variable `c` that contains the sum of the variables `a` and `b` we add the ranges R_a and R_b (a detailed description of the range arithmetic used here can be found in [14]),

$$R_c = R_a + R_b = \left[\min_a + \min_b, \max_a + \max_b\right] = [-1.5, 1.25]. \tag{4}$$

The precision P_c (fwl) for the sum `c` computes to the maximum of the precisions P_a and P_b,

$$P_c = \max(P_a, P_b) = 2. \tag{5}$$

The information on the range and on the precision of the variable `c` is sufficient to calculate the required word length

[3]An exception is the division, where the accuracy of the operation must be specified as well.

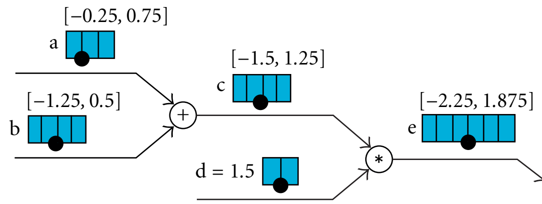


Figure 5: Example for interpolation of ranges/word lengths.

or integer word length for c. The correlation between fwl, range, and iwl yields the iwl of c:

$$\begin{aligned} \mathrm{iwl}_c &= \left\lceil \max\left(\log_2 \left|\min_c\right|, \log_2\left(\left|\max_c\right| + 2^{-\mathrm{fwl}_c}\right)\right) + 1\right\rceil \\ &= \lceil \max(0.58, 0.58) + 1\rceil = 2. \end{aligned} \tag{6}$$

Thus the resulting format for c is <4,2,tc>, where tc indicates the two's complement representation of c.

The next step for the interpolator is to compute the fixed-point format of the constant d. Since the range of d is $R_d = [1.5, 1.5]$ and the precision is $P_d = \mathrm{fwl}_d = 1$ the iwl of d can be calculated as

$$\mathrm{iwl}_d = \left\lceil \log_2\left(\max_d + 2^{-\mathrm{fwl}}\right)\right\rceil = \lceil \log_2(1.5 + 0.5)\rceil = 1. \tag{7}$$

After all fixed-point parameters of the input operands to the multiplication `e = d * c` are known to the interpolator, it continues with the calculation of the bit-true format and parameters for the variable e:

$$\begin{aligned} R_e &= R_c * R_d = [-1.5, 1.25] * 1.5 = [-2.25, 1.875], \\ P_e &= P_c + P_d = 2 + 1 = 3 \Longrightarrow \mathrm{iwl}_e \\ &= \left\lceil \max\left(\log_2 \left|\min_e\right|, \log_2\left(\left|\max_e\right| + 2^{-\mathrm{fwl}_e}\right)\right) + 1\right\rceil \\ &= \lceil \max(1.17, 1) + 1\rceil = 3. \end{aligned} \tag{8}$$

Hence we receive a fixed-point format of <6,3,tc> for the variable e.

Note that this is a rather conservative way of interpolation, bits that may contain any information are never discarded. For the MSB side this is called a *worst case* interpolation, since with the iwl calculated by the interpolator an overflow is impossible, while on the other hand it may lead to iwls much larger than actually needed. In this case the designer may add additional local annotations to cut back the iwl to a more suited value. For the LSB side this is called *maximum precision* interpolation (MPI) interpolation, that is, by default every LSB of the operands is kept, maintaining the highest possible accuracy. LSBs are only discarded if the word length exceeds the maximum word length specified in the *global annotation* file. This can lead to a large increase in the (fwl), but with additional local annotations the designer can also keep the *fwl* shorter. In [6] we also describe a method to have the interpolator calculate a less conservative value for the *fwl*.

(2) *Global annotations.* While local annotations express fixed-point information for single operands, the global annotations describe default restrictions to the complete design. For different targets, different global restrictions apply. For SW, the functional units to perform specific operations are already defined by the architecture of the processor. Consider a 16 × 16 bit multiplier writing to a 32-bit register. A global annotation can supply the information to the interpolator that the word length of a multiplication operand must not exceed 16 bits, while the result may have a word length of up to 32 bits.

4.1. Implementational issues

In a first step the FRIDGE front end parses in the hybrid description into a C++-based intermediate representation (IR). Then range propagation is performed to determine the bit-true format for all the operands. During this process, control and data flow analysis is also carried out. The information gained is stored in the IR. The advanced algorithms used for the analysis will be described in Section 5.

After this process the IR holds a bit-true description of the algorithm with additional control and data flow information. These data structures form the basis for additional transformation steps performed in the FRIDGE back ends that target different languages and platforms.

5. ADVANCED DATA FLOW ANALYSIS

During the development of the FRIDGE design environment, we have identified a need for accurate data flow analysis to cater the needs of the interpolation, the fast simulation code generation and the target specific code optimization. The published methods were not capable of matching the requirements, thus we have developed a novel approach for data flow analysis that can provide the necessary data for the FRIDGE back ends.

Researchers have worked on program analysis techniques since the 1960s and there is, by now, an extensive literature [16]. There are two major approaches to program analysis:

(a) There are static analysis techniques that analyze the program code at compile time. Usually, sets of equations are set up according to the program semantics and solved by finding their fixpoint. One of the best known static approaches is *Data Flow Analysis*. It is treated in depth in standard compiler books [17, 18]. Other techniques such as *constraint-based analysis* and *abstract interpretation* are also described in [19]. PAG [20] is a tool for generating interprocedural data flow analyzers that implement these techniques.

(b) On the other hand, there are techniques for dynamic analysis that are used for examining the behavior of program code during execution. Typically, these techniques are employed by profiling tools. Profiling information can for example be used by programmers to find critical pieces of code or as input to profile-driven optimizers. Dynamic program analysis techniques have been implemented in tools like *Pixie* [21] or *QPT* [22]. By principle, dynamic program analysis relies on input vectors to be

processed during execution. Thus the results are of no general nature.

Analysis techniques of neither category are suited for the needs of the FRIDGE design environment. Static analysis puts tight constraints onto the code to be analyzed. The use of pointers is usually not supported or yields too conservative results. Implementations of digital signal processing systems usually make extensive use of pointers, even, for example, for iterating over data arrays. Furthermore, static analysis is blind for program properties that result from run time effects. However, especially these properties have to be taken into account by FRIDGE in order to obtain precise results.

Dynamic analysis is to some extend capable of detecting these properties. Nevertheless, it is not applicable for the FRIDGE design environment for two reasons. First, the results are of statistical, numerical nature. There is no way to gain information about data flow or control flow properties. Second, the results are not generally valid, that is, they only reflect the behavior of the program running on the given input vectors. FRIDGE requires analysis results that are valid for all possible executions of the program though.

The requirements for the analysis employed by FRIDGE are different from those of standard tools like, for example, a general purpose compiler. FRIDGE is focused on digital processing systems. These systems are typically data flow dominated, that is, their execution is to a great extent independent from the data to be processed. Besides, the accuracy and quality of the results are more important than speed (of analysis). This allows for a more comprehensive code analysis than, for example, a general purpose compiler can apply. In order to gain precise results including also run time properties and being able to handle pointer operations, the code is interpreted. Since there is no concrete data to be processed, we process abstract data instead. In the following this methodology is referred to as *abstract execution*.

The data flow analysis unit in the FRIDGE design environment is based on three main components:

(1) The concept of data abstraction.
(2) The state controlled memory model.
(3) The concept of coupled iterators.

5.1. Data abstraction

While in concrete execution numeric values are written to and read from memory, we use *operations* for abstract execution. An *operation* is a collection of information about possible values. The two most important elements are

(1) the range, that is, the minimum value and the maximum value, and
(2) a reference to the expression in the code that corresponds to the operation.[4]

[4]This is for gaining data flow information.

Furthermore, operations may be ambiguous. Consider the code example below.

```
01 int func(int x, int y, int z){
02   int a, b, c, d;
03
04   switch(y){
05     case 1:
06       a = 8; break;
07     case 2:
08       a = 16; break;
09     case 3:
10       a = 32;}
11
12   if(z>0)
13     b = 0;
14   else
15     b = 1;
16
17   if(x>0){
18     c = 5;
19     d = a;}
20   else {
21     c = b;
22     d = 7;}
23
24   return c + d;
25 }
```

The only information available about parameters `x`, `y`, and `z` is that they are integers. Hence it cannot be decided which branches of the `switch`- and `if`-statements in lines 04, 12, and 17 are executed. This results in an ambiguous content, for example, of variable `b`, namely, values `0`[5] and `1`, referring to the expressions in lines 13 and 15, respectively. We combine both *operations* to an *ambiguous operation*. In addition, *ambiguous operations* are associated with conditions, under which the alternatives are chosen. In the example, alternative `0` is chosen if `(z > 0)` is `true`, alternative `1` if it is `false`. In general, there may be more than two alternatives and conditions may be combined by a logical `AND`.

Operations are arranged in graphs similar to binary decision diagrams introduced by Akers [23], where the nodes embody the *ambiguous operations* and the leafs the *unambiguous operations*.

In general, operations are described by the following rules:

(i) an *operation* is either an *unambiguous operation* or an *ambiguous operation*;
(ii) an *unambiguous operation* represents a possible content in memory during concrete execution of a program;

[5]When talking about a value, we mean an operation with a range degenerated to a value.

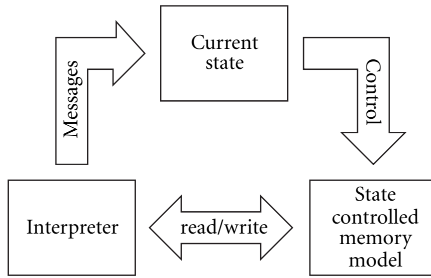


FIGURE 6: Abstract execution.

(iii) an *ambiguous operation* is associated with a control flow ambiguity in the code (dashed line in Figure 7) and matches each possible branch to an *operation*.

Thus these trees do not only contain the alternatives, but also the conditions under which the alternatives are taken. The conditions are determined by all the ambiguities along the path from the root to the alternative. Each ambiguity contributes to the condition in this way, that the condition for the execution of the control flow branch must be fulfilled, that is associated with the link to the next *operation* on the path. A logical AND is applied to the contributions of each ambiguity.

For example, the tree in Figure 7 with A_3 as its root shows the ambiguity tree corresponding to variable `d` in line 24. The path to *value* 32 (bold line) goes through ambiguities A_3 and A_4. A_3 is associated with the `if`-statement and the path follows the link that is associated with the `true`-branch. That yields the condition `(x > 0) == true`. Further on, the path passes through A_4 and follows the link to 32. A_4 is associated with the `switch`-statement and the link to 32 with `case 3`. That yields the condition `y == 3`. Thus the resulting condition for A_3 taking on the *value* 32 is[6]

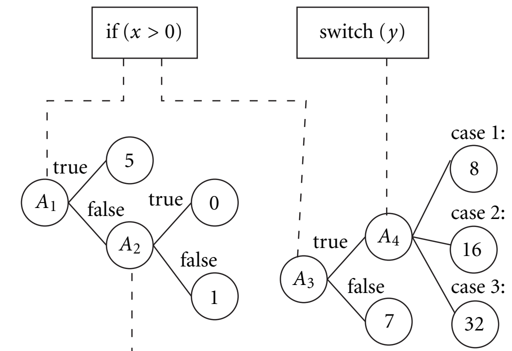
```
(x > 0) == true && y == 3
```

5.2. The state controlled memory model

As illustrated in Figure 6, the *state controlled memory Model* serves as a regular memory that can be read and written to. Besides, it is responsible for building the ambiguity trees described in Section 5.1.

As long as the *current state* is in initial state, the behavior of the state controlled memory model does not differ from a regular memory. Once the *current state* contains a condition, all changes done to memory contents only occur under that condition and result in appropriate ambiguity trees. The state is defined by a set of assumptions about the result of particular expressions in the code. A logical AND is performed on these assumptions. The initial state makes no assumptions at all. Other valid states could for example be "`(x > 0) == true`" or "`(x > 0) == true && y == 3`." During abstract execution, the state can be changed by the *interpreter*.

[6]This notation is according to C syntax.



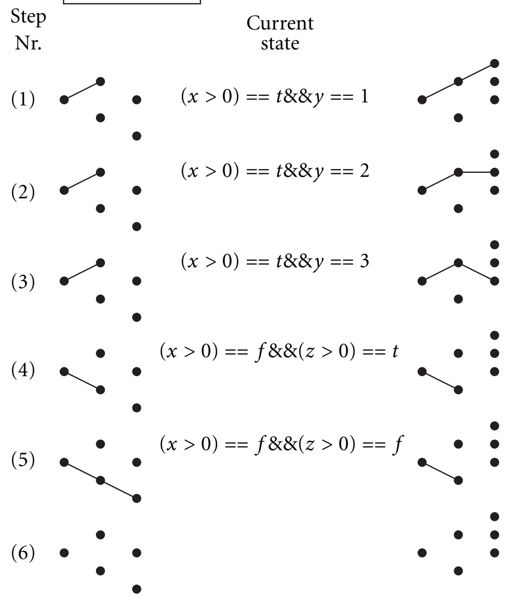


FIGURE 7: Iterating over ambiguities.

5.3. Iterating over ambiguities

When abstractly executing statements (Section 5.4) or computing the set of all possible evaluations of an expression,[7] We have to iterate over the alternatives of ambiguities. This is basically done by traversing the corresponding tree. However, the *current state* is taken into account, that is, only those alternatives are *visible*, whose conditions are not contradictory to the *current state*. Furthermore, when selecting an alternative from an ambiguity, the corresponding conditions are—if not yet included—added to the *current state*. This way, the following is achieved: All data couplings are taken into account, that is, no impossible cases are considered. Alternative executions of statements can be done without further thought about the *current state* (see Section 5.4).

Selecting an alternative from an ambiguity is done by building a path through the corresponding tree. The end of the path is an *unambiguous operation*. In principle, iterating

[7]For example, this is done when computing fixed-point parameters of an expression.

is performed on all successors of an ambiguity first, until it will be iterated over the alternatives of the ambiguity itself (depth first). When establishing a path through an ambiguity, two basic cases have to be considered:

(1) The *current state* contains a condition respective to the control flow fork that is associated with the ambiguity. In this case, the path *must* follow the link that corresponds to the condition and may not be altered. The node would be considered a *slave* node.

(2) The *current state* does not yet contain a condition respective to the control flow branch that is associated with the ambiguity. In this case, a possible branch is selected and the path is extended by the corresponding link. The corresponding condition is added to the *current state*. The node would be considered a *master* node. During further iteration, the path will switch to all other links successively. When this is done, the respective condition has to be updated accordingly. After that, the condition is removed from the *current state*.

The trees in Figure 7 show the contents of variables `c` (left-hand side) and `d` (right-hand side) connected to line 24 in the code. Figure 7 also illustrates how to iterate over all possible combinations of contents of both variables. Note how building a path through an ambiguity affects the *current state* and how the *current state* masks the visible alternatives of ambiguities. First of all value `5` is selected from ambiguity A_1. The corresponding condition `((x > 0) == true)` is added to the *current state*. Thus A_1 becomes a master node. When building the path through A_3, A_3 becomes a slave node, because the *current state* already makes an assumption about the control flow ambiguity that is associated with A_3 `((x > 0))`. Therefore, the path *must* follow the link from A_3 to A_4. Nodes A_2 and A_4 are associated with different control flow forks, respectively. They always become master nodes and never affect any other ambiguities. Steps 2 and 3 iterate over the remaining visible alternatives of the right-hand tree. Step 4 switches to the second alternative of master node A_1 (`false`). This affects the slave A_3 in this way as long as the path in the left-hand tree goes from A_1 to A_2 (steps 4 and 5), the only visible alternative of the right-hand tree is `7`. In step 6 the iteration has been completed.

5.4. *Execution of a program*

Figure 8 shows how statements are abstractly executed. The solid lines represent the control flow of a concrete execution. Abstract execution also follows that control flow. However, statements that depend on ambiguous data are executed multiple times (dashed lines), once for every possible vector of the involved ambiguities. The vectors are iterated over as described in Section 5.3. Thus every execution is performed in a different *current state*, such that changes in memory together with their corresponding states are stored in ambiguity trees. This algorithm is applied recursively for nested statements. Any code constructs can be executed this way.

Although a possibly large number of execution states exists, we found that the run time and the memory consumption of the analysis were remarkably low for typical signal processing algorithms. In most cases the control and data flow analysis was performed in less than one second on a 800 MHz PC.

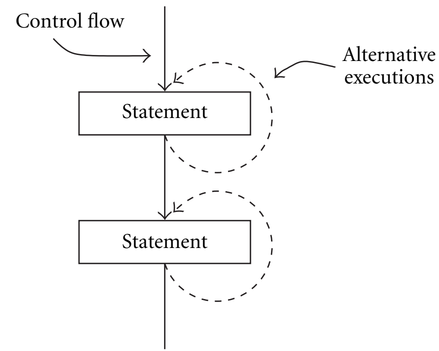


Figure 8: Abstract executions of sequential statements.

The information gained during abstract execution is stored in the intermediate representation of the algorithm. The FRIDGE back ends, which will be introduced in the next sections, access this information to perform several code transformation steps.

6. FAST BIT-TRUE SIMULATION

As pointed out in Section 1, transforming a signal processing algorithm from a floating-point to a fixed-point requires extensive simulations due to the nonlinear nature of the quantization process. The available C++-based fixed-point libraries [10, 11] offer a high modeling efficiency but the simulation speed of these libraries on the other hand is rather poor. This makes simulation speed a major bottleneck in the fixed-point design process.

Utilizing C-based fixed-point libraries like the ETSI basic arithmetic operations [24] does not overcome this problem as the simulation speed still has a considerable overhead compared to an equivalent floating-point implementation.

Existing C++-based simulation libraries model the fixed-point operands as objects. In order to offer generic fixed-point data types without word length restrictions, data container types are used as an internal representation. Bit-true operations are performed by operator overloading. Range checking, the choice of cast modes and many other decisions necessary for correct bit-true behavior are done at simulation time. The price for this flexibility and ease of modeling is slow execution speed as the generic fixed-point data types modeled by extensive C++ constructs cannot be efficiently mapped to the architecture of the host machine by today's C++ compilers.

A simulation speedup can be achieved by mapping the fixed-point operands to the mantissa of the floating-point hardware of the host machine and bit level manipulations to maintain bit-true behavior. This restricts the maximum word length of the fixed-point operands to the word length of the mantissa. This approach has been described by Kim et al. [25] and it is also implemented in the SystemC library [11].

Another mean of speeding up fixed-point simulations is the use of a hardware accelerator, for example, an FPGA to perform computationally expensive operations. The acceleration can be achieved either by utilizing configurable logic or by combining configurable logic with a processor. This approach has been described by De Coster [26]. The mapping of the algorithm to the different hardware units and the data transfer between the units make additional transformation steps necessary.

The work described in this article proposes a mapping of fixed-point algorithm in SystemC to an integer-based ANSI C algorithm that directly addresses the built-in integer ALU of the host machine. An efficient mapping includes an embedding of all fixed-point operands into the host machine registers, a cast mode optimization and many other aspects, and requires a detailed control and data flow analysis of the algorithm. Independently from the authors' work, De Coster [26] proposed a similar method, using DFL [27] as input language and targeting directly a Motorola DSP65000.

Our work presented here represents a continuation of the research results published by Keding et al. [6] and Willems [14] and introduces improved concepts for the mapping process that result in a considerable simulation acceleration.

For the fast simulation back end we assume that fixed-point attributes are assigned to every operation. The back end also requires the information collected during the control and data flow analysis stored in the IR. After a number of IR refinements, an ANSI C representation of the algorithm using only integral data types can be derived from the IR. It is important to note that the transformation in the back end, in contrast to the float-to-fixed transformation in the IR, does not change the behavior of the algorithm. The fully quantized algorithm coded in *SystemC* and the integer-only ANSI C algorithm yield bit-by-bit identical results, making the fast simulation back end output ideally suited for fast bit-true simulation on a workstation or PC.

7. TRANSFORMATION TO ANSI C

7.1. The lbp alignment

For the embedding of a fixed-point operand specified by a triple *(wl, iwl, sign)* into a register of the host machine with the machine word length (*mwl*) the minimum requirement is

$$\text{mwl} \geq \text{wl} = \text{iwl} + \text{fwl}. \tag{9}$$

Figure 9 illustrates different options for embedding an operand with a word length of 5 bit into a given mwl of 8. Obviously, for mwl > wl, a degree of freedom for choosing the location of binary point (lbp) exists:

$$\text{mwl} - \text{iwl} \geq \text{lbp} \geq \text{wl} - \text{iwl} = \text{fwl}. \tag{10}$$

Beside this degree of freedom, there are also a number of constraints for the selection of the lbp:

(i) *Interface constraints*. For interface elements, such as, function parameters or global variables, the lbp must be defined identically for a function and all calls to this function. Otherwise, the data written to or read from these data elements will be misinterpreted.

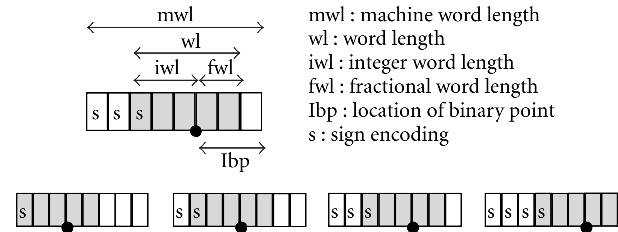


Figure 9: Embedding a 5-bit word into an 8-bit register.

(ii) *Operation constraints*. Each operation has an lbp *syntax*. This lbp *syntax* may include constraints on the lbp of the operand(s) of the operation and/or rules for the calculation of the lbp of the result. For example, the operands and the result of and addition must have the same lbp.

(iii) *Control and data flow constraints*. Generally, a read access to a storage element must use the same lbp as the preceding write access to the storage element. This implies that if a write operation to a memory location occurs in alternative control-flow branches, the lbp must be at the same position in both write operations, as no run time information about the lbp is available in a following read operation. The same applies to ambiguous write operations to arrays and write operations via pointers.

7.1.1 The lbp alignment algorithm

The lbp alignment algorithm implemented in the fast simulation back end is designed to take advantage of the degree of freedom described by (10), while meeting the constraints specified above. Meeting these constraints and maintaining the consistency of the lbps require precise information about the control and data flow of the algorithm. To obtain this information we used the data flow analysis method described in Section 5. The data flow information is represented basically as *define-use (du) chains* and *use-define (ud) chains* [17, 18], with additional and more accurate information about ambiguous control flow.

Initially, for all operands lbp = fwl is chosen. Thus all operands are *right aligned*. In a first step we set the lbps of all interface elements according to the *interface constraints*.

Then, in an iterative process, the data flow information is used to adjust the lbps by insertion of shift operations to meet the *operation constraints* and the *control and data flow constraints*. The algorithm terminates when all conditions are fulfilled and the lbps did not change during the last iteration.

The operation constraint lbp alignment algorithm basically consists of an iteration over all operations and an adjustment of the operand and result lbps according to the operation's lbp *syntax*.

The control and data flow constraint lbp alignment algorithm searches for all read accesses from a data element the associated previous write accesses to the same data element, that is, finding all *defines* for a *use* of a data element (ud-

chains). According to the control and data flow constraints the lbp of operands linked by such ud-chains are set to the same value.

Finally, the embedding of *constants* can be done in a way that the required shift operations when using the constant are minimized.

Unlike described by Kum et al. [28], we do not use a shift operation minimizing approach here, but using the degree of freedom in choosing a suited lbp (10) and the accurate data flow information, we found that there is not sufficient potential for this optimization to justify the effort.

7.2. Data type selection

The next step in the transformation process is the selection of suitable integral data types for fixed-point variables. The FRIDGE internal bit-true specification of the algorithm features arbitrary word lengths. With the SystemC back end this does not represent a problem, since the SystemC data types are generic and may be of any bit length required. With the fast-simulation back end, on the other hand, we only have the limited pool of the built-in data types of the host machine, that is, integral data types like `char`, `short`, `int`, `long`.

7.2.1 Basic constraints for any data element

A matching data type for every fixed-point variable has to be chosen. The minimum requirement for the data type chosen is that it can be embedded into the host machine data type with word length mwl at the correct location, (see Figure 9 for illustration) $\text{iwl} + \text{lbp} \leq \text{mwl}$.

7.2.2 Structural constraints

Additionally, the requirements introduced by data structures that force each of their elements to be of the same data type have to be met. An example for this behavior are arrays. The target data type for the N elements of an array must fulfill the following condition: $\max_{i=0}^{N-1}(\text{iwl}_{\text{array}}[i] + \text{lbp}_{\text{array}}[i]) \leq \text{mwl}$.

7.2.3 Semantical constraints

Another constraint becomes important if aliasing of data elements, for example, by pointers occurs: a pointer may point to different data elements. For syntax and semantics reasons all aliased data elements and the base type of the pointer must be identical [13]. This only causes a problem if data types are changed like it is done in fixed-point optimizations or the floating-point to fixed-point transformation process described in Section 2: initially, most numerical data types are floating-point types but after the transformation there are various different fixed-point data formats. Hence special care must be taken during the code generation process to ensure that the types are consistent. A detailed description of the data type selection algorithm used can be found in [29].

7.3. Cast mode transformation

Cast operations can reduce or limit the word length on the MSB side of a word (*overflow handling*) or at the LSB side of a word (*quantization handling*). They are used either to prevent indeterministic behavior of fixed-point systems[8] or to model a data path that is different from the host machine. This is often the case when algorithms for DSP systems are developed. Fixed-point libraries like in SystemC offer various generic overflow and quantization handling modes, which makes SystemC an efficient means of modeling fixed-point systems. For fast fixed-point simulation, on the other hand, the use of these generic casting modes are simply ruled out for performance reasons.

7.3.1 Overflow handling

Overflow handling is required if it is necessary to reduce the wl at the MSB side of the word or if the carry bit is set for the MSB. Examples for frequently used overflow handling modes in digital signal processing algorithms are *wrap-around* and *saturation* [30].

Saturation

In SystemC, a cast of an expression `expr` to a *wl*-bit tc data type with integer word length *iwl* applying saturation as overflow mode can be modeled as follows:

```
result = sc_fix(expr,wl,iwl,...,SC_SAT);
```

The fast simulation code generation on the other hand translates this into plain C code that first tests if the range of data type is exceeded, and if so it sets the resulting value to the minimum or maximum of this type, which is

$$\begin{aligned} \underset{\text{wl,iwl,lbp,tc}}{\text{MAX}} &= 2^{\text{iwl}+\text{lbp}-1} - 2^{\text{lbl}-\text{fwl}}, \\ \underset{\text{wl,iwl,lbp,tc}}{\text{MIN}} &= -2^{\text{iwl}+\text{lbp}-1} + 2^{\text{lbl}-\text{fwl}} - 1. \end{aligned} \tag{11}$$

Thus the fast simulation code construct generated is the following:[9]

```
int tmp;
result=((tmp=expr)>MAX)?MAX:(tmp<MIN)?MIN:tmp;
```

Introducing an additional temporary variable avoids multiple evaluations of `expr`.

Wrap-Around

The SystemC way of casting an expression *expr* to a *wl*-bit tc data type with integer word length *iwl* applying wrap-around as overflow mode is shown here,

```
result = sc_fix(expr,wl,iwl,...,SC_WRAP);
```

For the bit-true ANSI C equivalent of this operation several options exist. An example for a code construct for *wrap around* assuming two's complement arithmetic and a machine word length of mwl is

[8]In many cases, the ANSI C standard [13] does not specify the bit-true behavior of integral data types in case of overflow, quantization, and so forth.

[9]Note that for the code generation we also take the bit-true properties of the processor and compiler into account.

```
result = (expr << SHIFT) >> SHIFT;
```

The amount of shifts computes to SHIFT = mwl − iwl − lpb. The shift left eliminates the MSBs whereas the arithmetic shift right provides a sign extension for the new MSB.

7.3.2 Quantization handling

If the word length of an operand is reduced at the LSB side, we can apply different quantization handling modes. The most frequently encountered are *rounding* and *truncation*.

Rounding

In SystemC the method for casting an expression `expr` to a *wl*-bit two's complement data type with integer word length *iwl* applying rounding as quantization mode is

```
result = sc_fix(expr,wl,iwl,SC_RND,...);
```

Rounding is defined by adding DELTA = LSB/2 to the operand and eliminating the LSBs, for example, by shifting it right SHIFT = lbp − fwl bits. Thus the rounding operation can be realized in the fast simulation code by

```
result = ((expr + DELTA)>>SHIFT)<<SHIFT;
```

Truncation

The truncation operation, given in SystemC by

```
result = sc_fix(expr,wl,iwl,SC_TRN,...);
```

can be implemented efficiently by a bit mask operation,

```
result = expr & (~MASK);
```

Where MASK is given by $2^{\mathrm{lpb}-\mathrm{fwl}-1}$.

For several combinations of cast modes, for example, wrap-around combined with rounding or truncation, more efficient joint quantization and overflow handling C code constructs are generated. The shift operations introduced by the cast code constructs are also utilized to adjust the lbp of the expression, eliminating the need for additional scaling shifts.

8. EXPERIMENTAL RESULTS

The code generated by the FRIDGE fast simulation back end has been benchmarked against the fixed-point simulation classes, which are part of the C++-based *SystemC* language. The simulation classes offer two simulation modes: a mode supporting unlimited fixed-point word lengths based on concatenated data containers and a mode supporting limited precision up to 53 bits based on float-arithmetic and bit manipulations.

The benchmarks have been performed on a SUN Ultra 10 workstation running SOLARIS using the GCC compiler version 2.95.2 with the `-O3` option. The *SystemC* library version 1.0 was utilized for the bit-true simulations. The benchmark is based on typical signal processing kernels, *FIR* 17-tap FIR filter, *DCT* 8×8 JPEG DCT algorithm, *Autocorr* 25 elements 5th order autocorrelation, *IIR* 3rd order IIR filter, *FFT* complex FFT of length 8, *Matrix* 4×4 matrix multiplication.

Four different versions of the kernel functions have been benchmarked:

(i) *Floating-Point.* The execution speed of the floating-point implementation of the algorithms serve as reference for the benchmarks.

(ii) *SystemC.* The quantized bit-true version of the algorithms utilizing the *SystemC* fixed-point data types. The algorithms have been quantized using the *FRIDGE* design environment.

(iii) *SystemC limited precision.* The quantized bit-true code has been compiled with the *limited precision option* to speed up *SystemC* fixed-point operations.

(iv) *Fast simulation code.* The fast fixed-point simulation code based on integral data types has been generated by the FRIDGE back end applying the transformation techniques described in the previous sections. The code yields bit-by-bit the same results as the code utilizing the *SystemC* data types.

The experimental results are presented in Table 1. As the floating-point code has been used as a reference, the experimental data has been scaled relative to the execution speed of the floating-point code. The bit-true *SystemC* code consumes by a factor of 325 to 1103 more run time than the original floating-point code, making bit-true simulation a major bottleneck in the fixed-point design flow. Utilizing the *limited precision* mode of the *SystemC* library, a speedup by a factor of $3.1 \cdots 5.2$ can be achieved, but the fixed-point code is still by a factor of $67 \cdots 234$ slower than the floating-point reference.

The fast simulation code runs by a factor of $18.8 \cdots 90.9$ faster compared to the *SystemC* fixed-point code utilizing the *limited precision* option. For the *unlimited precision* the speedup is $91.0 \cdots 454.2$, respectively.

Compared to the floating-point reference code, the fast simulation code is by a factor of $2.5 \cdots 6.9$ slower. This is due to the host system's architecture and additional shift and bit mask operations necessary to perform lbp-alignment and cast operations to maintain bit-by-bit consistency with the quantized code.

The quantized DCT algorithm contains many cast operations to reduce fixed-point word lengths introduced by the quantization process. As these operations can be modeled efficiently by bit mask operations in the fast simulation code, the highest speedup was achieved for this kernel function.

9. DSP CODE GENERATION

During the recent years, new architectural approaches for DSP processors have been made. The current generation of high performance DSP processors features a pipelined VLIW architecture (very long instruction word), which offers a very high computing performance if a high degree of software pipelining in combination with instruction level parallelism is used. But programming these processors manually utilizing assembly language is a very tedious task. In awareness of this problem, the modern DSP architectures have been de-

Table 1: Relative execution speed.

	Floating-point ANSI C	SystemC	SystemC limited precision	Fast simulation code
FIR	1.0	386.5	102.7	2.8
DCT	1.0	1103.1	233.9	2.5
Autocorr	1.0	694.6	130.6	6.9
IIR	1.0	371.0	120.2	3.1
FFT	1.0	354.7	67.7	2.6
Matrix	1.0	325.9	71.2	3.6

veloped using a processor/compiler codesign methodology which led to compiler-efficient processor designs.

On the other hand, a significant gap in the system design flow is still evident; there is no direct path from a floating-point system level simulation to an optimized fixed-point implementation. Today a manual implementation on the DSP and target specific code optimization is necessary, increasing time-to-market and making design changes very tedious, error prone, and costly. Thus we have developed an optimizing FRIDGE back end to generate target optimized DSP C code. The target specific code generation is necessary for two reasons:

(i) The generic fixed-point data types used for fixed-point simulations are not suited for DSP implementation, as the currently available DSP compilers do not support C++ fixed-point data types. The upcoming generation of DSP compilers will support C++ language constructs, but compiling the fixed-point libraries for the DSP is no viable alternative as the implementation of the generic data types makes extensive use of operator overloading, templates, and dynamic memory management. This will render fixed-point operations rather inefficient compared to integer arithmetic performed on a DSP.

(ii) Compiling the FRIDGE-generated integer ANSI C code on a DSP is also not sufficiently efficient as the generic C code does not exploit the capabilities of the DSP hardware such as built-in saturation and rounding logic or SIMD processing.

As a case study, we have chosen the TMS320C62x processor and its C compiler as a target for the FRIDGE design environment.This enables a seamless design-flow from floating-point to optimized C62x C code utilizing integral data types. Generating a C62x optimized version of a signal processing algorithm using a different set of fixed-point parameters becomes a matter of hours instead of days or weeks using the conventional manual techniques. The C62x integer code generated by the design environment yields bit-by-bit the same results as the fixed-point code utilizing C++ simulation classes on the host machine. Thus a comparative simulation to the “golden reference model” gives the designer a high degree of confidence in the generated code.

The first objective of our case study was to find out which C code constructs compile into efficient C62x assembly code. Thus we applied the DSPstone benchmarking methodology to the C62x optimizing C compiler. The DSPstone project [31], conducted in 1994 by ISS, Aachen University of Technology established a benchmarking methodology for DSP compilers by comparing the performance of compiled C code to hand optimized assembly code in terms of program/data memory consumption and execution time. As a consequence, it allows to identify a possible mismatch between architecture and compiler. The benchmarking has been done using eleven typical signal processing algorithms (FIR, FFT, DCT, minimum error search, etc.). The benchmarking gives quantitative results for cycle count and program memory consumption.

In a second step, we used C62x specific C language extensions (intrinsics) and compiler directives to restructure the off-the-shelf C code while maintaining functional equivalence to the original code. These optimizations led to a considerable improvement in performance in many cases as the compiler was able to utilize software pipelining and instruction level parallelism to speed up the code. It has turned out that software pipelining is the key to achieving a high performance but, on the other hand, requires careful analysis and code restructuring. The evaluation [32] gave quantitative performance data for the C62x compiler and a set of code optimization techniques to generate efficient C62x C code.

In a third step, we benchmarked various implementations of the fixed-point quantization and overflow handling modes on the C62x. This led to a set of optimized implementations for the quantization and overflow handling functionality.

9.1. DSP code transformation

The FRIDGE C62x back end performs similar transformation steps as the fast bit-true simulation code generation presented in Section 6: lbp alignment, cast mode transformation, and data type selection. Additionally, target specific code optimization is performed.

The designer has to keep the special requirements of the DSP target in mind to reach a high level of efficiency. Through our experiments we found that, for example, the number of cast statements and shift operations has a strong influence on the efficiency of the generated code. Thus if the designer chooses settings for the *global annotations* and the *default cast mode* during the early stages of the transformation which do not represent the properties of the target architecture properly, the code optimization and the DSP compiler are not able to generate efficient assembly code.

The optimizations performed in the FRIDGE C62x back end are source level transformations to supply the C62x com-

piler with the best C code possible. The amount of analysis done in an optimizing compiler is usually limited due to constraints of the time used for compilation. In the FRIDGE design environment, control and data flow analysis is performed with the maximum possible accuracy utilizing the techniques presented in Section 5. The information gained during this analysis is available for the back end code transformation as well. Thus we are able to perform code restructuring techniques, which are usually beyond the scope of an optimizing compiler.

9.1.1 The lbp alignment

As the TI C6000 processor family has an *integer* multiplication mode, the *right alignment* strategy of the lbp alignment algorithm can also be applied in the C62x back end. This algorithm implicitly minimizes the number of scaling shifts. In contrast to the fast bit-true simulation, the number of scaling shifts generated is important for the C62x code generation. For the fast simulation code generation we found the potential of shift minimization limited to a performance improvement of 3% $\cdots$ 13% [29]. This is different for the C62x code generation. As the C62x can perform two scaling shift operations per cycle, a shortage of functional units limits the performance in highly software pipelined loops. Thus “shift poisoning” of loops must be avoided, for example, by choosing suitable fixed-point data types for function parameters and central data structures.

9.1.2 Data type selection

As the properties of the data paths of the C62x processor and the width of the integral data types supported by the C62x C compiler are known, the design environment can utilize this information during the transformation process. A set of global annotations for the C62x guides the interpolation process and a set of integral data types with a given bit length is supplied to the C62x back end.

9.1.3 Cast mode transformation

The generic overflow- and quantization handling modes offered by SystemC have to be mapped to the target hardware in an efficient manner. The C62x offers built-in saturation hardware which can be used by the back end. This is illustrated by the following example.

Cast mode: saturation

A cast of an expression to a *wl*-bit two’s complement data type with integer word length *iwl* applying saturation as overflow mode is modeled in SystemC as follows:

```
result=sc_fix(expr,wl,iwl,...,SC_SAT);
```

An implementation of this code construct in generic ANSI C is

```
int tmp;
result=((tmp=expr)>MAX)?MAX:(tmp<MIN)?MIN:tmp;
```

On the C62x the `sshl` intrinsic (saturating shift left) can be used to perform the saturation operation:

```
result=(signed)_sshl(expr,SHIFT)>>SHIFT;
```

where SHIFT is given by mwl – (iwl+lbp). Utilizing the built-in saturation hardware of the C62x via the `sshl` intrinsic allows the generation of code with linear control flow in contrast to the forked control flow in the ANSI C implementation. This significantly speeds up the code.

9.1.4 Loop optimizations

The key to high execution speed on the C62x is software pipelining and instruction level parallelism. This is especially important for loops, where most of the execution time is spent for most digital signal processing algorithms. The latest version of the C62x C compiler is able to perform quite sophisticated loop optimizations to achieve high performance. This can be further improved by restructuring the loops at source level, applying techniques like loop unrolling, scalar expansion and splitting data paths. By introducing *SIMD* (single instruction multiple data) intrinsics it is possible to reduce the required number of load/store operations significantly. The C62x back end utilizes the data- and control flow information and the code transformation infrastructure to identify possible loop optimizations and to perform the necessary loop restructuring. The design environment maintains the consistency of generated code.

10. EXPERIMENTAL RESULTS

We have benchmarked the cycle count performance of the generated C62x integer C code using two sets of typical signal processing kernel functions: The first set consists of six off-the-shelf kernels which have been initially coded without DSP specific code optimization. The second set of kernels has been extracted from TI’s C6000 compiler benchmarking suite.

10.1. Off-the-shelf kernels

This set of kernels consists of six signal processing functions, which also have been used for the benchmarks in Section 8: *FIR, DCT, Autocorr, IIR, Matrix, Dotprod*. The code has been translated using TI’s C6x compiler version 4.0 [33] and the performance has been compared with three reference codes:

(i) *C67x floating-point C code.* The C67x floating-point DSP is code-compatible to the C62x and its C compiler is mostly identical to the C62x C compiler, thus the performance of the generated fixed-point C code can be compared to the original floating-point C code.

(ii) *C62x floating-point emulation.* The floating-point emulation library which is part of the C62x compiler’s run time library allows the user to perform floating-point arithmetic on the C62x processor. The floating-point operations are executed as function calls.

(iii) *C62x integer ANSI C code.* The FRIDGE back end allows the designer to generate ANSI C fixed-point code without C62x specific optimization. This code can also be compiled and executed on the C62x processor. The efficiency of the target specific code optimization can be benchmarked using this code.

Table 2: Cycle count.

	Floating-point	Float emulation	Generic ANSI C	Target specific C
Device	C67x	C62x	C62x	C62x
FIR	132	1304	523	234
DCT	331	34163	1509	622
Autocorr	564	6581	3057	1041
IIR	73	708	82	81
Matrix	108	4999	1600	233
Dotprod	95	9436	1300	406

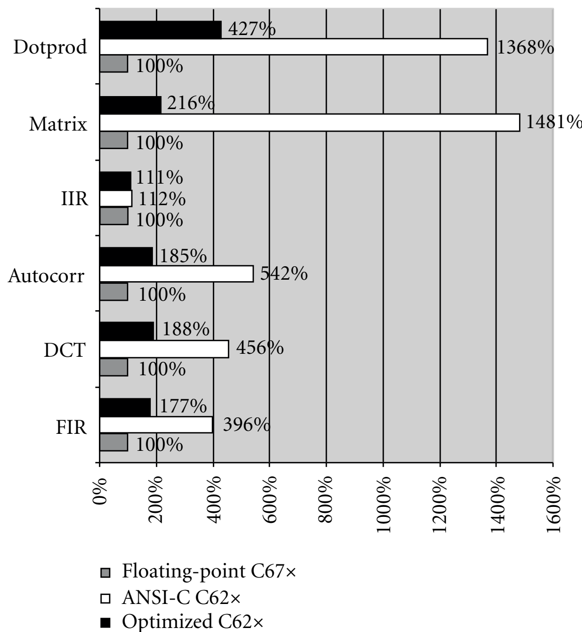


Figure 10: Cycle count relative to floating-point code.

Table 2 presents the benchmarking results for the six kernel functions. Figure 10 illustrates the relative cycle count. As the C67x floating-point code has been used as a reference, it was scaled to 100%. For readability the results of the floating-point emulation have been omitted in the bar graph.

As depicted in Table 2 the C62x floating-point software emulation has a cycle count which is by a factor of 9.7 to 103 higher than the cycle count of the same code compiled for the floating-point processor.

The generic ANSI C integer code without C62x specific language extensions is by a factor of 1.1 to 14.8 slower than the floating-point code. The integer code performs additional shift- and bit-masking operations to ensure the bit-true behavior. Some of the cast-operations cannot easily be modeled in generic ANSI C. Thus a significant overhead is introduced for kernel functions where many cast operations are inserted by the interpolation (e.g., the DCT).

The performance can be improved by matching the generated code to the target architecture. For example, utilizing the sshl intrinsic is a convenient way to access the C62x saturation hardware directly. This reduces the overhead introduced by the additional shift and cast operations to a factor of 1.1 to 4.3 compared to the floating-point code.

For the floating-point code of the *Dotprod* kernel function, the compiler was able to generate efficient code using 95 cycles for 64 vector elements. For the fixed-point code, the additional operations needed for cast operations in the inner loop prevent the compiler from achieving similar efficiency. Removing all scaling shifts and overflow protection from the inner loop of the fixed-point code for this kernel yields a cycle count of 83. Introducing a single scaling shift in the inner loop brings the cycle count up to 147, adding overflow protection yields 406 cycles. Similar effects appear in the *Matrix* kernel benchmark.

10.2. TI compiler benchmarking kernels

This set of kernels consists of six signal processing functions: *IIR* 16-coefficient IIR filter, *IIR cas biquads* 10 cascaded biquads, *FIR* 10-tap 40 sample FIR filter, *MAC VSELP* two 40 samples vectors, *VQ MSE* MSE between two 256 element vectors, *VEC SUM* vector sum of two 44 sample vectors.

For these kernels hand-optimized C62x assembly code and C62x integer C code is available on TI's website. It is noteworthy that neither the C code nor the assembly code was coded with overflow protection. For the embedding of input and output operands, implicit assumptions were made which reduced the number of scaling shifts in the kernel functions. Thus the hand-optimized C62x assembly code can serve as an "upper bound" for the efficiency of the FRIDGE C62x design flow.

We derived the floating-point code from the integer C code. The function interfaces in the floating-point code were manually annotated with fixed-point specifications to get hybrid code. The hybrid code was used as input to generate optimized C62x integer code from the FRIDGE C62x environment. The FRIDGE generated C62x code features full overflow protection and maintains consistency for the "location of binary point" for input and output operands. The code has been translated using TI's C6x compiler version 4.0 [33] and the performance has been compared to the reference codes:

(i) *C67x floating-point C code.* This is the floating-point code compiled for the C67x processor.

(ii) *C62x hand-optimized integer C code.* This is the original hand-optimized code from the benchmarking suite.

Table 3: Cycle count.

	Floating-point	Assembly	Hand optimized ANSI C	FRIDGE
Device	C67x	C62x	C62x	C62x
IIR	85	42	38	72
IIR BIQUAD	149	70	82	108
FIR	315	237	278	373
MAC VSELP	175	61	59	207
VQ MSE	559	279	275	275
VEC SUM	63	48	51	127

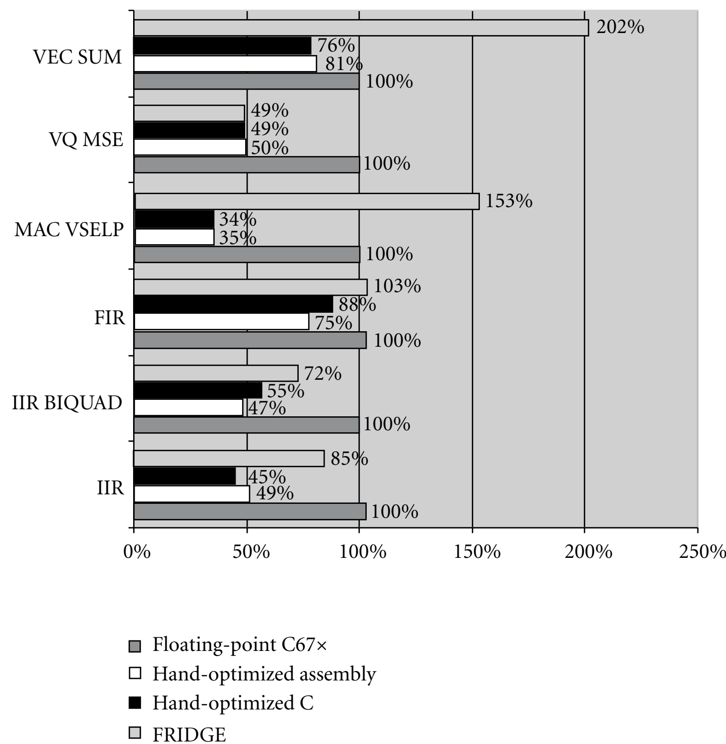


Figure 11: Cycle count relative to floating point code.

(iii) *C62x hand-optimized assembly code.* The hand-optimized assembly code served as a reference for the benchmarks.

Table 3 presents the benchmarking results for the six kernel functions. Figure 11 illustrates the relative cycle count. For consistency, the floating-point code has been used as a reference, it was scaled to 100%.

For these kernels, the C6x compiler was obviously able to generate very efficient code. For consistency we have measured the cycle count including the function call. This causes the hand-optimized C code to be faster than the hand-optimized assembly code for some kernels. The floating-point code is slower than the hand-optimized assembly and C code in all cases as the floating-point instructions need more execution stages than their integer counterparts. For this set of kernel functions the FRIDGE generated code consumes more cycles than the hand-optimized code as additional shift and cast operations for overflow protection are performed. For some kernels, such as, the *MAC VSELP* and the VEC SUM, this leads to a significant overhead as the hand-optimized code uses the processor's functional units in a very efficient manner. Introducing additional shift and bit mask operations in the innermost loop slows down the code, as no unused functional units are available in the very tight loop pipelining schedule. Especially the s-unit which performs shift operations is heavily used and becomes the performance bottleneck. Nevertheless, the FRIDGE generated code comes very close in performance to the hand-optimized code while offering full overflow protection and maintaining consistency of input and output data formats.

11. SUMMARY

The FRIDGE design environment presented in this article allows the designer to concentrate on the critical issues of floating-point to fixed-point design flow. Thus he is able to explore the design space more efficiently. The interpolative transformation which is based on analytical range propagation enables an accelerated development cycle and in consequence a shorter time-to-market.

The fast simulation code generation as well as the DSP back end benefits directly from the advanced control and data flow analysis techniques we developed. The concept of *abstract execution*, in combination with a state-driven memory model and coupled iterators, yields results with the precision necessary for the back end transformation steps.

The verification of the fixed-point algorithm has to be performed by means of simulation. Existing C++-based fixed-point libraries increase simulation-time by up to two orders of magnitude compared to the corresponding floating-point simulation. The FRIDGE fast simulation back end applies advanced compile-time analysis concepts, analyzes necessary casting operations, and selects the appropriate built-in data type on the host machine, thus a speedup by a factor of 20 to 400 compared to the *SystemC* code while maintaining bit-by-bit equivalence was achieved.

The target specific C code generation provides a direct link from a floating-point code to C62x C code using integral data types. The generated code yields bit-by-bit the same results as the bit-true SystemC code for host simulation, enabling comparative simulation to the reference model. As proven by the experimental data, the generated C62x C code comes very close to hand-optimized C- and assembly code.

These features make FRIDGE a powerful design environment for the specification, evaluation, and implementation of fixed-point algorithms.

REFERENCES

[1] Synopsys Inc., "CoCentric System Studio—User's Manual," Mountain View, Calif, USA.

[2] Mathworks Inc., "Simulink Reference Manual," March 1996.

[3] Cadence Design Systems, 919 E. Hillsdale Blvd., "SPW User's Manual," Foster City, Calif, USA.

[4] T. Grötker, E. Multhaup, and O. Mauss, "Evaluation of HW/SW tradeoffs using behavioral synthesis," in *Proc. Int. Conf. on Signal Processing Application and Technology*, Boston, Mass, USA, October 1996.

[5] B. Liu, "Effect of finite word length on the accuracy of digital filters—a review," *IEEE Trans. on Circuit Theory*, vol. 18, no. 6, pp. 670–677, 1971.

[6] H. Keding, M. Willems, M. Coors, and H. Meyr, "FRIDGE: A fixed-point design and simulation environment," in *Proc. European Conference on Design, Automation and Test*, pp. 429–435, Paris, France, February 1998.

[7] M. Willems, V. Bürsgens, and H. Meyr, "FRIDGE: Floating-point programming of fixed-point digital signal processors," in *Proc. Int. Conf. on Signal Processing Application and Technology*, pp. 1000–1005, San Diego, Calif, USA, September 1997.

[8] M. Willems, V. Bürsgens, H. Keding, T. Grötker, and H. Meyr, "System level fixed-point design based on an interpolative approach," in *Proc. Design Automation Conference*, pp. 293–298, Anaheim, Calif, USA, June 1997.

[9] S. Kim, K. Kum, and W. Sung, "Fixed-point optimization utility for C and C++ based digital signal processing programs," in *Workshop on VLSI and Signal Processing '95*, pp. 197–206, Osaka, Japan, November 1995.

[10] Frontier Design Inc., "A|RT Library User's and Reference Documentation," Danville, Calif, USA, 1998.

[11] Synopsys Inc., CoWare Inc., Frontier Design Inc., "SystemC User's Guide, Version 2.0," 2001.

[12] W. Sung and K. Kum, "Word-length determination and scaling software for a signal flow block diagram," in *Proc. IEEE Int. Conf. Acoustics, Speech, Signal Processing*, pp. 457–460, Adelaide, Australia, April 1994.

[13] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, USA, 2nd edition, 1988.

[14] M. Willems, *A methodology for the efficient design of fixed-point systems*, Ph.D. thesis, Aachen University of Technology, 1998.

[15] Mentor Graphics, "DSP Station User's Manual," San Jose, Calif, USA.

[16] C. Hankin, "Program analysis tools," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 1, pp. 6–12, 1998.

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

[18] M. J. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, Redwood City, Calif, USA, 1996.

[19] C. Hankin, F. Nielson, and H. R. Nielson, *Principles of Program Analysis*, Springer, Heidelberg, Germany, 1999.

[20] F. Martin, "PAG—an efficient program analyzer generator," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 1, pp. 46–67, 1998.

[21] MIPS Computer Systems, "UMIPS-V Reference Manual (Pixie and Pixstats)," Sunnyvale, Calif, USA, 1990.

[22] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 4, pp. 1319–1360, 1994.

[23] S. B. Akers, "Binary decision diagrams," *IEEE Trans. on Computers*, vol. 27, no. 6, pp. 509–516, 1978.

[24] European Telecommunication Standard Institute, "GSM full rate speech transcoding," GSM recommendation 06.10, February 1992.

[25] S. Kim, K. Kum, and W. Sung, "Fixed-point optimization utility for C and C++ based digital signal processing programs," *IEEE Trans. on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 45, no. 11, pp. 1455–1464, 1998.

[26] L. De Coster, *Bit-true simulation of digital signal processing applications*, Ph.D. thesis, KU Leuven, 1999.

[27] Mentor Graphics, "DSP Architect, DFL User's and Reference Manual," 1994.

[28] K. Kum, J. Kang, and W. Sung, "A floating-point to integer C converter with shift reduction for fixed-point digital signal processors," in *Proc. IEEE Int. Conf. Acoustics, Speech, Signal Processing*, vol. 4, pp. 2163–2166, Phoenix, Ariz, USA, March 1999.

[29] H. Keding, M. Coors, O. Lüthje, and H. Meyr, "Fast bit-true simulation," in *Proc. the Design Automation Conference*, pp. 708–713, Las Vegas, Nev, USA, June 2001.

[30] S. K. Mitra, *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, New York, NY, USA, 1998.

[31] V. Živojnović, J. Martínez, C. Schläger, and H. Meyr, "DSPstone: A DSP-oriented benchmarking methodology," in *Proc. International Conference on Signal Processing Applications and Technology*, Dallas, Tex, USA, October 1994.

[32] M. Coors, O. Wahlen, H. Keding, O. Lüthje, and H. Meyr, "C62x compiler benchmarking and performance coding techniques," in *Proc. International Conference on Signal Processing Applications and Technology*, Orlando, Fla, USA, November 1999.

[33] Texas Instruments, USA, "TMS320C6000 Optimizing Compiler User's Guide," March 2000.

Martin Coors received the diploma in electrical engineering from Aachen University of Technology (RWTH), Aachen, Germany. In 1997, he joined the Institute for Integrated Signal Processing Systems (ISS) at RWTH Aachen as a research assistant. His research interests include DSP code optimization techniques, fixed-point design methodologies and code generation for embedded processors.

Olaf Lüthje received the diploma in electrical engineering from Aachen University of Technology (RWTH), Aachen, Germany, and is currently working towards the Ph.D. degree in electrical engineering at the same institute. His research interests focus on fixed-point design methodology and data flow analysis.

Holger Keding received the diploma in electrical engineering from Aachen University of Technology (RWTH), Aachen, Germany. From 1996 to 2001 he was with ISS to work towards his Ph.D. thesis. Having finished his Ph.D., he joined the system level design group of Synopsys as a senior corporate application engineer. His research interests include fast bit-true simulation and fixed-point and system-level design methodology.

Heinrich Meyr received his M.S. and Ph.D. from ETH Zurich, Switzerland. He spent over 12 years in various research and management positions in industry before accepting a professorship in electrical engineering at Aachen University of Technology (RWTH Aachen) in 1977. He has worked extensively in the areas of communication theory, synchronization, and digital signal processing for the last thirty years. His research has been applied to the design of many industrial products. At RWTH Aachen he heads an institute involved in the analysis and design of complex signal processing systems for communication applications. He was a cofounder of CADIS GmbH (acquired 1993 by Synopsys, Mountain View, California), a company which commercialized the tool suite COSSAP extensively worldwide used in industry. He is a member of the Board of Directors of two companies in the communications industry. Dr. Meyr has published numerous IEEE papers. He is author together with Dr. G. Ascheid of the book "Synchronization in Digital Communications," Wiley 1990, and of the book "Digital Communication Receivers. He is also the author of "Synchronization, Channel Estimation, and Signal Processing" (together with Dr. M. Moeneclaey and Dr. S. Fechtel), Wiley, October 1997. He holds many patents. He served as a Vice President for International Affairs of the IEEE Communications Society and is a Fellow of the IEEE.