FRIDGE: AN INTERACTIVE CODE GENERATION ENVIRONMENT FOR HW/SW CODESIGN

Markus Willems, Volker Bürsgens, Thorsten Grötker and Heinrich Meyr

Institute for Integrated Systems in Signal Processing
Aachen University of Technology
Templergraben 55, 52056-Aachen, Germany
{willems,buersgen,groetker,meyr}@ert.rwth-aachen.de

ABSTRACT

Digital mobile systems are sensitive to power consumption, chip size and costs. Therefore they are realized using fixed-point architectures, either dedicated HW or fixed-point processors. On the other hand, system design starts from a floating-point description. These requirements have been the motivation for FRIDGE [1], a design environment for the specification, evaluation and implementation of fixed-point systems. FRIDGE offers a seamless design flow from a floating-point description to a fixed-point implementation. Within this paper we focus on the FRIDGE-concept of an interactive, automated transformation of floating-point programs written in ANSI-C into fixed-point specifications, based on an interpolative approach. Since HW and SW implementations of the same functionality in general require different fixed-point specifications, the design time reductions that can be achieved by using FRIDGE make it a key component for an efficient HW/SW-CoDesign.

I INTRODUCTION

Digital system design especially for mobile applications is characterized by ever increasing system complexity that has to be implemented within reduced time, resulting in minimum costs and short time-to-market. These characteristics call for a seamless design flow that allows to perform the suitable design steps on the highest level of abstraction.

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

Fixed-point system design requires a specific design flow that enables maximum abstraction for all design steps, as illustrated by fig.1

Algorithm design starts from a floating-point description. This abstraction from all fixed-point effects such as quantization noise and overflow problems allows an evaluation of the algorithm space, reducing the algorithmic alternatives before analyzing the quantization effects on the algorithmic behavior. In addition, it enables a maximum design reuse since the most general description of a functionality can be used (there is only one adder, not several adders handling different input and output wordlengths).

The transformation to the fixed-point level requires to assign a fixed wordlength and a fixed exponent to every operand, while the control structure and the operations of the floating-point program remain unchanged (the input and output parameters of the adder have to be determined). The fixed-point model is used to analyze the bit-true algorithmic system performance.

[1] Fixed-point pRogrammIng DesiGn Environment

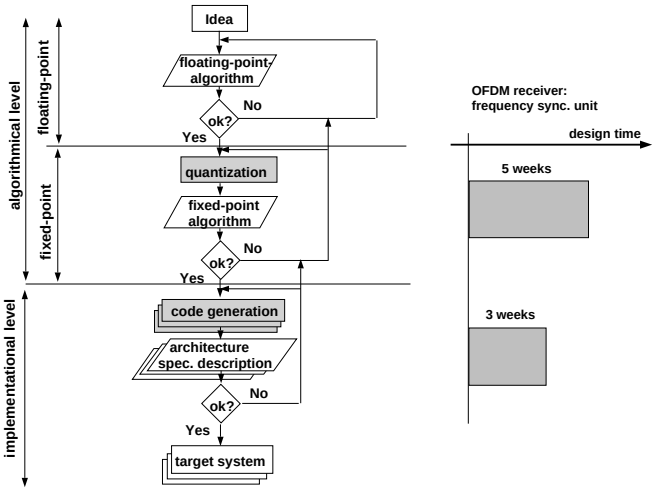

Figure 1. Fixed-point design process

To move to the implementational level, the fixed-point algorithm has to be transferred into the best suited target description, either using a HDL or a programming language (now it has to be decided how to implement the adder, while the bit-true behavior is completely specified).

Since both transformations reduce abstraction, they are not unique but address a complex design space. Within this paper we focus on the first transformation, converting a floating-point description into a fixed-point description. Although performing a transformation on the algorithmical level, even here one can no longer abstract from the target system. This is due to the fact that HW and SW put different constraints on the fixed-point specification: for SW, the wordlength is already fixed by the processor and the minimization of shift operations is a concern, while for HW the wordlength is free and its minimization is an issue. A typical HW/SW-CoDesign therefore requires multiple transformations, depending on different partitioning constellations. Existing approaches (see sec.2) require a manual float-to-fixed transformation, for more complex designs accounting for 50% or more of the design time once the floating-point algorithm is fixed [1]. Therefore, an automated transformation from the floating-point to the fixed-point level is an enabling feature for HW/SW-CoDesign.

This constellation motivated FRIDGE, an interactive design environment for the specification, emulation and implementation of fixed-point systems. The basic concepts of FRIDGE are presented in the following. After a brief overview of related work (sec.2), in sec.3 we describe the interpolative approach, which is a core functionality of FRIDGE. The realization of the interpolative concept and the resulting capabilities of FRIDGE are subject of sec.4.

II RELATED WORK

Several design environments allow to specify a fixed-point algorithm, starting from a floating-point description of the system. Two concepts exist:

1. block diagram based algorithm specifications where blocks represent the functionality and signals the data flow among these blocks. The designer assigns information about wordlength and the location of the binary point to the signals, but the blocks' functionalities are fixed [2,3,4]. The blocks' black box behavior is a severe limitation of these approaches. As a consequence, functionalities are limited to blocks with the fixed-point behavior completely specified by the interface, such as adders, multipliers, etc. For more complex functionalities, e.g. filters, different fixed-point behaviors require to manually exchange the block.

   A design bottleneck even as important as the black box behavior is that a complete fixed-point specification requires to specify **all** signals. For most concepts, this calls for a manual assignment. Recently, Sung [5] presented a concept to reduce the effort of a manual annotation based on exhaustive simulations to determine the fixed-point format of all non-specified signals. Long system response times restrict this concept to only a very few non-specified signals. Since typical designs often include 1000 signals or more [6], manual assignment is still necessary for most of the signals.

2. textual descriptions where the system's functionality is described using a programming language. Examples include DFL [7] as well as C^{++}-based concepts [8]. Both concepts allow a fixed-point instantiation of variables at **declaration time**, i.e. a variable keeps a unique fixed-point representation throughout the complete program. As for the block diagram concepts, for a fixed-point algorithm specification all variables have to be annotated manually.

III THE INTERPOLATIVE APPROACH

In the following we restrict ourself to a textual representation of an algorithm. This does not come as a limitation since most block diagram based design environments include code generators that transfer the block diagram specification into (mostly C) code. Note that all concepts can be applied to block diagram descriptions as well.

A fixed-point specification of an algorithm requires to assign a three tuple $<wl,iwl,s>$ to every operand, with wl the wordlength, iwl the number of integer bits and s the sign (which might be *unsigned* or *two's complement*). One can think of alternative representations which can be transferred into the presented format. See fig.2 for the representation.

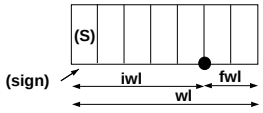

Figure 2. Fixed-point data type specification

As pointed out above, the manual annotation of all operands as required by the existing concepts is hardly acceptable even for a single transformation. It is even more of a design bottleneck for HW/SW-CoDesign where iterative transformations become necessary.

As a consequence, we propose an alternative design flow, denoted the **interpolative approach** which is illustrated by fig.3.

1. Local annotations:
   The design starts from the floating-point description.

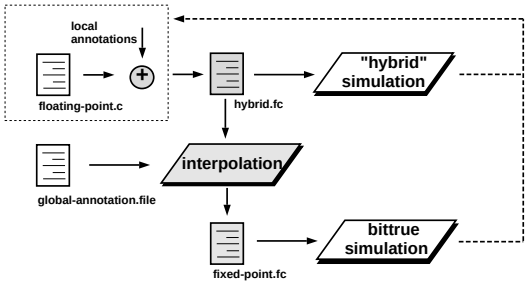

Figure 3. Design flow based on interpolation

   In addition, the designer assigns fixed-point information to **some** fixed-point operands that are critical to his design or already known with their fixed-point specification (e.g. the interface format of a system). This results in a hybrid specification, where some operands are already fixed-point, while others remain floating-point.

2. Simulation:
   Next, the hybrid specification is simulated. This allows the designer to check whether the locally annotated specification still meets the design criteria. If not, a modification of the local annotations or even the floating-point algorithm becomes necessary.

3. Interpolation:
   Once the annotated program matches the design criteria, the remaining floating-point operands are transferred to fixed-point operands by interpolation. 'Interpolation' expresses the determination of the fixed-point parameters of the non-specified operands from the information that is inherent to the annotated operands.

   The interpolation concept is based on three key ideas:

   (a) Worst case estimation:
   The principle can best be illustrated by an example: $a = b + c$
   For a, sign and sufficient integer wordlength depend on the range that a can take. A worst case range estimation is possible, given the range information for b and c: $min\{a\} = min\{b\} + min\{c\}$, $max\{a\} = max\{b\} + max\{c\}$. For the fractional wordlength, no information is lost if $fwl(a) = max\{fwl(b), fwl(c)\}$.

   (b) Global annotations:
   While local annotations express fixed-point information for single operands, global annotations describe restrictions that have to be matched throughout the complete design.

   For different targets, different global restrictions apply. For SW, the functional units to perform specific operations are already defined. Consider a 16x16 bit multiplier, writing to a 32 bit register. A global annotation informs the interpolator that the wordlength of a multiplication operand is not allowed to exceed 16 bit. The result's wordlength must not exceed 32 bits, what is guaranteed by the operation itself.

   For an ASIC implementation, no fixed wordlength constraints exist for specific arithmetic units. So global annotations might inform about a maximum wordlength max that shall not be exceeded. Once the interpolation would result in a wordlength exceeding max, it is reduced to *default*.

   Global annotations are the enabling feature for an efficient HW/SW-CoDesign. As already pointed out above, although starting from the same floating-point algorithm, in general different fixed-point specifications are necessary. If it is not known which

parts of the design to realize in HW and which parts in SW, global annotations allow to generate the different fixed-point specifications by exchanging a single file.

(c) Designer support:
If an interpolation is not possible for the complete design since the annotated information is not sufficient, the interpolator can inform about the location where it is impossible to continue and can ask for additional information.

The interpolation supplies a fully annotated program, where a unique fixed-point data type is assigned to each operand. Therefore, the effects of local and global annotations become completely visible to the designer.

4. Simulation:
Since the global annotations might have changed the algorithmic performance of the specification, the (now completely defined) fixed-point program has to be simulated again. If it is found that the system does not fulfill the design criteria, the initial description might be modified by adding annotations.

The interpolative design flow comes with several advantages compared to existing approaches:

- design time reduction: the designer can concentrate on the specifications which are important to his design while the effects to the remaining parts are evaluated in an optimum way by the interpolator.
- designer's control: the designer fully controls the transformation process since he can assign all information (either locally or globally) that is crucial for his design. The interpolation makes visible the effects on those parts of the design that have not been specified explicitly by local annotations. This simplifies iterative modifications by the designer when he wants to assign additional annotations.
- Design space evaluation: the evaluation of different fixed-point specifications becomes very easy since only some annotations have to be exchanged while the remaining specifications are automatically derived from this information. This is extremely useful especially for HW/SW-CoDesign, where different targets must be addressed within short time.

IV THE FRIDGE FRAMEWORK

FRIDGE is a complex and most advanced design environment for the specification and code generation for fixed-point architectures. See the illustration of the FRIDGE framework in fig.4

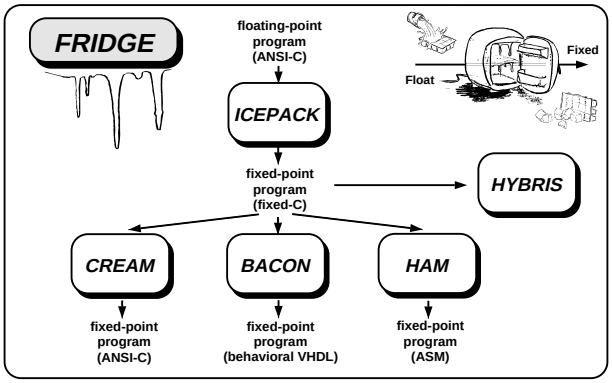

Figure 4. The FRIDGE framework

Within this paper we focus on the transformation from the floating-point algorithm to the fixed-point algorithm, which is based on the interpolative as described in sec.3. The FRIDGE-specific features of this concept shall be described in more detail.

4.1. fixed-C

The input specification of FRIDGE is a floating-point program written in ANSI-C. Most block diagram based design environments come with a C-code generator which allows to transfer these specifications into ANSI-C as well [2,4,7,9]. ANSI-C offers no efficient support for fixed-point data types (for a detailed discussion see [5]). Since the interpolative approach calls for the possibility of a hybrid specification of the algorithm, the ANSI-C standard has been extended to *fixed-C* by introducing two parameterizable fixed-point data types, named *fixed* and *Fixed*.

1. *fixed* $a, *b, c[8]$
A variable is declared to be of data type *fixed*, but no instantiation is performed at declaration time. *fixed* allows to handle pointers and arrays as known from ANSI-C.
Example: *a =fixed(wl,iwl,sign,cast,*b)*;
a receives data type *<wl,iwl,sign>*, the value of **b* is casted according to *cast*. Prior instantiations of *a* are overwritten. A data type instantiation of a *fixed* variable is performed only by an assignment. **Assignment-time** instantiation is motivated by the specific design flow: transformation starts from a floating-point program, where the designer does not care about fixed-point requirements when he specifies an assignment, actually he uses the floating-point description to abstract from these problems. If the language would require a declaration-time instantiation (as for the existing concepts [7,8]), two possibilities exist:
   - the initial instantiation covers all possible assignments, so no context specific fixed-point specification would be possible, or
   - variables are renamed, with a context specific instantiation at declaration time. This comes with severe problems for conditional and loop structures. Even as important, to simplify code analysis and maintainance it is highly desirable to keep the floating-point variable names for the fixed-point program.

   As a consequence, assignment-time instantiation is the method of choice for an efficient design flow.
2. *Fixed<wl,iwl,sign>* $d, *e, g[8]$ (*Forced-fixed*)
Different to data type *fixed*, FRIDGE performs a data type check for every assignment to a *Fixed* variable.
Example: $d = fixed(7, 4, s, sr, *e)$;
If $(wl \neq 7) \vee (iwl \neq 4) \vee (sign \neq s)$, (so the right hand side does not match the required format) the output depends on the selected mode:
   - warning mode: FRIDGE informs the designer about the mismatch but continues with assigning the right hand side to variable d (so d is of data type $< 7, 4, s >$). Except for the warning there is no difference whether d has been declared *Fixed* or *fixed*.
   - forced mode: FRIDGE interrupts execution and calls for annotations that guarantee the correct assignment to d.

   Fixed is the data type of choice for variables that serve as an interface to other functionalities. Therefore *Fixed* is the enabling feature for concurrent engineering.

Annotations are not limited to fixed-point parameters such as wl, iwl and $sign$, but it is possible to assign information about ranges, mean and variance of a specific operand. The fixed-point parameters can be derived from these informations.

4.2. HYBRIS: HYBRId Simulator

HYBRIS allows a bit-true simulation of *fixed-C* (and therefore hybrid) programs. Special features of HYBRIS allow to collect additional information for user-specified operands:

- identify the range
- identify mean and variance
- display value distribution using a histogram, supplying valuable information for local annotations.
- detect overflows for fixed-point operands: either interrupt simulation or generate a statistic. This is extremely helpful for analyzing the effects of local annotations.

HYBRIS can easily be integrated into all C-based simulation environments. This allows to enhance existing concepts using the advanced capabilities of FRIDGE: stimuli generation, post processing and performance analysis can be done by the old environment, while HYBRIS handles the processing part of the design.

4.3. ICEPACK: Interpolative CodE Processing PACKage

ICEPACK, the interpolator integrated in FRIDGE, is based on the interpolation principles as described in sec.3. A more detailed description can be found in [10]. Some capabilities of ICEPACK shall be highlighted by the following examples.

- sequential code:

```
float b = 2.75;        fixed b = fixed(4,2,u,sr,2.75);
float c = 5.0;         fixed c = fixed(3,3,u,sr,5.0);
float a;               fixed a;

a = c - b;             a = fixed(4,2,u,sr,c-b);
```

  For constants, ICEPACK determines the minimum requirements on *iwl, wl* and *sign* that are necessary to represent the data without loosing any information. By interpolation, ICEPACK determines range (and therefore *iwl* and *sign*) and sufficient fractional wordlength *fwl* of a. (For the example above, ICEPACK would have analyzed that $c - b$ is a constant and can be replaced).

- conditional structures:

```
float b = 2.75;        fixed b = fixed(4,2,u,sr,2.75);
float c = 5.0;         fixed c = fixed(3,3,u,sr,5.0);
float a;               fixed a;

if (condition)         if (condition)
   a = c - b;            a = fixed(4,2,u,sr,c-b);
else                   else
   a = 1.875;            a = fixed(4,1,u,sr,1.875);

d = a;                 d = fixed(5,2,u,sr,a);
```

  Depending on which branch is executed, a is instantiated differently. Before using a as an operand following these branches, ICEPACK merges the information inherent to all possible assignments so that no information can get lost.

- loop constructs:

```
float b[2] =           fixed b[2] =
      {2.75,-3.5};           {fixed(4,2,u,sr,2.75),
                              fixed(4,3,s,sr,-3.5)};
float a;               fixed a;

                       int wl_a[] = {5,3};
                       int *pwl_a = wl_a;
                       int iwl_a[] = {3,1};
                       int *piwl_a = iwl_a;

a = 0;                 a = 0;
for (i=0;i<2;i++)      for (i=0;i<2;i++)
  a = a + b[i];          a = fixed(*pwl_a++,*piwl_a++,
                                   s,sr,a + b[i]);
```

  ICEPACK analyzes the number of iterations. This is not limited to *for*-loops but covers *while* and *do-while* loops as well. For each iteration it determines the necessary fixed-point parameters separately. If ICEPACK identifies that the parameters are not equal for all iterations, it automatically generates an array that is accessed via a pointer. Notice the advantage of iteration specific instantiations that would be impossible with the declaration-time instantiation principle.
  ICEPACK comes with specific capabilities for loops of data dependent length by combining 'loop' and 'conditional' approaches (see [10]).

- Pointers and static variables:
  ICEPACK includes a comprehensive analysis for pointers and static variables. For a more detailed description see [10].

V SUMMARY

Digital system design especially for mobile applications requires to transfer floating-point programs into fixed-point programs. State-of-the-art approaches make it necessary to completely annotate fixed-point specifications to all operands manually, what has been identified to be error prone and time consuming. Complete manual annotations are hardly acceptable for traditional designs but become an unacceptable situation for HW/SW-CoDesign where the necessary evaluation of the design space requires multiple float-to-fixed transformations. This has motivated the construction of FRIDGE, an interactive fixed-point code generation environment. Within this paper we concentrated on the interpolative approach which is a key feature of FRIDGE. The interpolative approach allows to generate a fixed-point specification starting from local annotations for specific operands of the floating-point program. This became possible by introducing *fixed-C*, ANSI-C extended by two fixed-point data types. Efficient HW/SW-CoDesign is enabled by different algorithm transformation strategies that can be specified by target specific global annotations. These features in combination with target specific implementation strategies make FRIDGE a complex design environment for the specification, evaluation and implementation of fixed-point algorithms. This environment can be easily integrated into existing C-based design environments.

REFERENCES

[1] T. Grötker, E. Multhaup, and O. Mauss, "Evaluation of HW/SW Tradeoffs Using Behavioral Synthesis," in *Proc. of ICSPAT '96*, (Boston), Oct. 1996.

[2] Cadence Design Systems, 919 E. Hillsdale Blvd., Foster City, CA 94404, USA, *SPW User's Manual*.

[3] Angeles Systems, *VANDA-Design Environment for DSP Systems*, 1994.

[4] Mathworks Inc., *Simulink Reference Manual*, Mar. 1996.

[5] W. Sung and K. Kum, "Word-Length Determination and Scaling Software for a Signal Flow Block Diagram," in *Proceedings of ICASSP '94*, pp. II 457– 460, Apr. 1994.

[6] P. Zepter, T. Grötker, and H. Meyr, "Digital Receiver Design using VHDL Generation from Data Flow Graphs," in *Proc. 32nd Design Automation Conf.*, June 1995.

[7] Mentor Graphics, 1001 Ridder Park Drive, San Jose, CA 95131, USA, *DSP Station User's Manual*.

[8] 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*, (Osaka), pp. 197–206, Nov. 1995.

[9] Synopsys, Inc., 700 E. Middlefield Rd., Mountain View, CA 94043, USA, *COSSAP User's Manual*.

[10] M. Willems, V. Bürsgens, and H. Meyr, "FRIDGE: Tool Supported Fixed-Point Code Generation from Floating-Point Programs Based on An Interpolative Approach," in *DAC*, 1997. submitted for publication.