SYSTEM LEVEL FIXED-POINT DESIGN BASED ON AN INTERPOLATIVE APPROACH

Markus Willems, Volker Bürsgens, Holger Keding, Thorsten Grötker, Heinrich Meyr

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

ABSTRACT

The design process for fixed-point implementations either in software or in hardware requires a bit-true specification of the algorithm in order to analyze quantization effects on an algorithmical level, abstracting from implementational details. On the other hand, system design starts from a floating-point description, so that a transformation of a floating-point description into a fixed-point description becomes necessary. Within this paper we present a tool that allows an automated, interactive transformation from floating-point ANSI-C into a bit-true specification based on a new data type *fixed* that is introduced as an extension to ANSI-C. The concept is rooted in a sophisticated data dependency analysis that allows to handle control structures as well as pointers. It is part of the fixed-point design environment FRIDGE [1] which includes an advanced simulator that covers the extended ANSI-C syntax as well as target specific compilers which allow to generate efficient fixed-point implementations either for HW or for SW, starting from the bit-true algorithm specification.

I INTRODUCTION

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

Fixed-point implementations are preferred to floating-point implementations whenever the system is sensitive to power consumption, chip size and device price. Fixed-point system design requires a specific design flow, as illustrated by fig.1.

Algorithm design starts from a floating-point description. This allows to ignore the effects of finite wordlengths and fixed exponents and to abstract from all implementation details. The algorithm space can be evaluated in the most efficient way, only concentrating on whether the algorithm fulfills the performance requirements of the system, such as bit error rate or speech quality. Performance analysis in general is based on simulation.

34th Design Automation Conference

DAC 97, Anaheim, California
(c) 1997 ACM 0-89791-920-3/97/06 ..$3.50

[1] Fixed-point pRogrammIng DesiGn Environment

0-89791-847-9/97/0006/$3.50 DAC 97 - 06/97 Anaheim, CA, USA

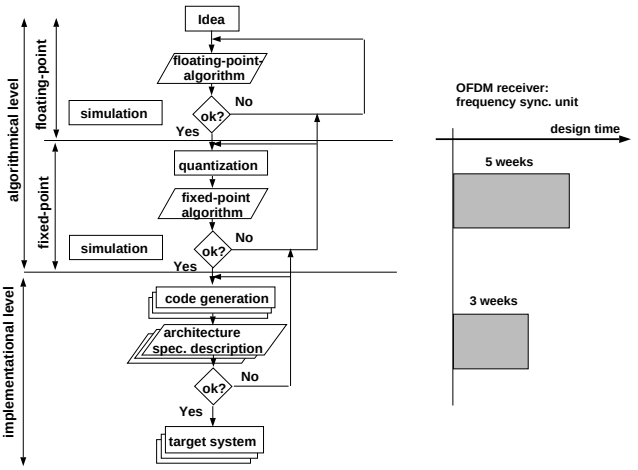

Figure 1. Fixed-point design process

The floating-point description can be done using a block diagram description [1,2,3,4,5,6] (which up to now is very much limited to dataflow oriented applications), where the functional blocks can be user defined (using a high level language) or come from a library. As well, a high level language such as ANSI-C can be used for the floating-point description of the algorithm.

On the bit-true level, a fixed wordlength and a fixed exponent is assigned to every operand, while the control structure and the operations of the floating-point program remain unchanged. This description is used to analyze whether the fixed-point model fulfills the algorithmic system requirements. Again, this has to be done by means of simulation, so that an efficient fixed-point simulation environment becomes necessary. As for the floating-point level, different concepts exist for a specification on the bit-true level:

- block diagram based modeling, e.g. [3,4,5], where it is possible to convert the floating-point signals into fixed-point signals. All these concepts lack the possibility to look inside the block's functionality, therefore the allowed functionality is restricted to simple operations (such as addition, multiplication) which simply are overloaded.
- textual modeling: the concepts of [3,7] allow to specify variables (not operands) in a bit-true way. While [3] uses a special language (DFL) for describing the algorithm, in [7] the specification is done using a C^{++} program.

On the implementational level, the bit-true model of the algorithm has to be transferred into the best suited target description, either using a HDL or a programming language.

The transformation of the floating-point specification into the bit-true specification is not unique but a complex design space exists. Design criterion is a performance true transformation, so that the bit-true specification fulfills the system requirements. So far, the transformation has to be done manually. This is a tedious, error prone and time consuming process even for a single transformation, for more complex applications accounting for more than 50% of the design time once the floating-point algorithm is fixed [8], as illustrated by the design times included in fig.1.

Although the transformation takes place on the algorithmical level, one can no longer abstract from the target architecture [8]. E.g., for SW-implementations the machine wordlength is fixed, and the minimization of shift operations is a concern. For HW, wordlengths have to be minimized. Therefore, for a typical design (and especially in HW/SW codesign), multiple transformations from the floating-point level to the bit-true level are necessary. Keeping in mind the increase in system complexities and time-to-market pressure, there is a strong demand for an efficient tool support for the transformation.

Recently, Sung [9] presented an automated, simulation based transformation concept. It assumes the floating-point algorithm to be described by a block diagram with all signals of type *float*, and converts it into a bit-true specification by assigning a specific wordlength and the information of the location of binary point to each signal. The procedure is as follows:
1) The range of each signal is analyzed by simulation, so that the appropriate number of integer bits can be evaluated.
2) For some signals the designer might specify the wordlength explicitly. Non-specified wordlengths are determined on a simulation based approach:

- for each signal s_i: set all signals but s_i to a maximum wordlength (64 bits) and determine the minimum wordlength $w_i(min)$ (at least two simulations are necessary, for wordlength $w_i(min) - 1$ and for $w_i(min)$)
- set all signals to their minimum lengths and simulate the system. If the system performance is not reached, sequentially increase the word lengths

This concept is the base for Alta Groups Fixed Point Optimizer within the Hardware Design System (HDS)[10]. The optimization goal is to minimize the hardware costs of the resulting design.

This concept suffers from several limitations:

- The assumed block diagram description is not suited for representing control functionalities.
- The block's functionality is limited to simple operations (addition, multiplication), since it is not possible to modify specifications inside blocks.
- The bit-true specification is limited to a single optimization goal, namely wordlength minimization for HW-designs. It is not suitable for an optimum fixed-point SW-design, and therefore lacks the capabilities for an efficient HW/SW-codesign.
- Most important, system response time which is determined by the number of simulation runs is only acceptable for a small number of unspecified signals. Realistic designs often come with 500 blocks or more, having 1000 signals [11], with each simulation running for hours because of the number of samples that have to be processed to achieve a sufficient statistics.

The demand for an automation of the transfer process from a floating-point to a fixed-point description and the limitations of existing concepts and tools have been the motivation for FRIDGE, a design environment for fixed-point systems. Within this paper we present the concepts implemented in FRIDGE for an automated and interactive transformation of a floating point program written in ANSI-C into a bit-true fixed-point program, based on an interpolative approach. The bit-true specification makes use of a *fixed-C*, an extension to ANSI-C (sec.2). The new *fixed* data type is instantiated by a number of parameters that have to be uniquely defined for all operands. Sec.3 describes the principles how to achieve these parameters. The data dependency analysis that is necessary for the parameter determination is subject of sec.4, with a special focus on control and loop structures as well as pointers. Finally, target specific transformations are briefly described in sec.5.

II *FIXED-C*

To describe a bit-true algorithm, ANSI-C is not suited since the fixed-point data types are restricted to two machine dependent wordlengths (`short, long`) and no explicit exponent can be assigned to an operand (for a detailed discussion see e.g. [9]).

Therefore, we defined *fixed-C* extending ANSI-C by two generic, parameterizable fixed-point data types named *fixed* and *Fixed*.

According to Louden [12], a data type is *a set of values, together with a set of operations on that values having certain properties.*

2.1. Operand Specification

A fixed-point operand is specified by a 3-parameter tuple:

wl	wordlength, number of bits
iwl	integer wordlength, number of bits left of the binary point
sign	**u/s**, unsigned or signed (2's complement) representation

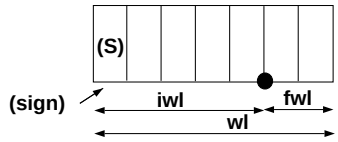

Figure 2. Fixed-point data type specification

fwl indicates the number of fractional bits: *fwl* = *wl* - *iwl*.

2.2. Operator Specification

For both fixed-point data types, *fixed* and *Fixed*, all *float* operators are defined. This is motivated by the fact that fixed-point variables are used within expressions that have been *float* expressions prior to transformation.

2.3. Casting operations

The transformation of one data type instantiation into a different instantiation is possible using *casting* operations:

$$(wl_1, iwl_1, sign_1) \xrightarrow{casting \quad rule} (wl_2, iwl_2, sign_2)$$

The casting rule has to specify how to handle overflow (reduction of the integer wordlength and/or change of the sign) and the reduction of the fractional wordlength.

fixed and *Fixed* offer two modes for overflow handling: *saturation* (**s**) and *wrap around* (**w**).
For the reduction of the fractional wordlength two modes exist, too: *rounding* (**r**) and *truncation* (**t**).
Therefore, four casting rules exist: $cast \in \{sr, st, wr, wt\}$.
The effects of the different casting rules are illustrated by fig.3. Notice that the fractional wordlength reduction is performed prior to overflow handling.

2.4. Assignment time instantiation vs. declaration time instantiation

All existing concepts require the fixed-point specification of a variable at declaration time [7,3]. As a consequence, whenever a specific variable is used in the program, it is of the fixed-point data type that has been assigned to it at declaration time. This concept is by no way suited for

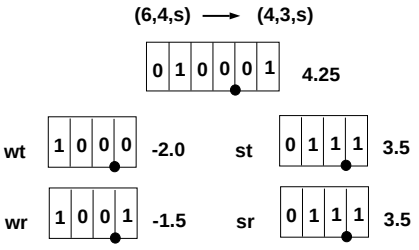

Figure 3. Different casting options

the transition of a floating-point specification to a fixed-point specification: when the designer starts floating-point programming, he does so to abstract from all quantization effects. Especially, he does not care whether different assignments to the same variable would have to match the same fixed-point data type.

In order not to exchange the program structure and to result in maximum flexibility, the *fixed* data type is based on the concept of assignment time instantiation. This is best illustrated by an example:

```
float a,*b,c[8];          fixed a,*b,c[8];
......                    ......

a = *b;                   a = fixed(5,4,s,wt, *b);
......                    ......

a = c[0];                 a = fixed(6,3,s,sr, c[0]);
```

The declaration part declares a to be a *fixed* variable. As long as no assignment is made to a, the fixed-point instantiation of a is undefined and a behaves like a *float* variable. The first assignment instantiates a with data type <5,4,s>, casting the contents of $*b$ to the specified format. With the next assignment, a receives data type <6,3,s>, with no need to exchange the variable's name.

Assignment time instantiation not only allows to keep the floating-point code structures untouched but allows to express fixed-point information within loops and conditional structures in an optimum way. This is explained in more detail in [13]

The second fixed-point data type, *Fixed* (**F**orced **fixed**) has been introduced to allow an efficient interface specification:

```
fixed        *b,c[8];
Fixed<5,4,s> a;
...

a = fixed(5,4,s,wt,*b);
...

a = fixed(6,3,s,sr,c[0]);
```

For every assignment to a *Fixed* variable, FRIDGE analyses whether the assigned parameter triple matches the specification as annotated in the declaration. For the example, the first assignment would be excepted, while the second assignment would result in an information about the data type mismatch. This concept allows to guarantee a unique interface specification throughout the complete transformation process.

Notice that for both data types, *fixed* and *Fixed*, pointers as well as arrays are defined as for the build in data type *float*.

2.5. Data type conversions

fixed-C extends ANSI-C by two fixed-point data types. Therefore it is possible to have hybrid expressions where one operand is of type *fixed*, while the other operand is of an ANSI-C data type.

For operations having a *fixed* and a *float* operand, the *fixed* operand is transferred to a *float* operand first, and a *float* operation is performed. A *float* operand can be casted to a *fixed* operand using the casting rules as described above.

The existing ANSI-C fixed-point data types (*short*, *int*, *long*) can be seen as subtypes of *fixed* with regard to their arithmetic behavior with the exception of overflow handling: according to ANSI-C [14], the overflow mode is machine dependent. This is not acceptable for a bit-true specification, therefore it has been fixed to wrap around. Notice that the logic operators are still restricted to integer-type operands.

Since *fixed-C* covers ANSI-C and because of the new assignment time instantiation concept, the complete control structure of the initial (floating-point) program might remain unchanged. Therefore, the transformation of a floating-point program written in ANSI-C into a bit-true program written in *fixed-C* can be identified to assign a parameter triple <*wl,iwl,sign*> to every operand and to specify the casting rules for the explicit instantiations.

Notice that *fixed-C* is pure C^{++}, making FRIDGE a framework that is entirely based on standard language interfaces.

III THE INTERPOLATIVE APPROACH

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.

Therefore, we propose an alternative design flow, denoted the **interpolative approach** which is illustrated by fig.4.

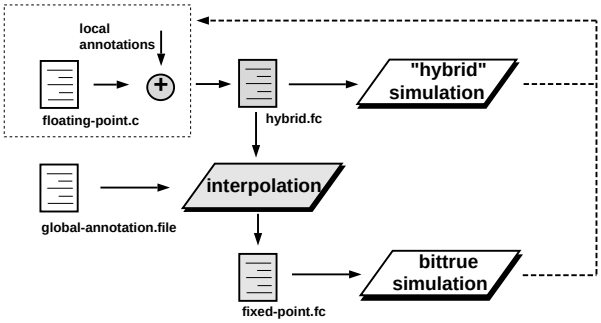

Figure 4. Design flow based on interpolation

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

   Local annotations are not restricted to complete fixed-point specifications specifying all 3 parameters (wl,iwl,sign) explicitly, but it is possible to inform about:

   - the operand's range
   - the operand's mean and variance
   - the maximum absolute error that is acceptable due to quantization
   - the maximum relative error (relative to the maximum value the operand can take) that is acceptable due to quantization

   These informations can be utilized by the interpolator.

2. Simulation:
   Simulation serves for two purposes:
   first it allows the designer to check 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. Second, by simulation it is possible to collect additional information that can serve as local annotations. This can be monitoring of range, mean or variance for some or all operands (what has to be traded with simulation time increases), displaying histograms for user specified operands or overflow detection.

   Using the simulation environment HYBRIS [2] which is an integral part of FRIDGE, it is possible to automatically back-annotate the collected information so that it becomes the base for interpolation.

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 might be illustrated by an example:
   $a = b + c$
   For a, *sign* and sufficient integer wordlength *iwl* 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 $fwl(a) = max\{fwl(b), fwl(c)\}$.
   Obviously, information about ranges and fractional wordlengths can be determined by the various local annotations as described above.

   (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.
   Examples for global annotations include:
   *global_cast(***cast***)*: if no local annotation about the casting mode is available, take mode **cast**.
   *global_wl_max(***max, default***)*: whenever worst case estimation leads to a wordlength exceeding **max**, reduce it to **default**.
   For more information about global annotation options, refer to [15]. 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.

[2] HYBRId Simulator

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.

The interpolative approach relies on a unique compile time identification of the information that is available for each operand. Therefore, a powerful data flow analysis becomes necessary.

IV DATA DEPENDENCY ANALYSIS

4.1. Control-Dataflow Graph

Unless the functionality that has to be transferred into a bit-true specification is a main function, it is assumed to be a function that is called within a loop of unknown size.

The interpolative approach depends on the possibility to identify the information to be propagated (range information or parameter information) at compile time. Therefore, it is necessary to analyze the data dependencies by a program interpretation. For the representation of data dependencies a modified control-dataflow graph (CDFG) [16] is introduced. Fig.5 shows the CDFG for a simple example.

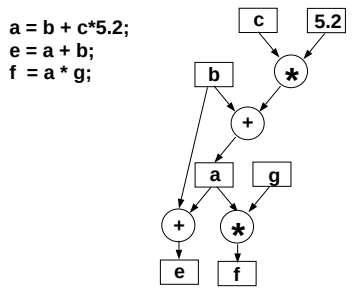

Figure 5. Dataflow representation

A CDFG consists of nodes which represent the operators, boxes identifying variables or constants. Arrows either represent assignments to variables and usages of variables as an operand or the usage of an operation result as an operand for the next operation (e.g. the result of $c * 5.2$ is not assigned to a temporary variable but becomes an operand of the addition). Each variable corresponds to a storage location in a virtual memory model. A variable is supposed to keep the assigned value as long as the last operation using it as an operand has been executed (in the example,

a is supposed to keep its value until both operations are executed). Prior to this, no new assignment to a storage location is allowed

So far, this representation does not include parameter and casting information. Therefore, the CDFG notation has to be extended, as it is expressed by fig.6

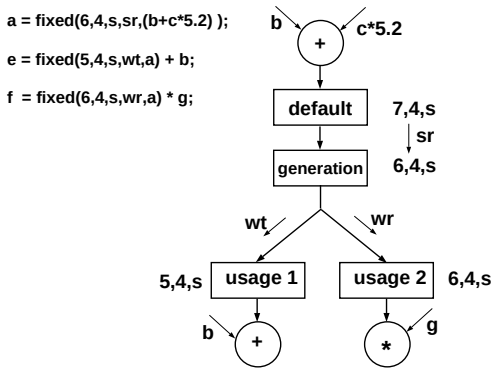

Figure 6. Extended CDFG

Default denotes the data type instantiation resulting from worst case estimation. This instance might be forced to a new instantiation when the result is assigned to a. The transformation is done according to the casting rules of a, here sr. Only the generated format is written to the storage location and is available for further processing.

When a is used as an operand, different parameter tuples can be assigned for each usage, resulting from different data type instantiations: when a is used as an operand of the addition, a wordlength of 5 is sufficient, but when used in the multiplication a wordlength of 6 is required.

4.2. Conditional structures

For a conditional structure, at compile time in general it is not possible to decide which branch is to be executed. As a consequence, depending on the executed branch different parameter triples might be assigned to a variable. This is contrary to the requirement of a unique information assigned to each operand. For the interpolative approach, which is based on worst case estimations, a unification of the information is necessary prior to the first access to the storage location outside the conditional structure. Fig.7 illustrates the extension to the CDFG.

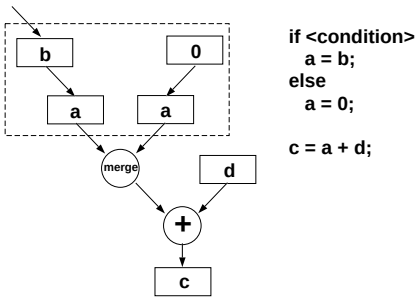

Figure 7. Dataflow representation for a conditional statement

The *merge* node combines the information of both operands:

$$\begin{aligned} fwl_a &= \max\{fwl_{a1}, fwl_{a2}\} \\ min(a) &= \min\{min(a1), min(a2)\} \\ max(a) &= \max\{max(a1), max(a2)\} \end{aligned}$$

The *merge* node guarantees that independent from the execution of any of the branches no information gets lost.

4.3. Loop structures

3 classes of loops can be identified:

1. loops of fixed length
2. loops of data dependent length, with N_{max} the maximum number of iterations known
3. loops of data dependent length with no information about the number of executions

The type of loop is identified by FRIDGE during the program interpretation, independent of its description by a *for, while* or *do-while* construct.

Fixed length The loop body can be unfolded, resulting in sequential code.

Data dependent length, N_{max} is known The loop can be unfolded, too. Fig.8 shows the CDFG for this constellation. Different to the loop of fixed length, now it is not

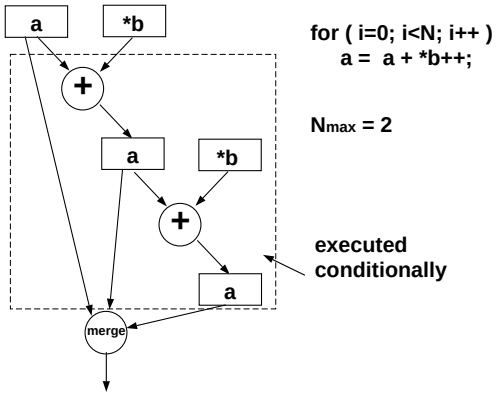

Figure 8. CDFG for a loop of maximum, data dependent length

possible to identify the final assignment to a storage location before leaving the loop. Therefore, prior to reading from a storage locations that might has been written to inside the loop, all possible information has to be combined using a *merge* node.

Data dependent length, no information For this constellation, no loop unfolding is possible. Again, it has to be guaranteed that independent from the number of iterations no information can get lost and a unique information is propagated. As a consequence, in the bit-true specification independent from the iteration a unique parameter triple is assigned to each operand. Therefore, in the CDFG only one loop iteration has to be represented, as illustrated by fig.9.

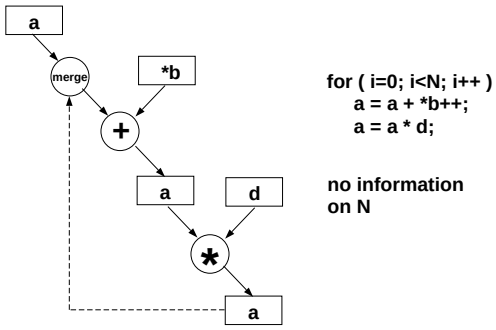

Figure 9. CDFG for a loop of data dependent length, no information on N

The dashed arrow (in the following called *static arrow*) indicates that for the next iteration the last instantiation of a within the loop becomes the instantiation of a at the beginning of the loop. The *merge* node indicates that either one of the instantiations becomes the input for the next loop iteration, and that both informations have to be combined.

4.3.1. Static Variables

Static variables provide private, permanent storage within a single function. Since it is assumed that the function is called within a loop of unknown size, static variables have to be treated as variables in a loop. This as well is represented by a static arrow (actually, the static variables motivated this notation) if the static variable is used as an operand prior to its first assignment. In the sequel, all variables resulting in static arrows in the CDFG shall be denoted static variables.

4.3.2. Pointer

Within ANSI-C, indirect addressing using pointers is a powerful mechanism for an efficient algorithm description. This concept is fully supported by FRIDGE, data types *fixed,Fixed* can handle pointers in the same way as the numeric data types already defined in ANSI-C.

For data type analysis, pointer analysis is a major challenge. During code interpretation, all possible storage locations a pointer might read from and might write to have to be identified.

For a reading pointer, the information inherent to all possible storage locations the pointer might read from have to be combined, using a *merge* node. If it is not possible to identify the referenced storage location, the designer is informed that he has to assign explicit information to the operand referenced by the pointer so that the operand is forced to a unique representation.

For a writing pointer, there might be different storage locations the result is assigned to. In case the storage location is not unique, at compile time it can not be guaranteed whether the potential storage location keeps its information or is overwritten by the pointer information. Therefore, both informations (those previously assigned to the storage location and those of the pointer) have to be combined using a *merge* node prior to using the contents of the storage location as an operand.

V ADDITIONAL FEATURES OF FRIDGE

Starting from the presented bit-true specification, FRIDGE includes target specific compilers that accept *fixed-C* as its input specification.

In [15], a compiler for generating bit-true or performance-true fixed-point ANSI-C code is presented. This compiler is intended for generating code that existing C-compilers can handle. In combination with the presented approach it allows a comfortable transition from a floating-point program written in ANSI-C to a performance-true fixed-point program in ANSI-C. This concept proved to be very efficient as is illustrated by the example of a Wiener filter:
The initial float format comes with 135 lines of C-Code, resulting in more than 2000 operations due to nested loops and conditional structures. The intended target platform has been a Motorola DSP coming with a word length of 24 bit which should be utilized in the best way to reduce quantization noise. Interpolation has been possible simply by specifying the input variables and the upper bound for the wordlength. Within less than 8 seconds an integer ANSI-C code has been achieved, including 500 shift operations, resulting in a performance degradation of less than 0.3dB compared to the initial floating point algorithm.

A compiler for generating behavioral VHDL suitable as an input specification for a behavioral synthesis tool [17] is currently under construction. This will extend the capabilities of FRIDGE to a integrated HW/SW codesign environment.

Future projects include the extension of existing ANSI-C compilers to the extended *fixed-C* syntax.

VI CONCLUSION

Data type conversion of a floating-point specification to a fixed-point specification has been identified to be a time consuming and error prone process. Within this paper we have presented FRIDGE, a tool that allows to achieve a bit-true specification of an algorithm starting from a floating-point description in ANSI-C. It is based on an interpolative approach that allows the designer to bring in all his specific knowledge but takes away the burden of specifying all operands explicitly. This allows to analyze different design option within a short time. The tool handles complicated control and loop structures and is capable of pointer arithmetic, too. This results in a most flexible design environment for various applications whenever fixed-point implementations are required.

REFERENCES

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

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

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

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

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

[6] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A platform for heterogenous simulation and prototyping," in *Proc. 1991 European Simulation Conf.*, (Copenhagen, Denmark), June 1991.

[7] 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.

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

[9] 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.

[10] Alta Group Inc., 555 N.Mathilda Ave., Sunnyvale,CA 94086, *HDS User's Manual.*

[11] 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.

[12] K. Louden, *Programming Languages.* Boston: PWS-KENT Publishing Company, 1994.

[13] M. Willems, V. Bürsgens, H. Keding, and H. Meyr, "FRIDGE: An Interactive Fixed-Point Code Generation Environment for HW/SW CoDesign," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, (München), Apr. 1997. accepted for publication.

[14] B. W. Kernighan and D. M. Ritchie, *The C Programming Language (second edition).* Prentice Hall, 1988.

[15] M. Willems, V. Bürsgens, H. Keding, and H. Meyr, "Automatic Fixed-Point C-Code Generation From Floating-Point Programs," in *Proc. Int. Conf. on Signal Processing Application and Technology*, (San Diego), Sep. 1997. submitted for publication.

[16] G. De Micheli, *Synthesis and optimization of digital circuits.* Mc Graw-Hill, 1994.

[17] D. W. Knapp, *Behavioral Synthesis.* Prentice Hall, 1996.