INTEGER CODE GENERATION FOR THE TI TMS320C62X

Martin Coors, Holger Keding, Olaf Lüthje, Heinrich Meyr

Institute for Integrated Signal Processing Systems
Aachen University of Technology, 52056 Aachen, Germany
{coors,keding,luethje,meyr}@iss.rwth-aachen.de

ABSTRACT

This paper presents a methodology which enables the generation of C62x optimized fixed-point C-code from a floating-point description of an algorithm. The FRIDGE design environment transforms floating-point ANSI-C code with local fixed-point annotations into an internal bit-true representation. From this representation we generate C62x optimized integer C code utilizing the code transformation techniques illustrated in this paper. A benchmark is presented comparing the efficiency of the generated code with C67x C-code, C62x floating-point emulation and generic integer ANSI-C code.

1. INTRODUCTION

Algorithm design for digital signal processing systems typically starts in the floating-point domain to abstract from implementation effects. On the other hand the implementation is usually done on fixed-point devices due to the advantage of fixed-point systems in terms of power consumption, chip size, and price per unit.

Prior to the actual system implementation a transformation from the floating-point to a fixed-point system is necessary, i.e an exploration of the fixed-point design space with respect to quantization noise, performance, and operand word lengths.

Modeling the fixed-point behavior of signal processing algorithms 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 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.

Generic fixed-point data types implemented in C++ based libraries [1, 2, 3] offer a high modeling efficiency. They also supply various casting modes for overflow and quantization handling. Nevertheless these fixed-point data types are not suited for DSP implementation, as the currently available DSP compilers do not support C++ fixed-point data types for fixed-point design. 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.

ANSI C-based fixed-point libraries such as the ETSI basic arithmetic operations used in [4] offer a set of two or three fixed-point data types and some data path elements like they are frequently encountered on programmable DSPs. Here, the lack of flexibility restricts the fixed-point design process.

Thus a significant gap in the design flow is evident: 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.

The FRIDGE C62x design environment presented in this paper 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 C62x has been chosen as a target device for several reasons:

- The C62x is a popular high performance fixed-point DSP platform which is intended to be programmed in C. An efficient C compiler for the C62x is available from TI.

- The C62x C compiler allows direct access to the built-in architectural features of the processor via intrinsics. Thus otherwise computationally expensive fixed-point operations can be directly mapped to the hardware. Additional information about the control- and data-flow in the algorithm can be passed on to the compiler.

- An evaluation of the C62x compiler [5] utilizing the DSP-stone benchmarking methodology [6] has shown that dramatic improvements of the performance of the compiled code can be achieved by target specific code restructuring and optimization. Some of the optimization techniques developed during the evaluation have also been implemented in the FRIDGE C62x design environment utilizing the information present in the internal representation of the algorithm.

This paper is organized as follows: section 2 will briefly introduce the FRIDGE fixed-point design environment which forms the basis for the C62x code generation. In the main section 3 the transformation process is presented. In section 4 we provide comparative benchmarking data for various signal processing kernel functions. Section 5 concludes the paper.

2. FRIDGE DESIGN ENVIRONMENT

The work presented in this paper is based on the **F**ixed-Point P**R**ogramm**I**ng **D**esign **E**nvironment (FRIDGE) which supports the designer in the floating-point to fixed-point transformation process. FRIDGE relies on data flow analysis and information propagation

as described e.g. in [7, 8] to transform a signal processing algorithm into a fully bit-true representation. The transformation is based on analytical range and precision propagation of fixed-point operands. Fig. 1 highlights the design flow. Starting point is a floating-point description of the algorithm in ANSI-C. First the designer adds *local fixed-point annotations* to the algorithm, specifying the bit-true format for input- and key-operands. The local annotations are specified using the fixed-point data types of *SystemC*. The result is a *hybrid description*, i.e. parts are specified in fixed-point whereas the majority of the operands still remains floating-point. In a first step the FRIDGE front end parses the hybrid description into an intermediate representation (IR). Then range and precision propagation is performed to determine the bit-true format for all 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 have been described in [9].

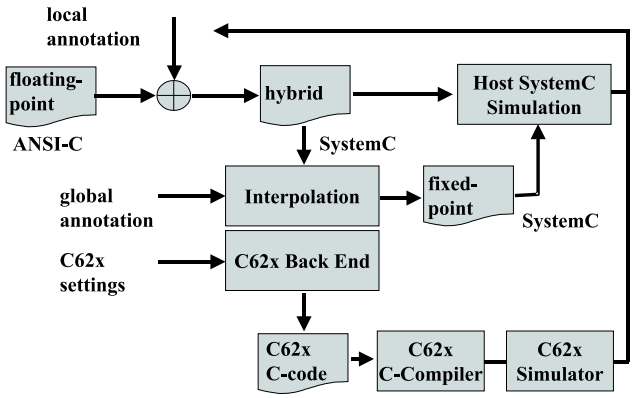

Fig. 1. Fixed-Point system design flow with FRIDGE

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 the transformation steps performed in the FRIDGE back ends that target different languages and platforms. The *SystemC back end* transfers the IR into a quantized algorithm using the *SystemC* fixed-point data types.

For the C62x integer code generation, further transformation steps are necessary. The *C62x back end* also requires additional information collected during the control- and data flow analysis. After a number of IR refinements, a C-code 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-based algorithm yield bit-by-bit identical results.

3. TRANSFORMATION

3.1. LBP Alignment

A fixed-point operand is specified by a 3-tuple $< wl, iwl, sign >$ where *wl* is the *word length* of the operand, *iwl* the *integer word length* and *sign* the *sign encoding* used. For the embedding of a fixed-point operand into a register of the DSP with the machine word length *mwl* the minimum requirement is

$$mwl \geq wl = iwl + fwl \tag{1}$$

Fig. 2 illustrates different options for embedding an operand with a word length of 5 bits into a given machine word length *mwl* of 8. Obviously for $mwl > wl$ a degree of freedom for choosing the **location of binary point** *lbp* exists:

$$mwl - iwl \geq lbp \geq wl - iwl = fwl \tag{2}$$

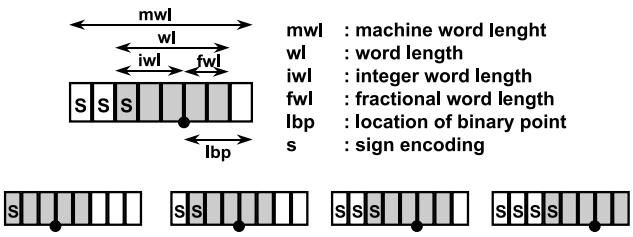

Fig. 2. Embedding a 5 bit word into an 8 bit register

Besides this degree of freedom there are also a number of limitations for the selection of the *lbp*:

- **Operation constraints**: certain operations require the *lbp* to be at the same position for both operands, e.g. additions or comparisons.

- **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.

- **Interface constraints**: for interface elements, e.g. function parameters or global variables the *lbp* must be predefined. Otherwise the data written to or read from these data elements can be misinterpreted by the calling function(s).

The *lbp* alignment algorithm implemented in the FRIDGE C62x back end is designed to take advantage of the degree of freedom described by equation 2, while meeting the constraints specified above. For meeting these constraints and maintaining the consistency of the *lbps* one requires precise information about the control and data flow of the algorithm. To obtain this information we used the data flow analysis method described in [9]. The data flow information is represented basically as *define-use (du) chains* and *use-define (ud) chains* [10], but with additional and more accurate information about ambiguous control flow.

As initial step for the embedding we choose *lbp=fwl* for all operands . Thus all operands are *right aligned*. In an iterative process, the data flow information is used to adjust the *lbps* by insertion of shift operations to meet

1. the interface constraints
2. the control- and data flow constraints
3. the operation constraints

For each constraint the algorithm terminates when all conditions are fulfilled and the *lbps* did not change during the last iteration. The embedding of constants can be done in a way that the required shift operations when using them are minimized.

3.2. Cast Mode Transformation

Cast operations can reduce or limit the word length at the MSB side of a word (*overflow handling*) or at the LSB side of a word (*quantization handling*). Fixed-point libraries like in *SystemC* offer various generic overflow and quantization handling modes which make them an efficient means of modeling of fixed-point systems. For DSP implementation the cast operations 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:

3.2.1. 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 utilizing plain C code first tests if the range of data type is exceeded. If so it sets the resulting value to the minimum or maximum of this type:
$MAX = 2^{iwl+lbp-1} - 2^{lbl-fwl}$
$MIN = -2^{iwl+lbp-1} + 2^{lbl-fwl} - 1$
Thus the code construct generated is:

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

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

On the C62x the `_sshl` intrinsic (**S**aturating **S**hift **L**eft) 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.

3.3. Data type Selection

The final step in the transformation process is the selection of suitable integral data types for fixed-point variables. The internal bit-true specification of the algorithm features arbitrary word lengths. With the *SystemC back end* this is no problem, since the *SystemC* data types are generic and may be of any bit length that is required. The C62x back end on the other hand only has the limited pool of the built-in data types supported by the C62x, `char`, `short`, `int`, and `long`. Similar to the *lbp* alignment algorithm, several constraints for the mwl have to be met: of course $iwl + lbp \leq mwl$ must always be fulfilled. Additional constraints occur in case of arrays and pointers: all array elements must be of the same type, a pointer and its target location must be of the same type. In case of aliasing of data elements, e.g. a pointer pointing to different data elements, advanced algorithms for data type selection are implemented in the FRIDGE C62x back end.

In contrast to the concepts presented in [11], the FRIDGE design environment determines the fixed-point parameters by an analytical approach combined with simulation data instead of pure simulation based range estimation. In the FRIDGE C62x back end, the choice of the *mwl* for each variable is flexible, based on the values of *iwl* and *lbp*.

4. EXPERIMENTAL RESULTS

We have benchmarked the cycle-count performance of the generated C62x integer C-code using typical signal processing kernel functions:

- **FIR** 17-tap FIR filter
- **DCT** 8x8 JPEG discrete cosine transformation
- **Autocorr** 25 elements 5th order autocorrelation
- **IIR** 3rd order IIR filter
- **Matrix** 4x4 matrix multiplication
- **Dotprod** 64 element dot product

The code has been translated using TI's C6x compiler version 4.0[12] and the performance has been compared with three reference codes:

- **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.

- **C62x floating-point emulation**

The floating-point emulation library which is part of the C62x compiler's runtime library allows the user to perform floating-point arithmetic on the C62x processor. The floating-point operations are executed as function calls.

- **C62x integer ANSI-C code**

A special version of 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.

	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

Table 1. Cycle Count

Table 1 presents the benchmarking results for the six kernel functions. Figure 3 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 1 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. E.g. 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.

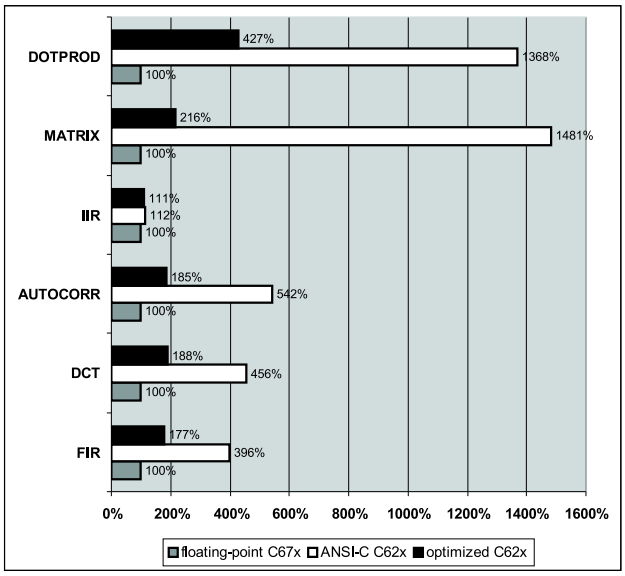

Fig. 3. Cycle Count Relative to Floating Point Code

5. CONCLUSIONS

In this paper we have presented a design environment for floating-point to fixed-point conversion and for the generation of optimized C62x C-code using integral data types. The concept and the necessary code transformation steps have been presented. To proof the concepts introduced in this paper we have implemented a software tool and we have benchmarked the generated C62x code against floating-point reference code running on a C67x DSP and ANSI-C integer code.

The FRIDGE C62x design environment enables a seamless design flow 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. A tool which generates C62x C-code with scaling shifts, integer constants and cast operations frees the designer from the tedious and error prone task of manually porting the code to the C62x platform and reduces time to market.

Future research work will focus on advanced loop optimization techniques for the C62x to exploit the features of the C62x processor and its C-compiler.

6. REFERENCES

[1] 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, Nov. 1995, pp. 197–206.

[2] Frontier Design Inc., 9000 Crow Canyon Rd., Danville, CA 94506, USA, *A|RT Library User's and Reference Documentation*, 1998.

[3] Synopsys, Inc., CoWare, Inc., Frontier Design Inc., *SystemC User's Guide, Version 1.0*, 2000.

[4] Recommendation GSM 06.10, *GSM Full Rate Speech Transcoding*, ETSI/TC SMG, 1992.

[5] M. Coors, O. Wahlen, H. Keding, O. Lüthje, and H. Meyr, "C62x Compiler Benchmarking and Performance Coding Techniques," in *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT)*, Orlando, Nov. 1999.

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

[7] M. Willems, V. Bürsgens, H. Keding, T. Grötker, and H. Meyr, "System Level Fixed-Point Design Based on an Interpolative Approach," in *Proceedings of the Design Automation Conference (DAC)*, Anaheim, Jun. 1997, pp. 293–298.

[8] H. Keding, M. Willems, M. Coors, and H. Meyr, "FRIDGE: A Fixed-Point Design and Simulation Environment," in *Proceedings of the European Conference on Design, Automation and Test (DATE)*, Paris, Feb. 1998, pp. 429–435.

[9] O. Lüthje, H. Keding, and M. Coors, "High Performance Code Analysis by Abstract Execution," in *Proc. of DSP Germany 2000*, Munich, Oct. 2000.

[10] Michael J. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing, Redwood City, CA, 1996.

[11] Jiyang Kang Ki-Il Kum and Wonyong Sung, "A floating-point to integer c converter with shift reduction for fixed-point digital signal processors," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 1999, pp. 2163–2166.

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