Computer Aided Implementation of Complex Algorithms on DSP's Using Automatic Scaling

Korina KASSAPOGLOU* and Martin VETTERLI**

* Ecole Polytechnique Fédérale de Lausanne
16, Chemin de Bellerive
CH-1007 Lausanne, Switzerland

** Center for Telecommunications Research
Columbia University
New York City, NY 10027

Abstract. A methodology for transforming complex floating-point algorithms into correct fixed-point DSP programs is presented. In particular, an automatic scaling scheme leading to overflow-free programs is described. Depending on the application, scaling generated automatically may either perfectly fit, or be modified in order to substantially improve accuracy. Each case is illustrated by an example (FFT and Recursive Least-Squares algorithms).

1. Introduction

Most current digital signal processors (DSP's) use fixed-point arithmetic, thus leading to scaling and precision problems when implementing complex algorithms. Therefore, development methodologies are required in order to cope with the problems encountered [1,2].

A computer-aided implementation method is briefly presented in this paper. Starting with a floating-point complex-structured algorithm and following a stepwise refinement approach, the corresponding optimized DSP code is obtained. Special attention is given to the scaling assignment issue. A matrix expression that controls evolution of the scaling throughout an algorithm is first constructed. Then it is used, in order to determine scale factors automatically, with respect to an overflow-free configuration.

The implementation method is briefly described in Section 2. In Section 3, we show how to control the scaling throughout an algorithm and we then derive the automatic scaling scheme Two different applications are described in section 4: a FFT algorithm is successfully transformed into assembler using the results of section 3, while scaling obtained automatically is modified in the case of a recursive least-squares algorithm (RLS) in order to increase its accuracy (thus leading to potential overflow risks). Note that the FFT is an FIR algorithm, while the RLS is an IIR one, where variables can vary substantially.

2. The methodology

The proposed Computer Aided Implementation method is a top-down step-wise refinement approach to efficient implementation of complex structured digital signal processing algorithms into machine code for a signal processor. The main steps are shown in Figure 1.

Starting with a floating-point program that describes the algorithm, the first transformation simplifies its structural complexity. Any recursion is then removed, while external information controls code linearization, which is introduced for execution speed purposes. In order to structure computations in the way a processor executes them, all arithmetic expressions are decomposed into simple operations that involve only two operands and assign the output to a variable. The next and quite important step performs the floating- to fixed-point conversion, according to the theoretical basis presented in Section 3. The final step translates the fixed-point Pascal code into the assembler of the target processor, mapping the algorithm into a series of macros, most of which have already been optimized and are available in a library.

At each stage of the methodology, the resulting program is error-free and executable. Additional external information required at each step must be validated using crosschecks. Any input program is required to be compilable, so that syntax tests are never needed. The whole development is performed in high level language (Pascal) and is processor-independent until the last transformation.

CH2396-0/87/0000-1027 $1.00 © 1987 IEEE

3. The automatic scaling scheme

3.1 Notation

Let x be a real number. We adopt the following representation:

$$x = \overline{X}\ 2^{sf(X)} \qquad (1)$$

where $\overline{X}$ is a signed number less than 1, and $2^{sf(X)}$ is the scale factor of x. Since fixed-point arithmetic deals with integers, $\overline{X}$ is factored with advantage as:

$$\overline{X} = X\ 2^{-[nb(X)-1]} \qquad (2)$$

where nb(X) denotes the wordlength used and X is an integer.

3.2 Scaling after an arithmetic operation

Before introducing the scaling mapping of an algorithm, it is imperative to clearly express the scale exponents of any operation output.

Truncation. If l least significant bits (LSB) and m most significant bits (MSB) are truncated from X, integer representation of x according to (1)-(2), this latter will be represented by:

$$x = Y\ 2^{-[nb(Y)-1]}\ 2^{sf(Y)} \qquad (3)$$

with

$$Y = X/(2^l)\ \text{, unless an overflow occured}$$

$$nb(Y) = nb(X) - m - l$$

$$sf(Y) = sf(X) - m \qquad (4)$$

Note that rounding can always be expressed as the addition of half the least significant bit, followed by a truncation.

Multiplication (c = ab). The product holds a priori on nb(A)+nb(B)-1 bits. If l LSB and m MSB are truncated from it,

$$P = AB/(2^l)\ \text{(if no overflow when truncating)}$$

$$nb(P) = nb(A) + nb(B) - 1 - m - l$$

$$sf(P) = sf(A) + sf(B) - m \qquad (5)$$

Addition (/Subtraction) (c = a+b). When adding, scale factors of operands must be equal. If this is not the case, they must be adjusted using shifts: the variable with smaller scale factor will be shifted right, so that no overflow is caused. Assuming that $\overline{A}$ has more bits than $\overline{B}$, the sum holds on nb(A)+1 bits. According to the adopted notation, operands are obviously left-aligned. If l LSB and m MSB are truncated,

$$nb(C) = nb(A) + 1 - m - l$$

Let $u = sf(A) - sf(B)$

If $u \geq 0$,

$$C = (A + B\ 2^{[nb(A)-nb(B)]}\ 2^{-u})\ /\ 2^l$$

$$sf(C) = sf(A) + 1 - m$$
$$= sf(A)/2 + [sf(B)+u]/2 + 1 - m \qquad (6a)$$

If $u < 0$,

$$C = (A\ 2^u + B\ 2^{[nb(A)-nb(B)]})\ /\ 2^l$$

$$sf(C) = sf(B) + 1 - m$$
$$= [sf(A)-u]/2 + sf(B)/2 + 1 - m \qquad (6b)$$

Division (c = a/b). Division is treated in a slightly different way. First, the wordlength of a division output depends on the desired accuracy and thus on the algorithm used. Secondly, in the case where A is always inferior to B, $\overline{A}$ is may be left-shifted, giving:

$$a = A\ 2^m\ 2^{-[nb(A)+m-1]}\ 2^{sf(A)} \qquad (7)$$

In general, the division output is given by:

$$C = (A\ 2^m\ \text{div}\ B)\ 2^{[nb(C)-nb(A)]} \qquad (8)$$

and thus,

$$sf(C) = sf(A) - sf(B) - m + nb(B) - 1 \qquad (9)$$

Equations (5), (6) and (9) give the scale exponent of any operation output. In the rest of this section, we assume that we are dealing with an algorithm, thus with a set of operations. The main idea is to group all such successive equations within a single expression and get each obtained scale exponent in terms of scale exponents of input variables only.

3.3 Evolution of scaling throughout an algorithm

We are given an algorithm A having n input variables x_i and containing p operations. We assume that each operation involves only two variables and produces a new one. Let S be a (n+p)-vector containing the scale exponents of all n+p variables. These latter are ordered as follows: first the n input variables, then the p operation outputs in the same order as they appear in A. The vector S is given by:

$$S - \mathbf{D}|\mathbf{E}\ S| = \mathbf{A}\ [sf(X_i)] - \mathbf{B}\ [m_j] + \mathbf{C} \qquad (10)$$

where

. $[sf(X_i)]$ contains the scale exponents of input variables only,

. $[m_j]$ gives the number of MSB truncated after each operation (and thus represents a kind of overflow-risk index).

A, **B**, **C**, **D** and **E** are matrices(/vectors) that depend on the algorithmic structure only. (Dimensions: (n+p)xn, (n+p)xp, (n+p)x1, (n+p)xp and (p)x(n+p)). They are constructed using five functions f, g, h, l and o. Functions f, g, h and l map each of the n+p variables to, respectively, an n-vector, a p-vector, a real and a p-vector. Function o maps each of the p operation outputs to a (n+p)-vector. Table 1 shows how these functions are constructed using the rules given in paragraph 3.2, and how matrices **A**, **B**, **C**, **D** and **E** are defined.

The right hand side of equation (10) contains contributions of, respectively, scale exponents of input variables, numbers (m_j) of MSB truncated (or lost) after each operation j and some constants that depend on the kind of operations. Correction $\mathbf{D}|\mathbf{E}\ S|$ upon S on the left hand side is non zero if and only if additions with scaling incompatibility are encountered. When adjusting scaling, no overflow may occur. Therefore, a comparision of two scale exponents is needed to indicate which of the two operands is to be shifted. This may create a dependence of **E** upon S, which is avoided by using the absolute values.

It is important to note that, by construction, matrices **B**, **D** and **E** are lower triangular. In addition, **E** has zeros along its diagonal. Therefore, equation (10) is solved in a straight-forward way, if one starts at line 1 and proceeds line by line: correction $\mathbf{D}|\mathbf{E}\ S|$ at line k uses the first k-1 elements of S only.

3.4 Automatically generated scaling

Scale exponents of S are actually parametrized by the p integers m_j, since matrices **A**, **B**, **C**, **D** and **E** are completely known once equation (10) is established and scale exponents of input variables are specified.

It is obvious that, if we wish an overflow-free configuration, none of the most significant bits may be truncated. Thus, all m_j's must be set to zero:

$$m_j = 0 \quad \text{for} \quad 1 \leq j \leq p \qquad (11)$$

This gives a solution S_0 to equation (11). As mentioned in the previous paragraph, this solution is easily obtained if one proceeds line by line.

In the case of algorithms where variables have small dynamic ranges and provided that the input variables are represented with maximum accuracy, the solution S_0 is quite satisfactory. It realizes a nice trade-off between two basic requirements, namely good accuracy and correct operation of the algorithm (no overflow, for instance).

In the case of algorithms where dynamic ranges of variables vary substantially, the solution obtained with (11) may be too severe, deteriorating the accuracy. If some statistics on variable ranges are available, one may introduce a limited number of low-probability overflows, thus improving the accuracy. This means that some values m_j are modified. The new resulting values of S can be obtained incrementally, using:

$$S - \mathbf{D}|\mathbf{E}\ S| = -[S_0 - \mathbf{D}|\mathbf{E}\ S_0|] - \mathbf{B}\,[\Delta m_j] \qquad (12)$$

3.5 How to use these results

When generating the Pascal program that supports fixed-point computation simulations according to Figure 1, all that is needed at operation j is m_j and the jth line of ES. If this is non zero, then a shift must precede this operation. The value of $sf(X_{n+j})$ will inform one how to read the contents of the corresponding memory cell. If decisions upon integers m_j are not satisfactory, one may change them and only solve the last equation (12).

4. Applications

The automatic scaling scheme was applied successfully to a 16-point FFT algorithm [3]. This is a FIR algorithm, and thus there are no constraints on output variable representations. Assuming that the input signal is normalized, the scale exponents of initial signal samples are equal to zero. Since coefficients are less than 1, their scale exponent is also zero. Solving equation (10) using the automatic scaling scheme (11), we determined that each stage increased the scale exponents of samples by 1. After the four required stages, scale exponents of FFT samples were equal to 4. The obtained FFT operates correctly, with good accuracy. (Actually, the accuracy can be improved by using a more detailed analysis taking into account that a DFT is a multidimensional rotation.)

The automatic scaling scheme was also applied to an IIR algorithm, namely a recursive least-squares identification one. When updating system parameters, one uses a covariance matrix **P**, which is also updated recursively. In order to get convergence, matrix **P** must be initialized to a high value. We initialized it to a value near 100 and fixed its input scale exponent at 7. After the scheme (11) was applied, the parameter scale exponent increased by 16, while that of matrix **P** raised by 18 after one recursion!

However, we know that once convergence is reached, parameter values do not vary much and values of P decrease. This means that there exists link equations between the variables and, of course, they are not taken into account by (10)-(11). Based on elementary statistics upon variable ranges, we assigned values greater than zero to the m_j and obtained the same scaling as in [4]. Scale exponents of recursive variables thus remain constant through recursions. Fixed-point simulations confirmed that, in reality, overflow probability is very small.

5. Conclusions

In this paper, we concentrated on the scaling issue encountered when an algorithm is implemented in fixed-point arithmetic. Equations that express the scale factor of a variable, which is the output of a simple operation, are presented. They lead to a matrix expression that controls the scaling throughout an algebraic algorithm: scale exponents of all variables are obtained in terms of scale exponents of input variables only and the numbers of most significant bits truncated (or lost) after every operation. An automatic scaling scheme, with respect to an overflow-free configuration is then derived.

It is obvious that direct manipulation of these equations is quite tedious. On the contrary, the context of a Computer Aided Implementation method is greatly attractive. We are on the process of developing programs to support this.

Finally, the automatic scaling scheme is applied to two examples: an FIR algorithm (FFT) and an IIR one (Recursive Least-Squares).

References

[1] L.R.Morris, "Automatic Generation of Time Efficient Digital Signal Processing Software", IEEE Trans. on Acoustics, Speech and Signal Processing, Vol. ASSP-25, pp.74-78, Feb. 1977.

[2] H.Hanselmann, "A Concept for Mostly Automatic Implementation of Control Algorithms", IEEE Computer Aided Control System Design Symposium, Arlington, Virginia, Sept. 1986.

[3] M.Vetterli, E.Debourse, M.Kardan, "Fast Fourier Transforms on the TMS 320 Signal Processor", Proceedings des Journées d'électronique 1985, Lausanne, Suisse, Oct. 1985.

[4] K.Kassapoglou, P.Hulliger, "Implementation of Recursive Least Squares Identification Algorithms on the TMS 320", Proceedings of the EUSIPCO-86 Conference, The Hague, Sept. 1986.

[5] K.Kassapoglou, "Control on Scaling throughout an Algorithm Leading to an Automatic Scaling Scheme", Internal Report of the Laboratoire d'Informatique Technique, Ecole Polytechnique Fédérale de Lausanne, Switzerland.

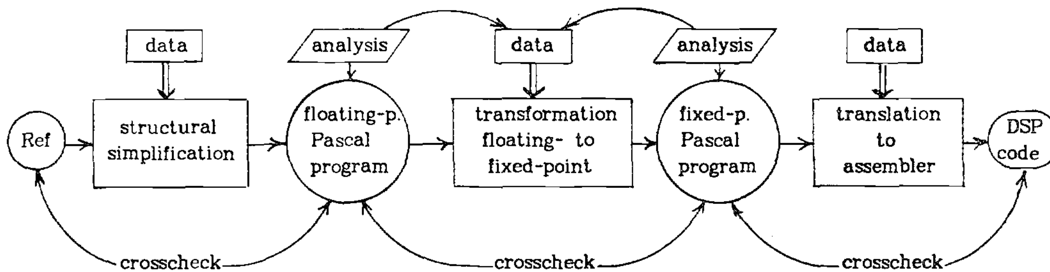

Figure 1.

Table 1. Construction of matrices **A**, **B**, **C**, **D** and **E**.

	$f(X_k)$	$g(X_k)$	$h(X_k)$	$l(X_k)$	$o(X_k)$
x_k $(k \leq n)$	$e_{k,n}^T$	$[0 .. 0]$	0	$[0 .. 0]$	-
$x_k=x_a x_b$	$f(X_a) + f(X_b)$	$e_{k,p}^T+g(X_a)+g(X_b)$	$h(X_a) + h(X_b)$	$l(X_a) + l(X_b)$	$[0 .. 0]$
$x_k=x_a+x_b$	$f(X_a)/2 + f(X_b)/2$	$e_{k,p}^T+g(X_a)/2+g(X_b)/2$	$1+h(X_a)/2+h(X_b)/2$	$e_{k,p}^T+l(X_a)/2+l(X_b)/2$	$e_{a,n+p}^T - e_{b,n+p}^T$
$x_k=x_a/x_b$	$f(X_a) - f(X_b)$	$e_{k,p}^T+g(X_a)-g(X_b)$	$nb(X_b)^*-1+h(X_a)-h(X_b)$	$l(X_a) - l(X_b)$	$[0 .. 0]$
matrices	$\mathbf{A} = \sum_{k=1}^{n+p} e_{k,n+p} f(X_k)$	$\mathbf{B} = \sum_{k=1}^{n+p} e_{k,n+p} g(X_k)$	$\mathbf{C} = \sum_{k=1}^{n+p} e_{k,n+p} h(X_k)$	$\mathbf{D} = \sum_{k=1}^{n+p} e_{k,n+p} l(X_k)$	$\mathbf{E} = \sum_{k=n+1}^{n+p} e_{k-n,p} o(X_k)$

where ^{T} stands for transpose, and $e_{k,n}$ for the kth column of the n-dimensional unity matrix
* assuming that wordlengths are already decided.