A FLOATING-POINT TO INTEGER C CONVERTER WITH SHIFT REDUCTION FOR FIXED-POINT DIGITAL SIGNAL PROCESSORS

Ki-Il Kum, Jiyang Kang and Wonyong Sung
School of Electrical Engineering, Seoul National University
Kwanak-gu, Seoul 151-742 KOREA

ABSTRACT

A floating-point to integer C program translator is developed for convenient programming and efficient use of fixed-point programmable digital signal processors (DSP's). It not only converts data types and supports automatic scaling, but also conducts shift optimization to enhance execution speed. Since the input and output of this translator are ANSI C compliant programs, it can be used for any fixed-point DSP that supports ANSI C compiler. A shift reduction method is developed for minimizing the scaling overhead of translated integer C programs. It considers the data-path of a target processor and profiling results. Using the shift reduction method, 4% to 37 % speedup is obtained. The translated integer C codes are 20 to 400 times faster than the floating-point versions when applied to TMS320C50, TMS320C60 and Motorola 56000 DSP's.

1. INTRODUCTION

Although the use of high-level languages for programmable digital signal processors is important in reducing the development time and retaining the portability, C compilers for fixed-point digital signal processors have met with little acceptance, especially because of the overhead of executing floating-point operations using a fixed-point data-path [1]. The development of fixed-point programs is considered tedious and difficult because it requires appropriate scaling for each data move and arithmetic operation to prevent overflows while maintaining accuracy [2][3]. The converter developed in this work, *Autoscaler for C*, can solve these problems because not only does it allow a programmer to avoid time-consuming assembly coding and manual scaling but also the translated C programs are executed very efficiently in fixed-point digital signal processors.

There are several recent research works for the automatic scaling and fixed-point implementation of general digital signal processing algorithms, such as the autoscaling assembler for TMS320C25 ('C25) [2][3], the fixed-point optimization utility for C and C++ based digital signal processing programs [4]-[6], the fixed-point C compilers for TMS320C50 ('C50) [7]. These tools still require assembly coding or source code modification because they do not support ANSI C language. However the developed *Autoscaler for C* accepts ANSI C based floating-point application programs, and generates ANSI C compliant programs including target specific codes. It also performs target dependent scaling shift minimization.

Figure 1 shows the design flow using the *Autoscaler for C*. First, the ranges of floating point variables are estimated by the simulation of the range estimation program that is automatically generated from the original floating-point version. The integer word-lengths, which are the number of bits used for the integer part, of the fixed-point variables are initially determined using the range estimation results. Second, the integer word-length of each variable is optimized to minimize the number of scaling shift operations using a data-path specific cost function. The simulated annealing algorithm is used for this shift-reduction. Finally, the floating-point variables and constants are replaced by the corresponding integer types, and appropriate scaling codes are inserted. The SUIF (Stanford University Intermediate Format) compiler system is used for source program parsing, analyzing, converting and generating the target programs [8].

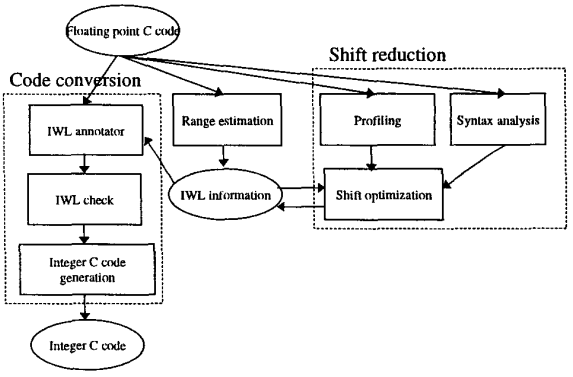

Figure 1. The floating-point to integer C converter design flow.

2. SIMULATION-BASED INTEGER WORD-LENGTH DETERMINATION

The fixed-point data format employed for this translator consists of sign, integer and fractional bits. The number of bits assigned to the integer is called the integer word-length (IWL), and that assigned to the fraction is the fractional word-length (FWL). Thus, the word-length (WL) corresponds to IWL+FWL+1. The range (R) and the quantization step (Q) are dependent on the IWL and FWL, respectively: $-2^{IWL} \le R < 2^{IWL}$ and $Q = 2^{-FWL} = 2^{-(WL-1-IWL)}$. Assigning a large IWL to a variable can prevent overflows, but it increases the quantization noise. Thus, the minimum IWL for a variable x, $IWL_{min}(x)$, can be determined according to its range, R(x), as follows.

$$IWL_{\min}(x) = \lceil \log_2 R(x) \rceil, \qquad (1)$$

0-7803-5041-3/99 $10.00 © 1999 IEEE

where $\lceil x \rceil$ denotes the smallest integer which is equal to or greater than x. A simulation based range estimation method was developed, where the range of each signal is measured during the floating-point simulation using realistic input signal files [4][5]. It employs the C++ class that traces the statistic information. It can be used for most algorithms described with C or C++ by simply replacing the float types with the range estimating C++ class. However, it cannot be applied to programs that have recursive functions, because the C++ class should be statically declared for collecting the statistical information, while the local variables in recursive functions are automatically declared and reside in stack frames [9]. Instead of the C++ class based method, a range estimation approach that uses a function call is employed for this study. It modifies the original C program by inserting a function call after every assignment statement. In this function, *range()*, the maximum absolute value, the sum, the squared sum and the number of assignments of a variable are traced. The function call insertion method is not only applicable to programs having recursive function calls, but also about 2.7 times faster than our previous C++ class based range estimation method as compared in Table 1. The execution time was measured using a Sun Ultra 1 workstation.

Table. 1. Comparison of execution time.

Program	original C code	C++ based range estimation	C based range estimation
IIR1 (1000000 samples)	0.22s	4.95s	1.88s
IIR4 (1000000 samples)	1.03s	25.99s	9.59s
QCELP (32640 samples)	8.58s	-	45.73s

The range estimation code is automatically generated as follows. First, it finds all of the floating-point variables by examining the symbol tables of the source program, and assigns a unique identification number to each floating-point variable. The identification numbers are attached to the corresponding variables in the symbol tables using the annotation function of the SUIF. Second, it traverses all of the expression trees and inserts the *range()* function call after every floating-point assignment. The range estimating program has a table that stores the maximum absolute value, the sum, the squared sum and the number of assignment of all the floating-point variables. In the function *range()*, the table elements indexed by the identification number are modified with the assigned value. The statistics are reported when program execution is completed.

3. CODE CONVERSION

Arithmetic and assignment operations for fixed-point variables or constants need scaling operations [2][3]. For example, a variable x having an IWL of 2 cannot be added directly to a variable y with an IWL of 1. The variable y should be shifted right by 1 bit before addition to align the binary-point. Note that the IWL of a variable is increased by the arithmetic right shift operation. If the IWL of the added result is greater than both IWL's of two input operands, the inputs should be scaled down to prevent overflows. Therefore, the scaling for addition is performed as shown in Table 2.

For fixed-point multiplication, it is important for preventing overflows to keep the upper part of the double precision multiplied result although integer multiplication in ANSI C only stores the lower part [7]. Since two's complement multiplication generates two sign bits, the IWL of the multiplied result becomes Ix + Iy +1, where Ix and Iy are the IWL's of two input operands x and y, respectively. In traditional C compilers, double precision multiplication followed by a double to single conversion is needed to obtain the upper part, which is obviously very inefficient [1]. However, in recent C compilers for some digital signal processors such as 'C50, the upper part of the multiplied result can be obtained by combining multiply and shift operations [10]. In the case of TMS320C60 ('C60), which has 32-bit registers and ALU's, but only 16 by 16-bit multipliers, the multiplication of the upper 16-bit of two 32-bit operands is efficiently supported by C intrinsics [11]. If there is no support for obtaining the upper part of the multiplied result in the C compiler level, an assembly level implementation of fixed-point multiplication is required. For Motorola 56000 processor, fixed-point multiplication can be implemented with a single instruction using inline assembly coding [12]. The implementation of the macro or inline function for fixed-point multiplication, *mulh()*, is dependent on the compiler of a target processor.

Table 2. Fixed-point arithmetic rules.

floating-point	fixed-point			result IWL
	Ix>Iy,Iz	Iy>Ix,Iz	Iz>Ix,Iy	
x = y	x = y>>(Ix-Iy)	x = y<<(Iy-Ix)	-	Ix
x+y	x+ (y>>(Ix-Iy))	(x>>(Iy-Ix))+y	(x>>(Iz-Ix))+ (y>>(Iz-Iy))	max(Ix, Iy,Iz)
x*y	mulh(x,y)			Ix+Iy+1 or Ix+Iy

z: variable that stores added result

The elements of an array variable are assumed to have the same IWL for simple code generation. For a pointer variable, the IWL is defined as that of the pointed variables. Since the IWL of a pointer variable is not changed at runtime, a pointer cannot support two variables having different IWL's. In this case, the IWL's of these pointers are equalized automatically at the shift optimization step.

4. SHIFT REDUCTION

Since a scaling is not needed for addition or assignment of operands having the same IWL, the number of scaling shifts can be reduced by equalizing the IWL's of relevant variables. Note that it is only allowed to increase the initial IWL's that are determined according to Eq. (1). Shift reduction requires global optimization since the IWL modification of a variable in an expression can incur more scaling shifts in other expressions. Shift optimization also depends on the architecture of the target DSP. If it has a barrel shifter, the number of shift bits does not affect the cycle time. However, if it has no barrel shifter and should conduct the scaling using one-bit shift operations, the shift overhead is also affected by the number of bits for a scaling operation. It is also needed for minimizing the execution time to reduce the scaling operations that are inside a long loop. Thus, this optimization requires program-profiling results.

The IWL modification that minimizes the overhead for scaling is conducted as follows. First, the number of shift bits for each expression is formulated with the IWL's of the relevant variables and constants. Second, the cost function that corresponds to the total overhead of scaling shifts is made based on the results of the first step, the target DSP architecture and the program-profiling information. Finally, the cost function is minimized by modifying the IWL's using the simulated annealing algorithms.

For a simple modeling, a floating-point expression is converted to several simple expressions having following form.

$$x_i = \sum_{j,k} x_j * x_k + \sum_l x_l \tag{2}$$

This expression will be converted to an integer expression with scaling shift insertion as follows.

$$x_i = \{\sum_{j,k}((x_j * x_k) >> s_{j,k}) + \sum_l (x_l >> s_l)\} << s_i \tag{3}$$

The shift amounts, $s_{j,k}$, s_l and s_i are determined as follows.

$$I_{rhs} = \max_{j,k,l}(I_{x_j} + I_{x_k} + 1, I_{x_l}, I_{x_i}) \tag{4}$$

$$s_{j,k} = I_{rhs} - (I_{x_j} + I_{x_k} + 1) \tag{5}$$

$$s_l = I_{rhs} - I_{x_l} \tag{6}$$

$$s_i = I_{rhs} - I_{xi}, \tag{7}$$

where I_x is the IWL of the variable x and I_{rhs} is the IWL of the right hand side expression.

For a DSP architecture without a barrel shifter such as Motorola 56000, total number of bits for shift operation is the overhead of scaling shifts. It is determined as:

$$c_t = \sum_i (d_i + \sum_j e_{i,j}) n_i \ , \tag{8}$$

where $e_{i,j}$ is the shift amount for the j-th term of the i-th expression, d_i is that for the assignment of the i-th expression and n_i is the number of execution counts of the i-th expression. The weight, n_i, is determined from the profiling results. For a DSP having barrel shifters, such as 'C25, 'C50 and 'C60, the number of scaling shift operations that are not zero is counted. The cost function is represented as follows.

$$c_t = \sum_i (f_B(d_i) + \sum_j f_B(e_{i,j})) n_i \ , \tag{9}$$

where $f_B(x)$ is zero when x is zero, and is one when otherwise. This means that no shift is needed when the number of shift bits is zero, and only one shift operation is needed when that is not zero.

The cost functions shown in Eq. (8) and (9) are minimized by modifying the IWL's with the following constraints. The first constraint is the IWL lower bound, which is determined by the range estimation. The second constraint is the IWL upper bound, which is required for avoiding a significant performance degradation. The third constraint is the IWL equality condition of pointer and array variables. As explained before, the variables sharing the same pointer should have the equal IWL. When a function has pointer or array variables in its parameters, the variables in the caller side and the callee side should have the same IWL also. The cost functions can be minimized using general optimization methods such as the simulated annealing algorithm [13].

Shift optimizer reads the IWL information file generated in the range estimation step, and writes back the optimized IWL information after minimizing the number of shifts. The implementation of the shift reduction program consists of three parts: source code profiling, syntax analysis and shift optimizing. The source code profiling collects the execution frequencies of floating-point expressions throughout the simulation of the profiling program that is automatically generated. The syntax analyzing part extracts the equations for the calculations of scale amounts and the IWL equality condition by analyzing the floating-point C program. The extracted information includes the simplified parse tree for floating-point expressions and the IWL equality constraints. The shift optimizer part generates a C program conducting the simulated annealing optimization with the syntax analysis results, the profiling results, and the initial IWL's.

5. IMPLEMENTATION EXAMPLES

5.1 Fourth order IIR filter

A part of the floating-point C code and the converted integer C code for this example are shown in Fig. 2.

```
x1 = 0.01* *x;
t1 = x1 + b1[0]*d1[0] + b1[1]*d1[1];
y1 = a1[0]*t1 + a1[1]*d1[0] + a1[2]*d1[1];
```

(a) The floating-point C code.

```
x1 = mulh(1374389534, *x)<<1;
t1=((x1>>5)+mulh(*b1,*d1)+mulh(b1[1],d1[1]))<<2;
y1=(mulh(*a1,t1)+mulh(a1[1],*d1)+
   mulh(a1[2],d1[1]))<<1;
```

(b) The integer C code before shift reduction.

```
x1=mulh(1374389534,*x);
t1=(x1+mulh(*b1,*d1)+mulh(b1[1],d1[1]))<<2;
y1=mulh(*a1,t1)+mulh(a1[1],*d1)+
   mulh(a1[2],d1[1]);
```

(c) The integer C code after shift reduction.

Figure 2. The C codes for the fourth order IIR filter.

Note that the floating-point constant of 0.01 is converted to 32 bit integer constant of 1374389534 with IWL of –6. The IWL's of the variable x, x1, t1, d1, a1, b1, and y1 are determined as 16, 9, 12, 12, 1, 1, and 13, respectively by the simulation based range estimation. The integer C code generated using these IWL's is shown in Fig. 2-(b). In this example, the speedup, which is the ratio in the execution time of the integer to the floating-point versions, was 29.8, 406, and 28.5 for 'C50, 'C60, and Motorola 56000, respectively, as shown in Table 3. The remarkable speedup of 'C60 is mainly due to the deeply pipelined VLIW architecture having a large register file and the efficient C compiler. This machine can execute up to 8 integer operations in one cycle and store all the variables of a small loop kernel in the registers, but needs a large number of no-operation cycles for floating-point function calls to flush the pipeline registers.

The developed shift reduction technique is applied to this example. The IWL's of the variables x, x1, and y are changed to 19, 14, and 14, respectively. Two scaling shifts are removed as shown in the Fig. 2-(c). The total number of shift operations in the whole C code is reduced from 7 to 2 with the IWL upper bound of 3 bits. The shift reduction results are shown in Table 4.

Table 3. Performance comparison for the fourth order IIR filter.

	# of cycles			SQNR
	floating-p.	integer	speedup	integer
'C50	2,980	100	29.8	49.3dB
'C60	3,659	9	406.6	57.9dB
56000	26,282	921	28.5	78.5dB

Table 4. The shift reduction results of the fourth order IIR filter.

		before shift reduction	after shift reduction
# of shifts in C codes		7	2
'C50	# of cycles	100	94
	speedup	-	6%
	SQNR	49.3dB	54.1dB
'C60	# of cycles	9	6
	speedup	-	33%
	SQNR	57.9dB	54.2dB
# of shifts in C codes		5	2
56000	# of cycles	921	577
	speedup	-	37%
	SQNR	78.5dB	78.5dB

5.2 QCELP Codec

The QCELP algorithm developed by Qualcomm [14] is implemented using TI's 'C60. This C program consists of 16 source and 4 header files having a total of 3648 lines. It has 381 floating-point variables including arrays and pointers. The system performance is measured by the SQNR of the input and the reconstructed speech signal samples. The simulation result shows 17.9 dB in the floating-point C version, and 17.36 dB in the fixed-point C version without shift reduction. The floating-point version requires about 27.1 million cycles for each 20 ms speech frame, while the converted integer version consumes only 1.1 million cycles, which shows that the integer version is 24.6 times faster than the floating-point version. According to the shifter reduction result, the shifter cost is reduced by 88%, and the program becomes 4.5% faster with only 0.1 dB performance degradation.

6. CONCLUDING REMARKS

The developed converter reads ANSI C programs without requiring any modification of the source codes, and generates ANSI C compliant scaled integer versions that are optimized for target processor architectures including TI's 'C50, 'C60 and Motorola's 56000 series. The converter consists of three parts, which are range estimation, shift reduction and code conversion modules. The SUIF compiler system is extensively used for the implementation of these modules. The simulated annealing method is used for the optimization of the number of shifts for scaling.

A fourth order IIR filter and the QCELP codec are implemented using the Autoscaler for C. The translated integer C versions are 20 to 400 times faster than floating-point C codes. For the fourth order IIR filter example, 71% of scaling shifts are removed and 6% to 37% of speedup is achieved according to the target processor architectures. The translator also reduces the quantization noise by keeping the upper part of the multiplied results and employing the simulation based optimum scaling method. For the fourth order IIR filter example, 49.3 to 78.5dB SQNR is obtained according to the word-length of a target processor. The total conversion time for the QCELP codec is less than 30 minutes because of the high level language based simulation for the range estimation and profiling.

7. REFERENCES

[1] V. Zivojnovic, "Compilers for Digital Signal Processors," *DSP & Multimedia Technology*, vol. 4, no. 5, pp. 27-45, July/August, 1995.

[2] W. Sung, "An Automatic Scaling Method for the Programming of Fixed-point Digital Signal Processors, " in *Proc. of the IEEE International Symposium on Circuits and Systems*, pp. 37-40, Singapore, June 1991.

[3] S. Kim and W. Sung, "A Floating-point to Fixed-point Assembly Program Translator for the TMS320C25," *IEEE Trans. on Circuits and Systems*, vol. 41, no. 11, pp. 730-739, Nov. 1994.

[4] S. Kim, and W. Sung, "Fixed-point Optimization Utility for C and C++ Based Digital Signal Processing Programs," in *Proc. of 1995 IEEE Workshop on VLSI signal processing*, pp. 197-206, Oct. 1995.

[5] S. Kim, K.-I. Kum, and W. Sung, "Fixed-point Optimization Utility for C and C++ Based Digital Signal Processing Programs," *IEEE Trans. on Circuits and Systems* accepted for publication.

[6] M. Willems, V. Buersgens, T. Groetker, and H. Meyr, "FRIDGE: An Interactive Code Generation Environment for HW/SW Codesign," in *Proc. of 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 287-290, Apr. 1997.

[7] Jiyang Kang and Wonyong Sung, "Fixed-point C Compiler for TMS320C50 Digital Signal Processors," in *Proceedings of the ICASSP '97*, pp. 707-710, Apr. 1997.

[8] *The SUIF Library*, Stanford Compiler Group, CA, 1994.

[9] Ki-Il Kum, Jiyang Kang and Wonyong Sung, "A Floating-point to Fixed-point C Converter for Fixed-point Digital Signal Processors," in *Proc. of the Second SUIF Compiler Workshop*, Aug. 1997.

[10] *TMS320C2x/C2xx/C5x Optimizing C Compiler (version 6.60)*, Texas Instruments Inc., TX, 1995.

[11] *TMS320C6x Optimizing C Compiler*, Texas Instruments Inc., TX, 1997.

[12] *DSP56KCC User's Manual*, Motorola Inc., 1992.

[13] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671-680, May, 1983.

[14] W. Gardner, P. Jacobs, and C. Lee, QCELP: A variable rate speech coder for CDMA digital cellular, Kluwer, MA, 1993.