Digital filter optimization for C language

Alexandru BÂRLEANU, Vadim BĂITOIU, Andrei STAN
Faculty of Automatic Control and Computer Engineering
Gheorghe Asachi University of Iasi, 700050, Romania
alexb@cs.tuiasi.ro, vadim.baitoiu@gmail.com, andreis@cs.tuiasi.ro

***Abstract*—A method for transforming C code with floating-point values into C code with integer variables is developed. The objective is to avoid any operations with floating-point data types, thereby increasing the execution speed of the program on a microprocessor without a math coprocessor. The original C code must be a dot product with floating-point literals and integer variables with known interval bounds. The transformation algorithm remodels the dot product form into a tree structure, to maximize the accuracy, but, on the other side, keeps the number of shift operations reduced. The integer code that is generated is ANSI C compliant. It is tested on 8-bit and 32-bit microprocessors using different compilers. The results show that the integer code is several times faster than the floating-point code, the only loss being a very low accuracy drop.**

***Index Terms*— embedded software, fixed-point arithmetic, filtering algorithms, design optimization, accuracy.**

I. INTRODUCTION

Many embedded computers are used to execute control algorithms and process digital signals. Because the processing power is often small and the timing requirements are severe, such operations must be carefully designed.

Special attention needs the C code containing floating-point variables. It is easy to write expressions with floating-point data types (because the C language is a high level language), but it is hard to foresee what results after compilation if no floating-point hardware is available.

On the other hand, fine optimization of embedded software can be itself time-consuming. But this is generally by far compensated, because low-cost/low-power devices do have an important market share and increasing the software quality can only be welcome.

A. Task description

The task can be formulated as follows: given a dot product in C language:

$$\sum_{i=0}^{n} a_i \cdot x_i \qquad (1)$$

where a_i are constant coefficients of type *float*, and x_i are bounded signed integers, a function must be written, in ANSI C, so that the value of the dot product is computed in minimum time and with maximum accuracy.

Original expression must be a dot product. A large number of formulas can be expressed as dot products. All FIR/IIR filters with Direct-Form I structure can be written as dot products.

Values x_i are bounded signed integers. Digital filters are used to process signals obtained usually from AD converters, which have a typical resolution of 8-12 bits. So it is possible to indicate that a certain filter input may vary between known boundaries.

Values x_i can be negative. This increases the complexity of the task, but is necessary for control algorithms, where values x_i can represent some errors.

Final code must be ANSI C compliant. This condition poses some difficulties (e.g. right shift of negative values), but offers portability, which is an important aspect. The generated code can be used with any C compiler and on any microprocessor.

The benefits of code transformation are multiple. For some applications the processor load can be decreased, for other the digital filter length can be increased to achieve better results.

The only risk associated with the generated code is determined by the precondition violation. If at least one filter input is out of the supposed bounds then the end result is very likely to be erroneous (no saturation on overflow).

B. Drawbacks of floating-point code

Compilers have built-in libraries for floating-point arithmetic. The functions used to handle operations with floating-point data types are highly optimized, but, anyway, the execution time is considerable. This is essentially determined by the fact that library functions return values with standard accuracy (IEEE-754 compliance).

On the other side, specific applications can tolerate specific degrees of accuracy loss. But the tolerable limits are unknown for the compilers. Hence standard accuracy is used.

C. Related work

Considerable work has already been done on automatic conversion of floating-point code to fixed-point [1]-[13]. The developed methods can be classified in two basic groups: statistical (search-based) and analytical.

The drawback of search-based methods is that the *trial and error* process can take a long time, and, even so, the optimum point can be local. This can be a blocker for multi-dimensional search spaces (a great number of fixed-point variables). The key of a successful search algorithm is the path selection.

Analytical methods require mathematical relations. It *matters* how fixed-point variables are connected (the data flow, in other words). Modifying the fractional length of an arbitrary variable has a predefined impact on the state of other variables. This is why analytical methods can yield solutions very promptly.

Some methods are focused on producing ANSI C code, other methods on assembly code or FPGA. The language (or platform) determines the usable fixed-point formats.

Digital Object Identifier 10.4316/AECE.2011.03018

II. CODE TRANSFORMATION

The original dot product expression is transformed into a sequence of statements containing only integer data types (fixed-point values). First, the expression is parsed and an abstract model is built. Then the model is optimized and the corresponding C code is generated.

A. Intermediate representation

At abstract level the digital filter is represented as a data flow graph. A node can represent an input or an operator. The implementation of nodes follows the class hierarchy shown in Fig. 1. Categories of nodes:

1. **Parameter nodes**. Their output values can be directly set to constant values or between specific limits.
2. **Operator nodes**. Their output values depend on some operands (child nodes).

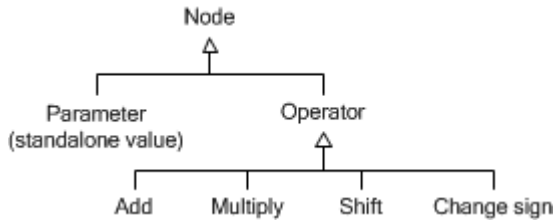

Figure 1. Node class hierarchy

A node stores information about a floating-point value. The value itself is essentially unknown. The low and high limits are relevant. The radix point is declared explicitly at design time, but is implicit at run-time (in the generated code). The position of the radix point is computed for each node in accordance with the fixed-point arithmetic rules [14]. The interval of a node of type operator is computed using the intervals of operands [15].

Each operator node has an associated *storage class* flag. This is used at code generation to indicate whether the node variable should be declared. It is desirable to have as many as possible variables allocated in registers, so it should be possible to generate code with as few as possible declared intermediary variables.

Four types of operator are used: *add*, *multiply*, *shift* and *change sign*.

B. Overflow avoidance

Nodes of type *add* and *multiply* are subject to overflow. The interval limits of a node of type operator are determined by the type of the operator and by the intervals of operands. To avoid an operator to overflow, it is necessary to decrease the length of its fractional part. But this cannot be done directly. The interval of an operator is not *writable*.

The only way to do the necessary adjustment of the fractional part is to manipulate the fractional parts of the operands (child nodes). For a node of type *add* it is necessary to do this for each child node (term). For a node of type *multiply* it is sufficiently in most cases to change the fractional part of a single node.

Interval reduction process is generally triggered from top to down. A node can be forced to reduce its interval because its interval does not fit in any of the available data types (it does overflow) or, it does not overflow, but a top node needs an operand with a smaller interval.

It is particularly simple to reduce the intervals for *shift* operators and for nodes that represent constant values. For *shift* operators the shift distances are changed and for constant values some least significant bits are discarded. On the other hand, for operators like *add* and *multiply* the process of reducing the intervals can be very complex. It's because the data flow transformation variants are usually multiple, and each variant has its own impact on the code complexity and accuracy.

Multiplication nodes imply usually loss of accuracy. This is because, in most cases, the intervals of factors must be reduced. Let M be a node of type *multiply* with two factors: F1 and F2. The only condition that must be met in order to obtain a valid value in node M is to avoid the multiplication overflow. (It doesn't matter the radix point position in F1 and F2.) To avoid overflow in node M, the intervals of F1 or F2, must be reduced. Which one to change, or how much to change each of the two, matters, because the accuracy is involved. Maximum accuracy is attained only if the intervals of F1 and F2 are as close as possible (effective bit lengths of F1 and F2 are equal). In practice, the most common case is when a node of type *multiply* has the following two factors:

1. A very precise constant (coefficient a_i).
2. An integer which can vary within relatively close limits (variable x_i).

In this case, the problem of overflow can be resolved by only tuning the fractional part of the constant node (which has no run-time cost and is the solution with the minimum accuracy loss).

C. Alignment of summation terms

Before performing a summation it is necessary to ensure that all terms have equal fractional part lengths. Aligning the radix points can be done, in general, in multiple ways, but the supplementary run-time operations must be considered. Let A be a node of type *add* with two terms: T1 and T2. Let the length of the fractional part of T1 be shorter than for T2. Possible alignment methods:

1. Increase the fractional part of T1. This method is rarely applicable, because it usually causes the node A to overflow. But, it yields very high accuracy.
2. Decrease the fractional part of T2. Some least significant bits of T2 are lost.

Method 1 is tried first; if it cannot be applied then method 2 is applied.

D. Split of summations with three or more terms

The intermediate representation can contain operators of type *add* with any number of terms. Since each term is subject to alignment before performing a summation, the association of terms determines the accuracy of the result. If the alignment of a term means arithmetic shift of bits to right then this implies loss of significant bits and must be done as *late* as possible. Fig. 2 illustrates the optimal grouping of a sum with three terms that have different fractional lengths. The basic idea is to sum first the values with the longest fractional length.

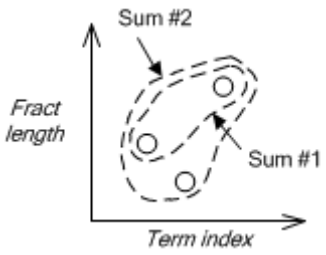

Figure 2. Optimal grouping of summation terms

Given an arbitrary operator of type *add*, optimal grouping

of terms can be accomplished in two steps:

1. Sort the terms by the length of the fractional part, in descending order.
2. Let n be the length of the list of terms. Create n-1 groups – in group 0 include terms 0 and 1, in group i ($i \neq 0$) include group i-1 and term i+1.

E. Dividing integers by two

Dividing integers by two is problematic for negative numbers (in two's complement) [16]. Some processors do not have hardware support for division, and bitwise shift to right would be the only acceptable alternative. But the standard for C language [17] specifies that the result of a bitwise shift to right, E1>>E2, where E1 is a negative value, is implementation-defined. The compiler decides whether to propagate the sign or not. So, if the compiler is unknown then it is uncertain if the generated C code with bitwise right shifts on possibly negative values will produce the correct results. But, even if the right shift of a negative value would produce also a negative value, this would not be an accurate division by two. The division of a negative and a nonnegative value by two with a right shift is asymmetric. To attain symmetry, before computing E1>>E2, if E1 is negative then it must be incremented.

The only way to produce functional ANSI C code without targeting a specific compiler is to completely avoid bitwise right shifts of values that can be negative. To do so, the following transformations can be applied to E1>>E2:

1) Reversing the sign of E1 if E1 is negative

Principle: if E1 is negative then it is reverse its sign, execute the shift and then reverse the sign of the result. The run-time disadvantage is the cost of comparison, sign changes, and branching that leads to non-constant execution time.

2) Biasing E1

Principle: add a specific value to E1 so that it becomes nonnegative; execute the shift operation, then substract a specific value from the shifted result. This is possible because the bitwise right shift E1>>E2 can be written in the following form (2):

$$\frac{E1}{2^{E2}} = \frac{E1}{2^{E2}} + \frac{x}{2^{E2}} - \frac{x}{2^{E2}} = \frac{E1 + x}{2^{E2}} - y \tag{2}$$

Symbols x and y are the above mentioned offsets. Value $E1+x$ must be nonnegative and 2^{E2} must divide x.

Offset x should be as low as possible. Equations (3) and (4) can be used to compute x and y.

$$y = \left\lceil \frac{-\ lowest\ \ value\ \ of\ \ E1}{2^{E2}} \right\rceil \tag{3}$$

$$x = 2^{E2} \cdot y \tag{4}$$

The main disadvantage with the usage of offsets is that two add operations must be performed at run time. But there are important advantages: the execution time is constant (no jitter is introduced) and the result can be rounded at no cost. Simply adding 2^{E2-1} to y (after computing x) causes the result to be rounded.

However, it is possible to write, instead of E1>>E2, directly $E1/2^{E2}$ and to let the compiler to optimize (i.e. to substitute the division with the appropriate right shift, in case the division is not implemented in hardware). But this optimization is compiler dependent. It is worth to consider, but does not represent a general solution.

F. Sign handling

The paths of the data flow graph can be classified in the following categories: paths that carry only positive values, paths that carry values with any sign and paths that carry only negative values.

The number of paths that carry negative values matters, because it is desirable to have as few as possible right shifts of values that can be negative. Such shifts are costly, as described in section ***Dividing integers by two***.

Some paths that carry only negative values can be transformed into paths carrying only positive values. This is because strictly negative intervals are produced usually by negative constants, which can be written as positive constants followed by *sign change* operators. *Sign change* operators can be moved up (to the root) to precede as many as possible other types of operators, which means that some nodes with strictly negative intervals get strictly positive intervals.

III. RESULTS

The code transformation method is tested with dot products containing 4-10 terms. For each dot product term $a_i * x_i$ the following conditions are set:

1. $a_i \in (-1.0; 1.0)$
2. x_i low value = 0 or low value $\in [-5201; -201]$, x_i high value $\in [200; 5200]$

TABLE I. FUNCTIONS COMPARED BY SPEED AND ACCURACY

Function	Description
round(dot product with floating-point data types)	The result is the closest integer value the real value of the dot product. The error interval is (-0.5; 0.5).
int(dot product with floating-point data types)	The result is obtained by casting the real value of the dot product to an integer. The error interval is (-1.0; 1.0).
automatically generated code with integers	The error interval is very close to (-0.5; 0.5).

For illustrative purposes, twelve random dot products are tested for speed and accuracy (with 4, 6, 8, and 10 terms). The generated ANSI C code uses only signed and unsigned 16-bit and 32-bit integer data types.

A. Accuracy

The accuracy is estimated on a high-speed computer. The error of the generated code is computed with random test cases (uniform distribution). For each function in Table I is computed the signal-to-quantization-noise ratio (*SQNR*). The accuracy test program stops automatically when the SQNR estimators settle.

As a rule, for each test case, the SQNR of *int*(floating-point code) is 3dB less than the SQNR of *round*(floating-point code) and the SQNR of the generated code is 0.003dB less than the SQNR of *round*(floating-point code). In other words, the accuracy of generated code is nearly ideal. The output of the generated code is always the same as the output of the function *round*(floating-point code), except for a limited number of input combinations.

B. Speed

The execution speed is estimated on 8-bit and 32-bit microcontrollers. The floating-point operations are done in simple precision (with 32-bit floating-point data types). The generated code that is tested is in the form of a single arithmetic statement. No intermediary variables are used, to

eliminate the load/store overhead.

TABLE II. MICROCONTROLLERS USED FOR SPEED TESTS

MCC	Architecture	Frequency	Compiler
ATmega16	AVR	4MHz	IAR 4.12
LM3S3748	ARM Cortex-M3	50MHz	IAR 5.41
STM32F		56MHz	gcc
LPC1768		50MHz	IAR 5.4

The generated code executes in significantly less time than the original floating-point code. The speed factor is, for all test cases, above 6 (Table III and Fig. 3).

TABLE III. AVERAGE SPEED FACTOR, WITH COMPILER OPTIMIZATIONS DISABLED

MCC / Compiler	Dot product length (number of terms)			
	4	6	8	10
ATmega16 / IAR	14,97	14,99	14,60	14,66
STM32F / gcc	11,44	10,96	11,25	11,59
LPC1768 / IAR	9,51	10,01	10,37	10,86
LM3S3748 / IAR	7,15	7,42	7,55	7,76

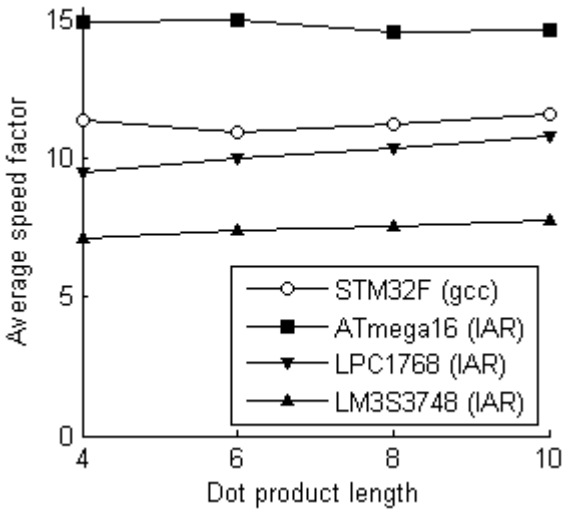

Figure. 3. Average speed factor, with compiler optimizations disabled

The execution time of the integer code grows linearly with the filter length, but is also determined by the intervals of the data flow paths. Digital filters with positive coefficients and unsigned input variables are the most advantaged, because no negative value must be shifted to right.

The execution time of the integer code is constant. This feature might be useful in applications with deterministic behavior. (The floating-point code has a specific jitter.)

IV. CONCLUSIONS

A floating-point to fixed-point convertor for ANSI C code is described. The optimization algorithm is designed to rewrite floating-point dot products with constant coefficients and bounded integer variables.

Special consideration is given to the code generation. The compiler defined behavior of the C code (such as when a signed integer is shifted right) cannot cause incorrect results. The generated code can be used with any C compiler.

The fractional word-lengths of the intermediary variables are made as long as possible, the only barrier being the integer overflow. The data flow structure is not, in every case, symmetrical. This is in contrast with other methods that produce code with a lot of variables with the same radix point position. This is the reason why the accuracy of the generated integer code is very high. It is as if the original floating-point code is used and the floating-point result is rounded to the nearest integer; except for very few input combinations (<0.5%), when the error is ±1.

The speed of the generated integer code is much better than the speed of the original floating-point code. The execution time is reduced by 6-14 times and is made constant. Right shift operations of negative values take a significant amount of time, but the proposed way of doing them offers cross-compiler portability.

REFERENCES

[1] K. I. Kum, J. Kang, W. Sung, "AUTOSCALER For C: An Optimizing Floating-Point to Integer C Program Converter For Fixed-Point Digital Signal Processors", IEEE Trans. on Circuits and Systems II: Analog and Digital Signal Processing, vol. 47, issue 9, pp. 840-848, Sep. 2000.

[2] D. Menard, D. Chillet, F. Charot, O. Sentieys, "Automatic Floating-point to Fixed-point Conversion for DSP Code Generation", in Proc. of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, Oct. 2002.

[3] D. Menard, R. Serizel, R. Rocher, and O. Sentieys, "Accuracy Constraint Determination in Fixed-Point System Design," EURASIP Journal on Embedded Systems, 2008, article ID 242584.

[4] D. Menard, R. Rocher, O. Sentieys, "Analytical Fixed-Point Accuracy Evaluation in Linear Time-Invariant Systems", in IEEE Trans. On Circuits and Systems I, vol. 55, issue 10, pp. 3197-308, 2008.

[5] C. Shi, R. W. Brodersen. "An Automated Floating-point to Fixed-point Conversion Methodology," in Proc. of IEEE International Conf. on Acoustics, Speech, and Signal Processing, vol. II, pp. 529-32, 2003.

[6] C. Shi, R. W. Brodersen, "Floating-point to fixed-point conversion with decision errors due to quantization," in Proc. of IEEE International Conf. on Acoustics, Speech, and Signal Processing, vol. 5, pp. 41-4, 2004.

[7] C. Shi and R. W. Brodersen, "Automated fixed-point data-type optimization tool for signal processing and communication systems," in Proc. of the Design Automation Conf., pp. 478–483, USA, 2004.

[8] A. Cilio, H. Corporaal, "Floating Point to Fixed Point Conversion of C Code," in Proc. of the 8th International Conf. on Compiler Construction, ETAPS'99, vol. 1575, pp. 229-243, 1999.

[9] K. Han, B.L. Evans, "Optimum wordlength search using sensitivity information," EURASIP J. on Applied Signal Processing, article ID 92849, pp. 1-14, 2006.

[10] K. Han, I. Eo, K. Kim, H. Cho, "Numerical word-length optimization for CDMA demodulator", in IEEE International Symposium on Circuits and Systems, vol. 4, pp. 290-293, 2001.

[11] P. Belanovic, M. Rupp, "Automated floating-point to fixed-point conversion with the fixify environment," The 16th IEEE International Workshop on Rapid System Prototyping, pp. 172-178, 2005.

[12] N. Sulaiman, "A Multi-objective Genetic Algorithm for On-chip Real-time Optimisation of Word Length and Power Consumption in a Pipelined FFT Processor targeting a MC-CDMA Receiver," in NASA/DoD Conf. on Evolvable Hardware, pp. 154-159, 2005.

[13] M. Leban, J. F. Tasic, "Word-length optimization of LMS adaptive FIR filters," in Proc. of the 10th Mediterranean Electrotechnical Conf., pp. 774–777, 2000.

[14] Fast Floating-Point Arithmetic Emulation on Blackfin® Processors, EE-185, Analog Devices, Inc, 2007.

[15] R. B. Kearfott, "Interval Computations: Introduction, Uses, and Resources," Euromath Bulletin, vol. 2, no. 1, pp. 95–112, 1996.

[16] R. J. Mitchell and P.R. Minchinton, "A Note on Dividing Integers by Two," The Computer Journal, 32, No. 4, Aug 1989, 380.

[17] Programming languages — C, International Standard, ISO/IEC 9899:TC2