Fixed-point arithmetic for ASIP code generation

Andrea G. M. Cilio   Ireneusz Karkowski   Henk Corporaal

Department of Electrical Engineering,
Delft University of Technology,
P.O. Box 5031, 2628 CD Delft, The Netherlands

A.Cilio@et.tudelft.nl   I.Karkowski@et.tudelft.nl   heco@cs.tudelft.nl

Keywords: Embedded system design, Retargetable code generation, SUIF

Abstract

In this paper we present an environment supporting fixed-point arithmetic in C programs. It allows the specification of the position of the binary point in the source code and code generation for architectures with hardware support for the fixed-point arithmetic. Our tools are implemented on the top of the SUIF C compiler[1] and do not require any special adaptations. We demonstrate the validity of our approach on a series of experiments on a test program. The results show that fixed-point arithmetic executed on an integer datapath has a limited impact on the accuracy when compared to a floating-point version of the same program. In the same time the performance difference is remarkable: the fixed-point code is 3.7 to 10 times faster than its equivalent floating-point emulation on an integer datapath.

1 Introduction

In order to meet the increasingly tight time-to-market constraints, code generation for complex embedded systems is shifting towards high level languages and code compilation. The C language, although not ideal for embedded applications, is a popular imperative specification language as it combines the capabilities of a typical HLL with low-level assembly language features like bit manipulation. Furthermore, C has become the *de facto* standard specification language of several international standards: for example MPEG (IEC 13838)[2] and ADPCM (G.722)[3].

One of the limitations of C is that it does not support fixed-point integer types. In the embedded systems context, the motivations for fixed-point arithmetic are the speed of the algorithm and the cost and complexity of the hardware. The overhead reported for a floating-point software emulation ranges between a factor of 10 and 500 [4]. The trade-offs between an algorithm implementation using floating-point software emulation and a fast, albeit less accurate, fixed-point algorithm often favor the latter solution.

In manual fixed-point programming the designer replaces the floating point variables with fixed-point ones, encoded as integers. To avoid overflows and reduce the loss of precision he must scale the integer words. Determining the number of shifts is known to be error prone and time consuming. Automatic conversion from floating-point to fixed-point is an attractive alternative, addressing these problems.

This paper presents a design environment that allows the specification of a fixed point representation of selected floating-point variables. A tool called *float2fix* automatically performs floating- to fixed-point conversion of a given program. The converter works in combination with the code generator, which is able to translate the generated intermediate representation (IR) into a target instruction set that does not directly support fixed-point arithmetic (by inserting the appropriate scaling instructions). Alternatively, when the target machine has some form of support for fixed-point arithmetic, the code generator recognizes the instruction patterns corresponding to fixed-point operations and maps them into the specialized instructions.

The remainder of this paper is organized as follows. Section 2 presents a survey of research on automatic fixed-point code generation from C. Section 3 reviews the basic concepts of fixed-point arithmetic and introduces our approach to the specification of the fixed-point type of a variable in C source code. Section 4 presents the code transformations performed by *float2fix*. In section 5 the code transformations are demonstrated by means of an example program. Finally section 6 concludes this paper and lists its contributions.

2 Related work

In the last years the field of automatic conversion of C programs from floating-point to fixed-point has gained much attention. This is due to the appearance of digital signal processors offering hardware support for fixed-point arithmetic (like Texas Instruments' TMS320C50). Usually, the design of a fixed-point algorithm starts with floating-point code which is then converted to fixed-point in a semi-automatic fashion. Two alternatives have been considered regarding instantiation of the fixed-point format of a floating-point variable:

- *Instantiation at definition time* involves a unique instantiation at the location where the variable is defined.
- *Instantiation at assignment time* requires that the optimal format be determined every time the variable is assigned.

According to Willems et al.[5], the implementation of algorithms for specific target machines requires a bit-true specification at source level. This specification can then be transferred into a HDL or a programming language. In [6] he proposes a non-ANSI extension to C in which two fixed-point parameterizable types are introduced. The first allows the user to instantiate the fixed-point format of a variable at definition time. With the second, more flexible type the converter determines the best format at assignment time. This approach leads to very accurate fixed-point conversion. The behavior of the operators on the new types is fully specified, including overflow handling and rounding mode[1] . The user is free to specify the fixed-point format for some variables and let the converter determine the format for the remaining ones. This is achieved by propagating the known formats along the data-flow graph and by profiling the floating-point variables left unspecified by the user. Statistics of these variables are collected by means of a hybrid (floating-point and fixed-point) simulator and are used to estimate the optimal number of bits for the integer and the fractional parts. The process is interactive: the converter can ask the user to supply additional data when the information available is not sufficient. Once the fixed-point format of all the variables has been determined, the conversion tool can generate new ANSI-C code based only on integers types, with the appropriate scaling and masking operations. An open question is how well can compiler optimizations reduce the overhead due to integer bit-true transformations when generating code for a specific target. Also, it is questionable whether a bit-true control of the algorithm at the source level is "cost-effective". The target machine in fact dictates what are the most efficient overflow and rounding modes. Software emulation of a different behavior is inefficient, hardly acceptable for implementations in which execution speed is at a prize.

One disadvantage of the instantiation at assignment time used by Willems is that it requires two specific simulators: a hybrid and a bit-true simulator. The former, as mentioned above, is needed for profiling, the latter to simulate the accuracy of the application on the target processor. Another complication comes from pointers. The converter must estimate the result of loads and the values that might have been stored into a location. The fixed-point format of all the possible values must then be combined.

In [7][8] Sung, Kum et al. propose an automated, simulation based fixed-point transformation methodology. Also in this case the user can specify the fixed-point format of selected variables. The range, mean and variance of the remaining variables are measured by profiling. A range estimator[9] uses the collected statistical data to determine the optimal fixed-point format. The conversion uses the concept of *definition time instantiation* only. The authors focus on specific target architectures (TSM320C5x) with dedicated hardware for fixed-point arithmetics. Moreover, their approach requires custom adaptations to existing compilers. Their results show that fixed-point programs using 16-bit integer words are 5 to 20 times faster than the software floating point simulation. The speedup drops to 2 times when 32-bit fixed-point words are used.

The simulation based format determination, used in both mentioned approaches, has two disadvantages. One of them is that it depends on the profiling input data. A more serious problem is that it is slow, as the optimal fixed-point format is determined running several simulations for every single variable. The estimator is typically implemented by means of C++ classes, which introduce a severe overhead compared to a base type [10].

Our approach differs in several aspects from the above described ones. While choosing *definition time instantiation*, we do not support automatic, simulation-based determination of the fixed-point format. Furthermore, we generate machine code for a wider target space[11]. The machine code and the number of scaling instructions are optimized for the specific target, which can have any level of support for fixed-point arithmetic.

3 Specification and representation of the fixed-point format

In this section we review some basic concepts of fixed-point binary representation and we address the issue of how to specify the fixed-point format in the C source.

[1] In ANSI-C these aspects are dependent on the implementation.

3.1 Fixed-point representation

A fixed-point quantity can be thought of as an integer multiplied with a two's power with negative exponent. In other words, the weight 1 is assigned to a bit other than LSB of the word, and the bits to the right of that bit represent the fractional part of the value. The fixed-point format is thus determined by at least[2] two parameters: the total wordlength WL and the integer word length, IWL (the sign bit is not included). Obviously, the fractional word length is: $WL - IWL - 1$. Figure 1 shows a fixed-point 16-bit word having $IWL = 4$.

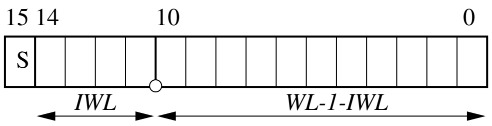

Figure 1: A two's complement fixed-point word.

The range of values R and the quantization step Q of a fixed-point word are determined by WL and IWL:

$$-2^{IWL} \leq R \leq 2^{IWL} - 1$$

$$Q = 2^{-(WL-1-IWL)}$$

In the following subsections we will focus on a single value of WL and different IWL values. For a discussion of how to implement code generation for user-definable integer wordlengths see [12].

3.2 Fixed-point arithmetic rules

Two fixed-point data can be added by a normal integer unit, provided the position of their binary points is the same. If the source operands have different IWL, the word with the smaller IWL must be scaled so as to align the binary point positions:

$$IWL = \max\{IWL_1, IWL_2\} \tag{1}$$

The result must be scaled to the IWL of the destination operand.

The two's complement integer multiplication of two words yields a result of wordlength $2WL$. The IWL of the result is given by:

$$IWL = IWL_1 + IWL_2 + 1 \tag{2}$$

In integer multiplication the least significant half of the result is accessible, whereas the upper part is often discarded and possible overflows are simply truncated. In a fixed-point multiplication this is not acceptable, since the integer wordlength of the result occupies the most significant part of the word. Therefore, we need to retain the upper half of the result. There are three solutions:

1. Make the upper part of the result accessible.
2. Scale the source operands before multiplying so that the result will fit in the word.
3. Implement the fixed-point multiplication as a macro that computes the upper part of the result.

The first solution, used in [7], is convenient when the target architecture upper word is accessible. However, in the mentioned work this approach requires custom adaptations to the compiler. The second approach shows severe precision degradation: the precision is actually reduced to half every time a multiplication is executed. The last approach retains the precision of the first at the price of cycle count increase. Our converter supports any of the three solutions; the latter two have been implemented in *float2fix* and compared (see section 5 for the experimental results).

3.3 Fixed-point specification

Our approach to fixed-point specification avoids adaptations to the compiler front-end and retains compilability with any ANSI-C compiler. The proposed solution uses the SUIF (Stanford University Intermediate Format) compiler [1] and takes advantage of its flexible intermediate representation. In SUIF new unique types can be created by adding annotations to existing types. This enables us to extend the IR with a fixed-point type system without any change to the compiler.

Our methodology uses *definition time instantiation* of the fixed-point format. This limitation reduces the complexity of the converter at the expenses of accuracy. The user specifies the fixed-point format of `float` and `double` variables using annotations. Annotations are specified by means of a custom `#pragma` directive, as shown in the following example. This directive is ignored by normal compilers. This means that the same source file can be used for both floating-point and fixed-point compilation.

Example 1 Specification of a fixed-point variable

```
float signal_out[100], *out, *in_ptr;
#pragma suif_annote "fix" signal_out 5
#pragma suif_annote "fix" out 5
```

The base type of both the array `signal_out` *and the pointer* `out` *is a fixed-point integer with* $IWL = 5$. *The other pointer is left undeclared by the user. The converter will try to determine its format from the data-flow.*

□□

The user is expected to annotate all variables for which the format cannot be determined by the converter by means of static analysis of the program's data-flow graph (DFG). Floating point variables initialized with a constant need not be annotated by the

[2] Other parameters, like signedness, casting mode and overflow handling can be taken into account.

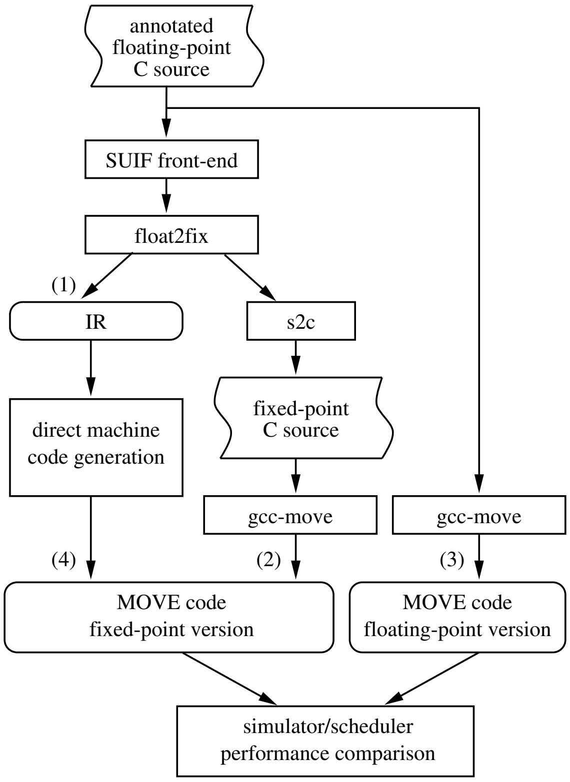

Figure 2: Code generation trajectory.

user, because *float2fix* can determine their fixed-point format from the constant value. For all the intermediate values, like compiler defined temporaries, the fixed-point format can be determined from the format of the other operands, at assignment time. We define this as a limited form of 'interpolation', similarly as in [5].

4 Fixed-point conversion and code generation

In this section we present a general view of our code generation environment, with special attention for the aspects specifically related to floating-point to fixed-point code conversion. This is followed by a more detailed description of our conversion tool, *float2fix*, and an example that illustrates the code transformations involved.

4.1 The code generation environment

Figure 2 shows the passes necessary to generate fixed-point code for a MOVE processor from a C source.

The designer starts with the manual annotation of `float` and `double` variables, as explained in section 3. This annotated source file is translated into an IR by the SUIF front-end. A new compiler pass, *float2fix*, is run immediately after the front-end. It

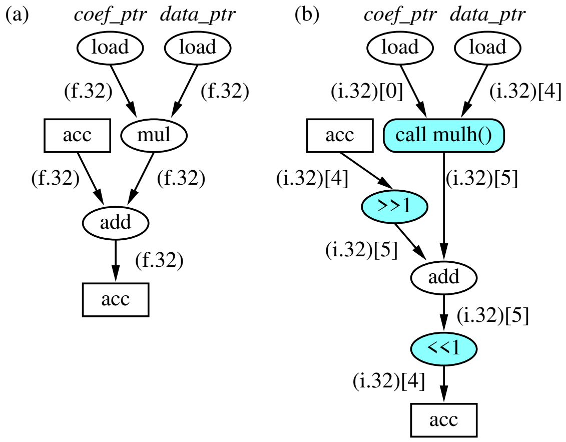

Figure 3: Example of code transformation on a SUIF instruction tree, representing: `acc += (*coef_prt)*(*data_ptr).` (a) Original tree. (b) Transformed tree.

reads the annotated IR and translates it to an fixed-point (integer encoded) IR (1) that can be converted back to a C integer source. This source can be compiled by *gcc-move* into fixed-point MOVE code (2). The annotated source file can also be directly compiled by *gcc-move* into floating-point code (3) (either hardware-supported of software emulated). This allows to run simulations and perform comparisons between the two versions of the algorithm. The user can evaluate the performance and the accuracy of the fixed-point code and adjust the fixed-point format of the variables. An alternative path to code generation (4), not yet fully implemented, will use the SUIF based back-end and will be able to recognize fixed-point instruction patterns (like shift-operation-shift) and map them into dedicated instructions. Note that only in this case the back-end needs to be modified.

4.2 Fixed-point conversion

Float2fix is implemented as an IR-to-IR compiler pass on the SUIF representation. This pass transforms the IR of a floating-point program annotated by the user into a fixed-point equivalent in the following steps:

1. For each distinct fixed-point format specified by an annotation, it generates and installs the corresponding type and updates the type of the annotated variable.
2. It converts every floating-point constant into fixed-point integer representation and installs the corresponding type. If the constant is not linked to an annotated variable definition, its fixed-point format is determined by:

$$IWL = \max\{\lceil \log_2 |constant| \rceil, 0\}$$

For initialized arrays, $|constant|$ is replaced with $\max_i \{|constant_i|\}$ where $constant_i$ is the initializer for element i of the array.

3. It propagates the fixed-point format along the instruction trees. The objects that may need type conversion are: variables that the user did not annotate, compiler generated temporary variables, or single definition–single use values (edges of the instruction tree). The IWL of these objects is determined using formulae (1) and (2).
4. When the source and the destination operands of a fixed-point instruction do not match, or when the source operands need binary point alignment, it inserts the appropriate scaling instructions to convert between different fixed-point formats.

Figure 3 illustrates the last two transformations on statement taken from the test program used in section 5: `acc += (*coef_ptr)*(*data_ptr)`. This expression is translated into a DFG in which the nodes represent instructions and the boxes represent variables. Constant addresses are shown in italics. Edges represent definition-uses of data. Next to every edge is the type of the transferred datum; the first letter indicates whether the type is floating-point (f) or integer (i), the following number indicates the wordlength, WL. Enclosed in square brackets is the value of IWL. Note that a standard 32 bit `int` type would be described with (i.32)[31].

The fixed-point multiply is translated to a macro invocation that computes the upper part of the integer multiplication using the formula[3]:

$$\mathrm{mulh}(a, b) = a_h * b_h + (a_h * b_l + a_l * b_h) >> 16$$

Where a_h, b_h are the upper parts, and a_l, b_l are the lower parts of a and b, given by:

$$a_h = a >> 16; \quad a_l = a \;\&\; 0\text{xFFFF}$$

Note that this macro introduces an approximation due to the fact that the product between the lower parts is disregarded.

5 Experimental results

In this section we present experimental results of our fixed-point code generation trajectory and we compare them with two possible floating-point implementations. The chosen test application is a fir filter of 35th order. It has been implemented with `float` variables and then converted to fixed-point C, compiled, scheduled and simulated.

Table 1 shows the results for four versions, corresponding to rows one to four, respectively:

1. **fp-hw** Floating-point implementation using floating-point unit.

[3]This formula is valid for $WL = 32$.

Ver.	Cycles	Moves	Instr.	SQNR
fp-hw	11859	74521	76	–
fp-sw	142582	581332	152	70.9 dB
fix-s	14431	75194	54	33.2 dB
fix-m	24151	147846	58	73.7 dB

Table 1: Results for a 35th order fir filter.

2. **fp-sw** Floating-point implementation with integer instructions (software emulation).
3. **fix-s** Fixed-point implementation with fast multiply (the operands are scaled down by 16 bits).
4. **fix-m** Fixed-point implementation with mulh() macro call substitution for fixed-point multiply.

The cycle counts, in the second column, have been obtained by scheduling the code on a 8-bus processor with 2 load-store units, 2 immediate units, 2 integer units an a floating-point unit (FPU). In this architecture the FPU supports double-precision only. The `float` source operands are extended when loaded from memory. The third column shows the number of moves. This relates to a peculiar characteristic of our target architecture: data transports, or moves, are explicitly programmed[13]. The fundamental unit of 'control' of the processor is the data transport, as opposed to the machine instruction. The fourth column shows the code static size. The last column shows the Signal to Quantization Noise Ratio (SQNR) of the last three implementations in comparison to the first one. From these results we can draw the following conclusions:

1. The speedup factor of fix-s and fix-m implementations relative to fp-sw is large: 9.9 and 5.9 respectively. Also resource utilization contributes to this result: in fix-m, the machine was busy 77% of the cycles.
2. The SQNR of fix-s implementation is rather poor, whereas fix-m shows a ratio of 74dB, which is acceptable in most applications.
3. Note that the fp-sw implementation introduces some error compared to fp-hw. This is due to the fact that the software really emulates `float` values, whereas the FPU uses double precision.
4. Remarkably, the SQNR of fix-m is higher that of fp-sw. This is due to the fact that, for ranges of values that in the floating-point representation have a small exponent, the fixed-point representation can devolve up to 31 bits for the fractional part, whereas in a `float` (IEEE 754) 8 bits are taken by the exponent and only 23 are left to the significand.
5. The overhead of the mulh() macro call is 67% respect to fix-s and 100% respect to fp-hw. These values indicate that the compiler and the scheduler performed very well in optimizing the code

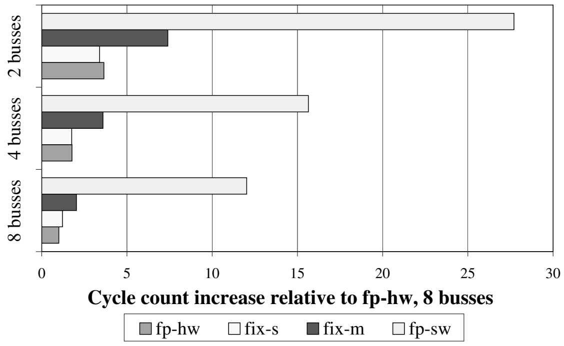

Figure 4: Scalability of the four program versions.

and reducing the impact of the macro computations. In particular, a feature related to the explicit programming of moves, namely *software bypassing*,[4] reduced the register pressure to a level very close to that of fix-s.

The high level of parallelism allowed by the target configuration used in the tests is somewhat biased towards fixed-point code, which shows a higher amount of inherent parallelism. To verify this, we run a number of tests with smaller target configurations, to see how much impact do restricted resources have on the overhead of the fixed-point implementations (see figure 4). Reducing the number of busses and integer units by half increased the cycle count by 76% and 44% in fix-m and fix-s, respectively, whereas fp-sw resulted only 30% slower. This suggests that fp-sw does not scale with the machine resources as effectively as a fixed-point implementation. One of the reasons is that floating-point operations are implemented by function calls, which reduce the scope of the scheduler. On the other hand, fixed-point conversion introduces operations in the expression trees without affecting the scheduling scope. As a result, fix-m is still 4.4 times faster than fp-sw. Even on a minimal configuration with two busses, fix-m is 3.7 times faster.

Notice that in all these tests the processor configuration is the same: the integer versions do not use the FPU. This choice is biased towards the fp-hw version, since the expensive resources utilized by the FPU, or part of them, could be invested in more busses and integer units.

6 Conclusions

Data type conversion of a floating-point specification to a fixed-point specification has been implemented and tested on a fir filter example. We devised two alternatives for fixed-point multiplication. The results show that the loss of accuracy due to the fixed-point representation is highly dependent on the implementation of the multiplication. With the most accurate alternative, we obtain a Signal to Quantization Noise Ratio of 74dB with respect to a double-precision, hardware supported implementation. The comparison with a floating-point implementation on an integer datapath showed that, depending on the level of parallelism sustainable by the target machine, a speedup factor from 3.7 to 5.9 is achieved with the more accurate fixed-point version, and a speedup from 8.2 to 9.9 with the less accurate one.

We feel that the approach presented in this paper is promising and the results encourage us to continue in the direction of code generation for targets with direct support for fixed-point, like shifters at the functional unit inputs and access to the upper part of the integer multiplication result.

[4] Software bypassing is a technique whereby a transport from the result register of a functional unit to the functional unit that uses it is explicitly programmed, bypassing the write and read of a register.

References

[1] Stanford Compiler Group. *The SUIF Library*. Stanford University, 1994.

[2] Joint Technical Committee ISO/IEC JTC1/SC29/WG11. ISO/IEC 13818 "Information technology – Generic coding of moving pictures and associated audio. International standard, ISO/IEC, June 1995.

[3] International Telegraph and Telephone Consultative Committee. General Aspects of Digital Transmission Systems. Terminal Equipments Recommendations G.700–G.795. International standard, CCITT, Melbourne, November 1988.

[4] Loughborough Sound Images. Evaluation of the performance of the c6201 processor & compiler, 1997. Internal paper: http://www.lsi-dsp.co.uk/c6x/tech.

[5] Markus Willems, Volker Bürsgens, Holger Keding, Thorsten Grötker, and Heinrich Meyr. System Level Fixed-point Design Based on an Interpolative Approach. In *Design Automation Conference*, 1997.

[6] Thorsten Grötker Markus Willems, Volker Bürsgens and Heinrich Meyr. FRIDGE: An Interactive Fixed-point Code Generator Environment for HW/SW CoDesign. In *proceedings of the IEEE International conference on Acoustic, Speech and Signal Processing*, pages 687–690, München, April 1997.

[7] Wonyong Sung and Jiyang Kang. Fixed-Point C Language for Digital Signal Processing. In *Twenty-Ninth Annual Asilomar Conference on Signals, Systems and Computers*, October 1995.

[8] Wonyong Sung Jiyang Kang. Fixed-point C compiler for TMS320C50 digital signal processor. In *proceedings of ICASSP'97*, 1997.

[9] Ki-Il Kum Seehyun Kim and Wonyong Sung. Fixed-point Optimization Utility for C and C++ Based Digital Signal processing Programs. In *proceedings of IEEE Workshop on VLSI Signal Processing*, October 1995.

[10] Stanley B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, 1996.

[11] Jan Hoogerbrugge. *Code Generation for Transport Triggered Architectures*. PhD thesis, Technical University of Delft, February 1996.

[12] Andrea G. M. Cilio. Efficient Code Generation for ASIPs with Different Word Sizes. In *proceedings of the third conference of the Advanced School for Computing and Imaging*, June 1997.

[13] Henk Corporaal. *Microprocessor Architectures; from VLIW to TTA*. John Wiley, 1997. ISBN 0-471-97157-X.