Floating Point to Fixed Point Conversion of C Code

Andrea G.M. Cilio and Henk Corporaal

Delft University of Technology
Computer Architecture and Digital Techniques Dept.
Mekelweg 4, 2628CD Delft, The Netherlands
{A.Cilio,H.Corporaal}@its.tudelft.nl

Abstract. In processors that do not support floating-point instructions, using fixed-point arithmetic instead of floating-point emulation trades off computation accuracy for execution speed. This trade-off is often profitable. In many cases, like embedded systems, low-cost and speed bounds make it the only acceptable option. We present an environment supporting fixed-point code generation from C programs. It allows the user to specify the position of the binary point in the source code and let the converter automatically transform floating-point variables and operations. We demonstrate the validity of our approach on a series of experiments. The results show that, compared to floating-point, fixed-point arithmetic executed on an integer datapath has a limited impact on the accuracy. In the same time the fixed-point code is 3 to 8 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)[9] and ADPCM (G.722)[7].

One of the limitations of C is that it does not support fixed-point integer types. For embedded systems in which tight cost constraints do not allow the use of floating-point hardware, using a fixed-point version of the algorithm to implement is an attractive alternative to floating-point software emulation, for which the reported overhead ranges between a factor of 10 and 500 [6]. Often, the trade-offs between an algorithm implementation using floating-point software emulation and a fast, albeit less accurate, fixed-point algorithm 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 supports semi-automatic conversion of floating-point code into fixed-point. The user is allowed to specify the fixed-point representation of selected, critical floating-point variables; a tool called *float2fix* automatically performs floating- to fixed-point conversion of the remaining floating-point variables and inserts the appropriate scaling operations. The code generator can then map the converted intermediate representation (IR) into a target instruction set that supports only integer arithmetic.

The rest of this paper is organized as follows. Section 2 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 3 describes the code transformations performed by *float2fix*. In Sec.4 the code transformations are tested on a number of benchmarks. Section 5 presents a survey of research on automatic fixed-point code generation from C. Finally, Sec.6 concludes this paper and summarizes its contributions.

2 Representation and Specification of Fixed-Point Numbers

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

2.1 Fixed-Point Representation

A fixed-point number can be thought of as an integer multiplied by a two's power with negative exponent. In other words, the weight 1 is assigned to a bit other than the LSB of the word, and the bits to the right of that bit represent the fractional part of the value. We can associate a fixed-point type to this representation. The minimal set of parameters that determine a fixed-point type are the signedness, the total length of the word WL and the length of its integer part, IWL. The fractional word length of a 2's complement number n is thus[1] $FWL = WL - IWL - 1$ and the value represented by its bits, $a_{WL-1}, \dots, a_0$ is:

$$a = \left(-a_{WL-1} 2^{WL-1} + \sum_{i=0}^{WL-2} a_i 2^i \right) \cdot 2^{FWL} \tag{1}$$

Figure 1 shows the signed fixed-point representation of a number. The values of WL and IWL determine two important properties of the fixed-point representation: the range of representable numbers R and the quantization step Q:

$$R = [-2^{IWL}, 2^{IWL}); \qquad Q = 2^{-(WL-1-IWL)}$$

[1] Note that we do not consider the sign bit a part of IWL.

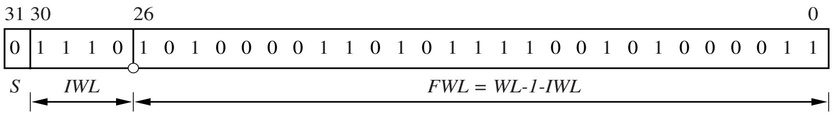

Fig. 1. Signed fixed-point representation ($WL = 32, IWL = 4$) of the floating-point number 14.631578947368421. The accuracy of the fractional part is equivalent to 13 decimal digits.

Two more parameters must be specified to describe at source level the bit-true behavior of a fixed-point type: *casting mode* and *overflow mode*. The casting mode specifies what happens to a fixed-point number when it is shifted to right. The least significant bits can be ignored *(truncate mode)* or used to round off the number. The overflow mode specifies how to handle the result of a fixed-point operation that overflows. The most significant bits can be discarded *(wrap-around mode)* or the result can be replaced with the maximum value that can be represented with that fixed-point type *(saturation mode)*.

In our opinion, a bit-true specification at the source level, which must include overflow and rounding mode, has little use; the target machine dictates what are the most efficient type parameters and thus the behavior of fixed-point operations. In practice, if a particular behavior is not supported in hardware, it must be emulated in software, and this is highly inefficient. Emulation is thus hardly acceptable in applications where the focus is on performance. On the other hand, if the designer can explore the target architecture solution space, he might want to change the specifics of casting mode or overflow mode of the target machine without having to change the source code in order to adapt it to the new fixed-point parameters. For these reasons, we decided to let casting and overflow be target-dependent aspects and define a fixed-point type only by its *WL* and *IWL*.

In the following discussion, we will consider only signed fixed-point numbers and a unique value of *WL*. In [1] we discuss how to implement code generation for types with user-definable *WL*. We consider supporting unsigned fixed-point types a straightforward extension that adds little to the approach presented in the following sections.

2.2 Fixed-Point Arithmetic Rules

The following rules specify the format of the result of a fixed-point arithmetic operation $c = a \star b$. The *IWL* of the result is derived from the fixed-point representation (1).

Addition and comparison. Two fixed-point numbers can be added or compared 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:

$$IWL_c = \max\{IWL_a, IWL_b\} \tag{2}$$

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

$$IWL_c = IWL_a + IWL_b + 1 \tag{3}$$

Notice that the second most significant bit of the result is normally just a duplicate of the sign bit and $IWL_a + IWL_b$ bits are sufficient to represent the integer part of the result.[2] In many processors, integer multiplication returns only the lower WL bits. The upper part is discarded and overflows can be signaled. The reason for this is that in high-level languages multiplication maps two source operands of integer type into a destination of the same type, therefore compilers do not generate code that exploits the precision of a full result. In a fixed-point multiplication this behavior is not acceptable, because the upper WL bits of the result contain the integer part of the number. There are three alternatives to obtain the upper half of the result:

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

The first alternative, used in [13], is convenient when the target architecture upper word is also accessible. However, this approach requires custom adaptations to the compiler. The second approach, called *integer multiplication*, shows poor accuracy. The last approach can be very accurate at the price of additional computation. Our converter supports the latter two alternatives, but leaves the possibility to efficiently map multiplications to target processors in which the upper part of the result is accessible.

Division. Two fixed-point numbers can be divided using a normal integer divider. From (1) follows that the IWL of the result is:

$$IWL_c = WL - 1 + IWL_a - IWL_b = \tag{4}$$

The division is the trickiest of the fixed-point arithmetic operations. Without careful scaling of the source operands, the chances to loose accuracy are very high. Notice that if the IWL of the denominator IWL_b is small, then the accuracy of the result is poor. If $IWL_a > IWL_b$, the result cannot even be represented with a WL-bit word. In this case, we must clearly insert scaling operations to reduce the resulting IWL. In Subsec.3.3 we present some strategies to limit the loss of accuracy in fixed-point divisions.

[2] It is easy to verify that the two most significant bits of the result are not equal only if both source operands are -2^{WL}.

2.3 Fixed-Point Specification

Our fixed-point specification does not require special adaptations to the compiler front-end and is *transparent* to the compiler. The user specifies the value for *IWL* of `float` and `double` variables by means of annotations introduced by a reserved `#pragma` directive. This directive is ignored by a compiler that does not recognize it, thus the same source file can be used for both floating-point and fixed-point compilation. The user can also specify the *IWL* of arguments of external functions, as shown in the example below.

Example 1 (Specification of fixed-point variables and functions).

```
double sin(double);
float signal_out[100], *in_ptr;
#pragma suif_annote "fix" signal_out 5
#pragma suif_annote "fix" sin 8 1
```

The base type of the array `signal_out` is a fixed-point integer with $IWL = 5$. The fixed-point type of the pointer `in_ptr` will be determined by the converter using data-flow information. The function `sin()` takes a fixed-point argument with $IWL = 8$ and returns a fixed-point result with $IWL = 1$. □

The user is expected to annotate all floating-point variables for which the fixed-point format cannot be determined by the converter using the static analysis of the program described in Sec.3. Note that floating-point variables initialized with a constant need not be annotated by the 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.

3 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. Then we describe in more detail our conversion tool, *float2fix*, and we demonstrate the code transformations involved through an example.

3.1 The Code Generation Environment

The conversion to fixed point uses the SUIF (Stanford University Intermediate Format) compiler [4] and takes advantage of its flexible intermediate representation. In SUIF new unique types can be defined 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. Figure 2 shows the passes necessary to generate fixed-point code for our target architecture, called *MOVE*, from a C source. The designer starts with the manual annotation of `float` and `double` variables, as explained in Sec.2. This annotated source file is translated into SUIF IR by the front-end. Our converter, *float2fix*, is run immediately after the front-end. It reads the annotated IR and translates it to a fixed-point (integer encoded) IR (see (a) in Fig.2) that can be converted back to a C integer source. This source can be compiled by *gcc-move* into fixed-point MOVE code (b). The annotated source file can also be directly compiled by *gcc-move* into floating-point code (c) (either hardware-supported or 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 (d), 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.

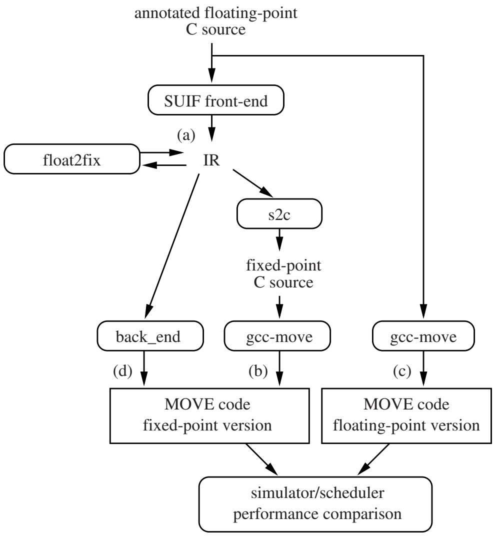
annotated floating-point C source
SUIF front-end
(a)
float2fix
IR
s2c
fixed-point C source
back_end
gcc-move
gcc-move
(d)
(b)
(c)
MOVE code fixed-point version
MOVE code floating-point version
simulator/scheduler performance comparison

Fig. 2. Code generation trajectory.

3.2 Fixed-Point Conversion

Float2fix is implemented as an IR-to-IR transformation pass that translates the SUIF representation of a floating-point program annotated by the user into a fixed-point equivalent in the following steps:

1. It generates and installs the fixed-point types specified by the user's annotations. The type of the annotated variable is updated to the new type.
2. It converts floating-point constants to fixed-point format and installs the corresponding types. If the constant is not linked to an annotated variable

definition, its fixed-point format is determined by:

$$IWL = \max\{\lceil \log_2 |constant| \rceil, 0\}^{3}$$

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

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, and single definition-use values (edges of the instruction tree). The IWL of these objects is determined by applying rules (2,3, 4) and the techniques to be presented in Subsec.3.3.
4. It inserts the appropriate scaling instructions to align the binary point of source operands or to convert from one fixed-point type to another.

Figure 3 illustrates the last two transformations on a statement taken from one of the test programs of Sec.4: `acc += (*coef_ptr)*(*data_ptr)`. This expression is shown as a graph in which the nodes represent instructions and the boxes represent variables. 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 32-bit `int` type would be described with (i.32)[31]. The fixed-point multiply is translated to the macro invocation mulh(a, b), that computes the upper part of the integer multiplication using one of the following formulae[4]

$$a_h * b_h + ((a_h * b_l + a_l * b_h) >> 16) \tag{5}$$

$$a_h * b_h + (((a_h * b_l + a_l * b_h) + (a_l * b_l >> 16)) >> 16) \tag{6}$$

where

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

Note that (5) introduces some approximation due to the fact that the product between the lower parts is disregarded; in (6) the first shift is unsigned.

3.3 Precision-Improving Techniques

Hereby we present a number of heuristics that improve the precision attainable with fixed-point arithmetic operations.

Multiplication. As stated in Sec.2, one bit of the result's integer part is a copy of the sign bit and conveys no additional information. We can therefore scale down the result by one position. This gives us one additional bit for the fractional part. Moreover, a chain of multiplications automatically converted to fixed-point can

[3] Notice that we could allow *negative* values of IWL, i.e. numbers whose binary point falls out of the word. An analogoue extension is possible to represent numbers for which $IWL \geq WL$.

[4] These formulae are valid for $WL = 32$.

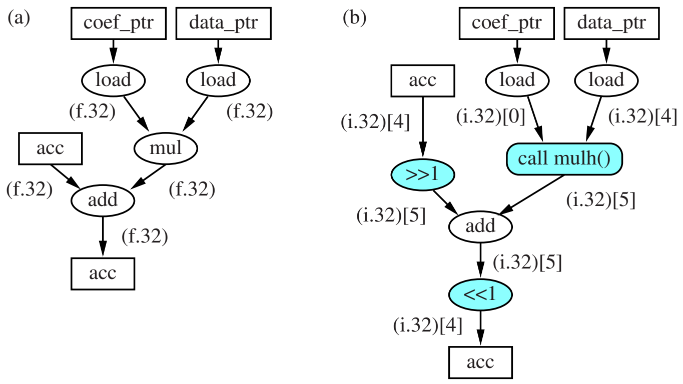

Fig. 3. Example of code transformation on a SUIF instruction tree, representing: `acc += (*coef_prt)*(*data_ptr)`. (a) Original tree. (b) Transformed tree.

produce a result with unnecessarily high IWL and therefore little accuracy. By scaling the result we alleviate this problem.

Another important improvement is possible for macro (6) when the destination of the multiply is a fixed-point variable d, and the IWL of the result, as computed by (3), is higher than IWL_d. In this case, we can modify the macro so that it computes exactly the bits that are to be found in d. Given $D = IWL_a + IWL_b + 1 - IWL_d$, the modified macro $\text{mulh}(a, b, D)$ is

$$((a_h * b_h) << D) + (((a_h * b_l + a_l * b_h) + ((a_l * b_l) >> 16)) >> (16 - D)) \quad (7)$$

Division. Equation (4) summarizes the difficulty of fixed-point division: a denominator with small IWL_b with respect to IWL_a yields very poor accuracy. The solution is to scale the denominator and increase IWL_b before performing the division. This is necessary when $IWL_b > IWL_a$, as the division would otherwise produce an overflow. When the denominator is a constant value B, it is possible to shift out the least significant bits that are '0' without introducing any loss of accuracy. We can extend this idea by computing Err_{denom}, the error caused by shifting out the n least significant bits of the denominator:

$$Err_{denom} = B_{n-1:0}/(B - B_{n-1:0})$$

where $B_{n-1:0}$ is the unsigned integer value represented by the n least significant bits. We can then compare this error with the maximum quantization error of the result, Err_Q, and estimate an amount of shifting that minimizes both Err_{denom} and the potential quantization error. This approach can be further generalized into a heuristic that can be applied to variable denominators. We compute the amount of shifting using the following expression

$$\lfloor (FWL_b) \cdot \alpha \rfloor \quad (8)$$

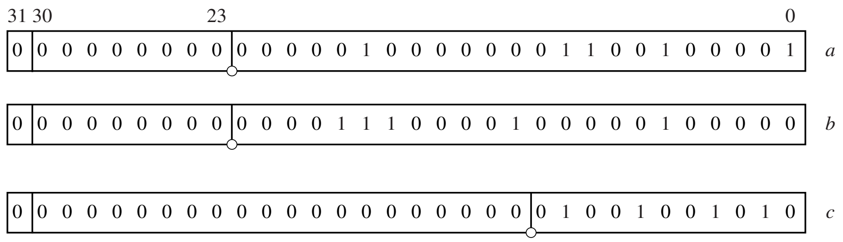
31 30 23 0
0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 1 a
0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 1 1 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 b
0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 1 0 0 1 0 0 1 0 1 0 c

Fig. 4. Example of fixed-point division between $a = 0.015720487$ and $b = 0.054935455$; the fixed-point format of the result $c = 0.28616285$ is the most accurate one among the many that are possible for a/b.

α is a parameter representing the aggressiveness of scaling; it is the fraction of bits of the fractional part that have to be shifted out. The above expression takes into account both Err_{denom} and Err_Q. Although this technique is risky, in that it may introduce spurious divisions by zero, it turned out to work very well in practice, as shown in Sec.4. Example 2 demonstrates the idea

Example 2 (Scaling the denominator to improve division accuracy). Consider the operation $c = a/b$ where a, b are the fixed-point numbers whose value and IWL are shown in Fig.4. An integer division delivers zero as quotient, and thus a 100% error. The same result is obtained if b is shifted one position right. If we scale b by 2 to 5 positions, we are shifting out zeroes and we cannot loose accuracy. Shifting out the first 4 bits gives invariably 0.25, which corresponds to an error of 12%; shifting by 5 position drastically improves the accuracy: 0.28125 (1.72% error). Shifting out the first '1', in the sixth position, does not affect the result. Shifting out all the subsequent '0' bits steadily improves the accuracy: by shifting 8 positions the error becomes 0.35%, by shifting 11 positions 0.01%. By shifting out 12 bits the error increases to 0.42%. From this point on, shifting more bits makes the accuracy smaller, because the new least significant bits computed are erroneous. Only when we start to shift out the three most significant '1' bits Err_{denom} increasingly offsets the reduction of quantization error, and the overall error rises up to 76%. If also the last non-zero bit of b is shifted out we have a division by zero. The macro that we implemented will replace it with the largest number that can be represented with WL bits. □

4 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 applications are FIR, a 35th-order fir filter and IIR, a 6th-order iir filter [3]. Both have been coded using `float` variables and then converted to fixed-point C, compiled, scheduled and simulated. We tested four versions of the programs:

	FIR Filter				IIR Filter			
Version	**Cycles**	**Moves**	**Instr.**	**SQNR**	**Cycles**	**Moves**	**Instr.**	**SQNR**
fp-hw	32826	86862	66	–	7422	22367	202	–
fp-sw	151849	542200	170	70.9 dB	39192	107410	258	64.9 dB
fix-s	23440	102426	58	33.1 dB	5218	27861	61	20.3 dB
fix-m	39410	175888	68	74.7 dB	8723	51899	81	55.1 dB

Table 1. Performance and accuracy results for the test applications.

1. **fp-hw** Floating-point implementation using floating-point unit.
2. **fp-sw** Floating-point implementation using software emulation.
3. **fix-s** Fixed-point implementation using integer multiplication (see Sec.2).
4. **fix-m** Fixed-point implementation using invocation to multiply macro (5).

Table 1 shows the accuracy and performance of these four versions. Each row shows the results for the version whose name is in the first column. The cycle counts in columns 2 and 6 were obtained by scheduling the code on a 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. Columns 3 and 7 show the number of moves. This relates to a peculiar characteristic of our target architecture: data transports, or moves, are explicitly programmed [2]; we roughly need 2 moves per basic (RISC-like) operation. Our target machine has 8-move busses and therefore can execute around 4 instructions per cycle. The fundamental unit of 'control' of the processor is the data transport, as opposed to the machine instruction. Columns 4 and 8 show the code static size. As accuracy metric we chose the Signal to Quantization Noise Ratio (SQNR), defined as follows:

$$\mathrm{SQNR} = 10\log_{10}\left(\frac{S}{N}\right)$$

where S is the average of the signal's absolute value and N is the average of the error, defined as the difference between the original signal and the quantized signal. Column 5 and 9 show the SQNR of the last three implementations in comparison to fp-hw. From these results we can draw the following conclusions:

1. For both programs, the speedup factor of fixed-point implementations relative to fp-sw is large, above 6 for fix-s, above 3 for fix-m. Good resource utilization contributes to this result: in fix-m, for example, the machine buses were busy 55% of the execution time in FIR and 74% in IIR.
2. The SQNR of fix-s implementations is poor, whereas fix-m FIR shows a ratio of 74dB, which is acceptable in most applications. For IIR the accuracy of the results is not completely satisfactory.
3. The SQNR ratio of fp-sw implementation shows that it 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, in FIR the SQNR of fix-m is higher than that of fp-sw. This is due to the fact that, for floating-point numbers that have a small exponent, the fixed-point representation can use up to 31 bits for the fractional part, whereas in a `float` (IEEE 754) only 24 bits are used to represent the significand.
5. The execution overhead due to macro (5) is 68% respect to fix-s. This indicates that the compiler and the scheduler were effective at optimizing the code and reducing the impact of the macro computations. In particular, a feature related to the explicit programming of moves, namely *software bypassing*,[5] reduced the register pressure to a level very close to that of fix-s.
6. In IIR the scheduler did not find enough parallelism to keep the 8 busses busy in the fp-hw implementation. As a result, fix-m is only slightly slower, while fix-s outperforms fp-hw. These cases suggest the use of more accurate macros, like (6) and (7).

We tested some of the precision improvements presented in Subsec.3.3. By scaling down the result of (5) by one bit we obtained a 9% cycle count reduction and at the same time measured an improvement of accuracy of 1.1dB. More tests remain to be done using macros (6) and (7).

Scalability of converted fixed-point code 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 on FIR, to see how much impact do restricted resources have on the overhead of the fixed-point implementations (see Fig.5). 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 the tests of all versions were performed on the same processor configuration. Since 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.

Accuracy of division. Although the test programs did not contain fixed-point divisions, we also measured the accuracy attainable with the heuristic (8). We performed a large number of fixed-point divisions on random values and collected statistical data. Usually floating-point operands of real programs are not

[5] 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 general purpose register.

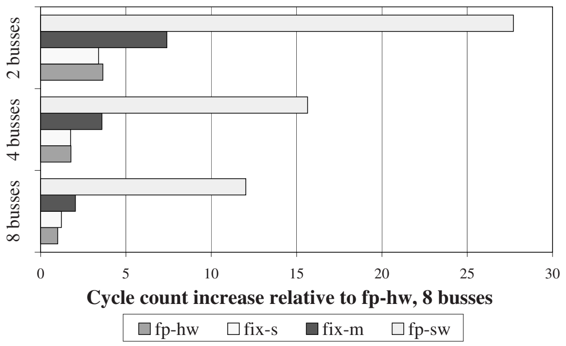

Fig. 5. Scalability of the four program versions.

randomly distributed over the entire range; to account in part for this, we added a parameter that determines the ratio between range of the random distribution for the numerator and the denominator. Figure 6 shows the results when the largest number is smaller than 2.0 ($IWL = 1$ bit). On the X axis is the value of the heuristic's parameter, α. On the Y axes is the error introduced by the fixed-point conversion, expressed in dB. As one can see, the precision steadily increases for all versions up to $\alpha = 0.4$. For high values of α, the error due to coarse quantization of the denominator offsets the accuracy gained due to a smaller IWL for the result. The effect of the heuristic is less pronounced when IWL is larger. As a limit case, the heuristic gracefully degrades to integer division when the range of both operands is $WL - 1$, and we obtain flat curves with integer precision, which entirely depend on the ratio.

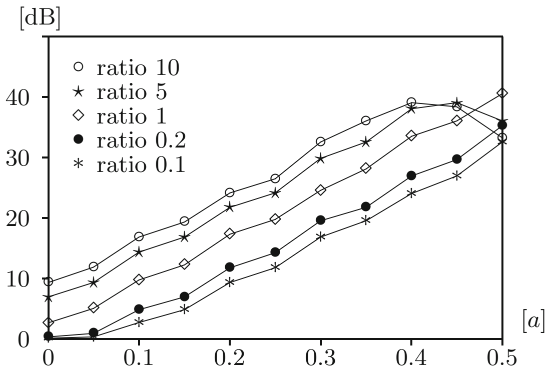

Fig. 6. Effect of heuristic (8) on division's accuracy.

5 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 the popular Texas Instruments' TMS320C50). Usually, the design of a fixed-point algorithm starts with floating-point code which is then converted to fixed-point, manually or in a semi-automatic fashion. Two alternatives have been considered regarding instantiation of the fixed-point format of a floating-point variable:

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

According to Willems et al.[14], 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 [11] he proposes a non-ANSI extension to C in which two fixed-point parameterized 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[6] . 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 result 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 in implementations for which execution speed is critical.

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

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

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 [13][8] Sung, Kum et al. propose an automated, fixed-point transformation methodology based on profiling. 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 [12] 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 a factor 2 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. The choice of *definition time instantiation*, substantially simplifies the algorithm. Also, it contributes to more efficient code, as the number of shift operations is likely to be smaller. Although we do not support profiling to determine the fixed-point format, the results showed that static analysis and the described heuristics can deliver the same accuracy. Finally, and differently from the other approaches, we generate machine code for a wide target space [5].

6 Conclusions

Data type conversion of a floating-point specification to a fixed-point specification has been implemented and tested on two digital filter algorithms. We devised several 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 alternatives, we obtain a Signal to Quantization Noise Ratio of 74dB and 55dB with respect to a double-precision, hardware supported implementation. For one test program, the comparison with a floating-point implementation on an integer datapath (software emulation) 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, compared to floating-point software emulation.

The accuracy and the execution speed attained in the experiments show that the approach presented in this paper is promising. The results encourage us to continue in the direction of fine-tuning the heuristics and generating code for specialized 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.

References

1. 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. 231
2. Henk Corporaal. *Microprocessor Architectures; from VLIW to TTA*. John Wiley, 1997. ISBN 0-471-97157-X. 238
3. Paul M. Embree. *C Algorithms for Real-Time DSP*. Prentice Hall, 1995. 237
4. Stanford Compiler Group. *The SUIF Library*. Stanford University, 1994. 233
5. Jan Hoogerbrugge. *Code Generation for Transport Triggered Architectures*. PhD thesis, Technical University of Delft, February 1996. 242
6. Loughborough Sound Images. Evaluation of the performance of the c6201 processor & compiler, 1997. Internal paper: http://www.lsi-dsp.co.uk/c6x/tech. 229
7. 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. 229
8. Wonyong Sung Jiyang Kang. Fixed-point C compiler for TMS320C50 digital signal processor. In *proceedings of ICASSP'97*, 1997. 242
9. 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. 229
10. Stanley B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, 1996. 242
11. 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. 241
12. 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. 242
13. 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. 232, 242
14. 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. 241