Fixed-Point Code Synthesis Based on Constraint Generation

Sofiane Bessaï, Dorra Ben Khalifa, Hanane Benmaghnia, Matthieu Martel

Fixed-Point Code Synthesis Based on Constraint Generation*

Sofiane Bessaï[1], Dorra Ben Khalifa,[1]
Hanane Benmaghnia[1], and Matthieu Martel[1,2]

[1] University of Perpignan, LAMPS laboratory, 52 Av. P. Alduy, Perpignan, France
[2] Numalis, 265 Av. des États du Languedoc, Montpellier, France
sofiane.bessai@etudiant.univ-perp.fr, {dorra.ben-khalifa, Hanane.Benmaghnia, matthieu.martel}@univ-perp.fr

Abstract. Fixed-point arithmetic is a well-known alternative to floating-point arithmetic on embedded systems. It is used to reduce some computation costs in terms of speed and power consumption on certain platforms, e.g. medical devices, cars, and robots. In this article, we present POPiX, a novel fixed-point program synthesis tool based on static analysis. The originality of our method is to solve a system of constraints generated from the program source code. Thus, the solution of our constraints gives the new fixed-point formats while accomplishing the accuracy required by the user. Basically, POPiX takes as input an imperative program running in floating-point arithmetic and synthesizes a new program coupled to a fixed-point library relying on integers only. We evaluate POPiX on a collection of floating-point benchmarks coming from FPBench. Results demonstrate the efficiency of our analysis by achieving memory savings up to 75% with energy savings up to 3.5×.

Keywords: Fixed-point arithmetic, code synthesis, precision tuning, linear programming, static analysis.

1 Introduction

Floating-point arithmetic is the dominant approximation to represent a large spectrum of real numbers. Although it offers better precision, programmers do not always need the high level of accuracy offered by the largest floating-point formats. In addition, owing to its complex internal circuitry and the increased memory requirements, floating-point arithmetic can be exorbitant in terms of speed and power consumption on certain platforms such as mobile phones, video game consoles, and digital controllers. To bridge this gap and since many embedded architectures can be implemented using very low bit-width numbers, the solution is to deploy the fixed-point arithmetic as an alternative to the floating-point one as it can be efficiently realized using integer arithmetic. Fixed-point numbers in a certain format maintain a fixed divisor (so the name fixed-point).

* This work is supported by La Région Occitanie under Grant GRAINE - SYFI: https://www.laregion.fr.

With this kind of arithmetic, the number of bits splits into two parts, respectively named integer part and fractional part with a radix point that lies between them. Besides, many fields have revived their interest in fixed-point arithmetic when searching for cost effective hardware processors with less design effort. For instance, machine learning algorithms and models have been recently implemented using fixed-points with little accuracy loss [8,10]. The conversion of code designed for embedded systems from floating to a fixed-point equivalent version is a long-established problem addressed in the litterature [3–6,12]. Nevertheless, rarely we found tools (except for [4]) that deal with adjusting the floating-point formats in the original program before the conversion pass. Practically, programmers do not always need the high level of accuracy in the floating-point formats. However, manually adapting the precision of the variables may require considerable programming skill and application domain expertise. Given this consideration, automating the task of adjusting the program variables precision to improve its performance characteristics, before conversion, can help programmers to achieve performance benefits. Generally, this process of adjustment, also called precision tuning, involves a user requirement of accuracy and a semantic analysis of the program. The benefit of this tuning phase is to provide an optimized mixed-precision programs and pieces of information indicating the most suitable fixed-point data formats in the converted program.

The goal of this article is to propose POPiX: a new tool to transform a given numerical floating-point program into semantically equivalent one that exploits fixed-point computations with integers only. The key idea of this work relies on a semantic modelling of the numerical errors propagation throughout the floating-point program. In order to achieve the conversion, POPiX combines two fundamental steps. The first step consists in generating an Integer Linear Problem (ILP) from the original program in order to obtain the minimal formats (number of bits before and after the radix point) which fulfill the accuracy requirements. Basically, this is done by reasoning on the most significant bit and the number of significant bits of the values. The ILP problem can be optimally solved in one shot by a classical linear programming solver (LP) with no iteration. To the best of our knowledge, this is the first work that interests in statically synthetizing fixed-point code using an ILP formulation of the program. At the end of the tuning phase, the second step collects each information provided by the former step. The tool internally calls a fixed-point library to convert the associated indications into ones that exploit fixed-point computation with the number of bits required for each of the integer and the fractional parts. We evaluate POPiX on a set of benchmarks coming from the FPBench community. The POPiX source code, and all the data and results presented in this article are publicly available at: `https://github.com/sbessai/popix`.

The remainder of this article is organized as follows. In Section 2 we present a motivating example describing our approach. Section 3 provides all the technical details about our new developed fixed-point code synthesis framework. Experimental results are presented in Section 4. We discuss related work in Section 5, and conclude in Section 6.

```
1   [...]
2  NumTuples|10| = 40.0|5,10|;
3  create_vector(x,40);
4  create_vector(y,40);
5  create_vector(z,40);
6   [...]
7  while(i<NumTuples) {
8   x2|-6,10|=x[i]|-3,10| * x[i]|-3,10|;
9   y2|-1,11|=y[i]|-1,11| * y[i]|-1,11|;
10  z2|-6,10|=z[i]|-3,10| * z[i]|-3,10|;
11  tm|-1,10| = x2|-6,6| + y2|-1,11|
12  + z2|-6,5|;
13  magSqrt[i]|-1,10|=sqrt(tm)|-1,10|;
14    i|5,10| = i|5,11| + 1.0|0,6|;} ;
15  [...]
16 while(n<NumTuples) {
17  res|0,11| = 0.0|0,11|;
18  i|0,10| = 0.0|0,10|;
19  while (i<LpfFiltLen) {
20    if (n-i>=0.0) {
21      aux0|2,10| = lpfCoeffs[i]|2,10| *
22      magSqrt[n-i]|-1,10|;
23      res|8,10| = res|8,11| + aux0|2,9|;};
24    i|3,9| = i|3,10| + 1.0|0,7|;} ;
25    n|5,8| = n|5,9| + 1.0|0,4|;};
26    [...]
27 require_nsb(res,8);
```

```
1    [...]
2  int NumTuples; // <0,10>
3  int16_t aux0;  // <2,10>
4  int16_t res; // <8,11>
5  int16_t tm; // <1,10>
6  int8_t derivCoeffs[5]; int16_t x[40];
7  int16_t y[40]; int16_t z[40];
8    [...]
9  while(i<NumTuples) {
10  x2 = (int64_t) (x[i] * x[i]);//<-6, 20>
11  x2 = x2 >> 10;//<-6,10>
12  y2 = (int64_t) (y[i] * y[i]);//<-2,22>
13  y2 = y2 >> 10;//<-2,11>
14  y2 = y2 << 1;//<-1,11>
15  z2 = (int64_t) (z[i] * z[i]);//<-6,20>
16  z2 = z2 >> 10;//<-6,10>
17  x2 = x2 << 5;//<-1,10>
18  y2 = y2 >> 0;//<-1,10>
19  tm = x2 + y2 +z2;  // <-1,10>
20  z2 = z2 << 5; // <-6,10>
21  magSqrt[i]=sqrt_fix(tm,-1,9,8);//<-1,10>
22  i = i + 1;}
23 while(n<NumTuples ) {
24  while(i<LpfFiltLen) {
25    if(n - i >= 0) {
26      aux0=(int64_t)(lpfCoeffs[i]
27      *magSqrt[n-i]);//<1,20>
28      aux0=aux0 >> 9; // <2,10>
29      res = res >>0; // <8,10>
30      aux0 = aux0 << 6; // <8,10>
31      res = res + aux0;} //<8,10>
32  i = i + 1;} n = n + 1;} [...]
```

Fig. 1: Left: tuned program generated with a pair $|\mathsf{ufp}, \mathsf{nsb}|$ for each variable as highlighted in blue. Right: Program C generated with fixed point formats.

2 Overview

Before we dive into the technical details of our tool, we present in this section an overview of our method using the example of a FIR low-pass filter code given in Figure 1. The starting point of our analysis is to assume that all the variables are in a given IEEE754 precision (here we use single precision) and that a range is given for the inputs of the program. In addition, the statement `require_nsb(res,8)` is a postcondition added by the user to specify that `res` must have 8 significant bits at the end of the execution. POPiX first performs a range determination by dynamic analysis for all the program variables at each control point. Based on semantic equations, POPiX generates an ILP problem from the program source code annotated with the results of the range analysis and the accuracy requirement. This yields a system of constraints:

$$C = \left\{ \begin{array}{l} \mathsf{nsb(+)}^{\ell_{548}} >= \mathsf{nsb(res)}^{\ell_{549}}, \mathsf{nsb(res)}^{\ell_{549}} >= \mathsf{nsb(if)}^{\ell_{551}}, \\ \mathsf{nsb(res)}^{\ell_{511}} >= \mathsf{nsb(res)}^{\ell_{545}}, \mathsf{nsb(res)}^{\ell_{545}} >= \mathsf{nsb(+)}^{\ell_{548}} + 8 + carry() - 8 \\ \mathsf{nsb(res)}^{\ell_{545}} >= \mathsf{nsb(+)}^{\ell_{548}} + 8 + carry() - 8, \mathsf{nsb(res)}^{\ell_{548}} >= \mathsf{nsb(res)}^{\ell_{549}}, \\ \mathsf{nsb(aux0)}^{\ell_{547}} >= \mathsf{nsb(+)}^{\ell_{548}} + 6 + carry() - 8, \mathsf{nsb(aux0)}^{\ell_{543}} >= \mathsf{nsb(aux0)}^{\ell_{547}} \end{array} \right\} \quad (1)$$

For instance, the system C of Equation (1) below describes the constraints generated for the addition and assignement statements for Line 23 of Figure 1 (left hand side). Some notations can be highlighted for the system of Equation (1). First, each variable of our program is assigned to a unique control point $\ell \in Lab$ in order to determine easily their number of significant bits. Second, the function $carry()$ is used to compute if a carry bit can occur through the operation (returns 0 or 1). Concerning scalability, we generate a linear number of constraints and variables in the size of the analyzed program (≈ 500 for the FIR low-pass filter code). The solution to our system of constraints gives the minimal number of bits needed with an accuracy guarantee on the results (highlighted in blue in the left hand side of Figure 1). If we take back Line 23 under discussion, the pair $|8, 10|$ denotes that the unit in the first place of variable `res` is 8 whereas it has 10 significant bits. More details about the nature of constraints that we generate for the language of our input programs was detailed in [1].

Based on the tuning results, POPiX synthesizes the C code given in the right hand side of Figure 1. First, it selects the best format (int16_t, int32_t, etc.) for each variable (this is called mixed-precision). For example, at lines 6 and 7, vectors `x`, `y` and `z` are defined as int16_t variables while the vector `derivCoeffs` is defined as int8_t. The data type selected by POPiX for each variable is the minimal one enabling us to encode the fixed-point value following the formats coming from the ILP solution. For example, the variable `res` has 10 significant bits (lines 29 and 31 of Figure 1) and can consequently be encoded into int16_t. POPiX determines the initial formats $\langle M, L \rangle$ of the variables occurring in the code and synthesizes the alignments needed to change the formats, before performing some operation. For example, the shifts performed at lines 29 and 30 are done in order to align the operand of the addition of Line 31. Similarly, the shift of Line 28 is done to obtain the right format for the result of the multiplication of Line 27. Currently, the fixed-point operation are generated sequentially and some additional optimizations could be done, for example by using only one shift for lines 27 and 29.

3 Floating to Fixed-Point Programs Synthesis

POPiX workflow is based on two frameworks as depicted in Figure 3: a developed fixed-point library (Section 3.1) and a precision tuning framework (Section 3.2). In the rest of this section, we explain how the combination of these features are achieved and which benefits they provide.

3.1 Fixed-Point Arithmetic

Since fixed-point operations rely on integer operations, computing with fixed-point numbers is highly efficient for embedded systems with small memories and simpler CPUs. However, this arithmetic is more difficult to handle for the developer. There exists some fixed-point libraries such as Libfixmath[3], Fixmath[4]

[3] `https://code.google.com/archive/p/libfixmath/`
[4] `http://savannah.nongnu.org/projects/fixmath/`

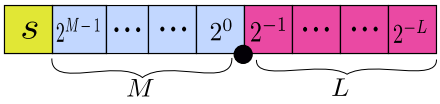

Fig. 2: Fixed-point representation of a in a format $\langle M, L\rangle$.

and FPM[5], but we have developed our own library in order to dispose of all the features we need such as mixed-precision, elementary functions, etc.

A fixed-point number is represented by a sign $s \in \{0, 1\}$, an integer value $V \in \mathbb{N}$ in base 2 and a format $\langle M, L\rangle$ as shown in Figure 2. The number of bits before, respectively after, the radix point is $M \in \mathbb{Z}$ (respectively $L \in \mathbb{N}$). The value of a fixed-point number is obtained by multiplying the integer value V by the sign s and a scaling factor 2^{-L} as follows:

$$a = (-1)^s \times V \times 2^{-L} \ . \tag{2}$$

Example 1 *The fixed-point number* $a = 3_{<2,1>}$ *corresponds to the value 1.5. Using Equation* (2)*, we obtain* $a = (-1)^0 \times 3 \times 2^{-1} = 1.5$.

Let us note that the number of bits M of the integer part already presented in Figure 2 is computed through the unit in the first place (ufp) defined by

$$\forall x \in \mathbb{F}, \quad \mathsf{ufp}(x) = \begin{cases} \min\{i \in \mathbb{Z} : 2^{i+1} > |x|\} = \lfloor \log_2(|x|)\rfloor & \text{if } x \neq 0, \\ 0 & \text{if } x = 0 \ . \end{cases} \tag{3}$$

Hence, the number of bits M before the radix point is given by

$$M = \mathsf{ufp}(|a|) + 1 \ . \tag{4}$$

Let W be the number of bits used to encode a. The number of bits L of the fractional part of a is

$$L = W - M - 1 \ . \tag{5}$$

The difficulty of the fixed-point representation is to manage the format $\langle M, L\rangle$ manually against the floating-point representation which manages it automatically, thanks to the exponent. Let $*$ be a fixed-point elementary operation with $*$ $\in \{\oplus, \ominus, \otimes, \oslash\}$ and let us consider the fixed-point numbers a, b and c such that $c = a * b$. For the addition and subtraction, the resulting format of c is given by

$$\langle M^c,\, L^c\rangle \ = \ \langle \max\left(M^a,\, M^b\right) + 1,\, W^c - \max\left(M^a,\, M^b\right) - 1\rangle \ . \tag{6}$$

For the multiplication and division, the resulting formats of c are

$$\langle M^c,\, L^c\rangle \ = \ \langle M^a + M^b,\, W^c - M^a - M^b > \ \ . \tag{7}$$

and

$$\langle M^c,\, L^c\rangle \ = \ \langle M^a + L^b,\, W^c - M^a - L^b\rangle \ . \tag{8}$$

[5] `https://github.com/MikeLankamp/fpm`

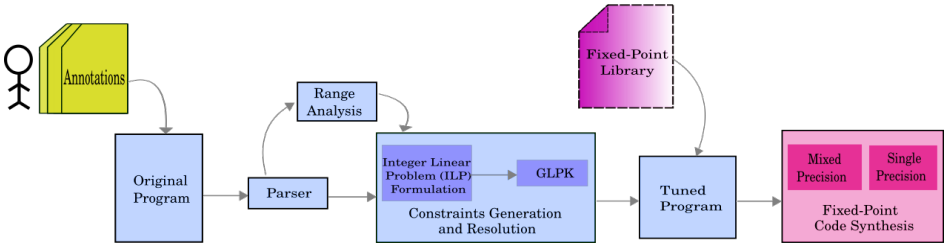

Fig. 3: POPiX workflow.

For a fixed-point number c, the formats of $c \ll k$ and $c \gg k$ are

$$\langle M^c, L^c \rangle = \langle M^a - k, L^c + k \rangle \text{ and } \langle M^c, L^c \rangle = \langle M^a + k, L^c - k \rangle \ . \tag{9}$$

The algorithms of these elementary operations are detailed in [11, 12]. Let us consider a fixed-point elementary function $f \in \{\text{abs, sqrt, sin, cos, arctan}\}$ and the fixed-point number $c = f(a)$. For the square root function, the value of the result can be approximated by the digit recurrence iteration algorithm defined in [12]. For $f \in \{\text{sin, cos, arctan}\}$, Taylor's formula is used to approximate the result. For example, the corresponding formula of sine is $\sin(a) \approx a \oplus (((a \otimes a) \otimes a) \oslash 6)$. The order of the Taylor series development depends on the number of significant bits needed for the result. In the following, we describe how we compute the fixed-point numbers occuring in our programs.

3.2 Constraint Generation by Static Analysis

POPiX presents a novel static technique based on a semantic modelling of the propagation of the numerical errors throughout the code. In practical terms, our approach depends on two integer quantities: *i*) The ufp of the values (see Equation (3)); *ii*) A user requirement denoting the final accuracy wanted for the outputs. Hereby, the term accuracy refers to the number of significant bits required by the user on a variable of the program, denoted by nsb. Formally, let $\hat{x}$ be the approximation of x in finite precision and let $\varepsilon(x) = |x - \hat{x}|$ be the absolute error. So, if $\mathsf{nsb}(x) = k$, for $x \neq 0$, then we have

$$\varepsilon(x) \leqslant 2^{\mathsf{ufp}(x)-k+1} \ . \tag{10}$$

An ILP problem can be generated from the program source code which can be optimally solved by a LP solver (we use GLPK[6], in practice). Concerning our resulting data types, the key feature of our method consists in finding directly the minimal number of bits needed at each control point of the original program. Next, these precisions can be approximated to the upper number of bits corresponding to an existing format int16_t, int32_t, etc. By way of illustration, if a variable x has $\mathsf{nsb}(x) = 18$ bits, then x is tuned to the int32_t format. After

[6] `https://www.gnu.org/software/glpk/`

solving the ILP problem, POPiX collects a new key information concerning the optimized precisions (along with the ufp integer quantity already computed by range analysis) in order to fully specify the fixed-point formats $\langle M, L \rangle$ introduced in Section 3.1. Finally, we call our fixed-point library to synthetize a fixed-point version of the program with only integer numbers. Through this technique, it is possible to achieve memory savings up to more than double with a precision cost that depends on the original program being optimized and energy savings up to 3.5× (see Section 4).

Implementation Details POPiX has been developed in JAVA and C++ and uses the ANTLR tool v4.7.1[7] to parse the input programs. To be able to guarantee that no overflows will occur in computations, we perform a range analysis by launching the execution of the program a certain number of times in order to determine dynamically an under-approximation of the range of variables by using the ufp of the values. In the future, we plan to use a static analyzer. Nevertheless, POPiX uses the simple imperative language below.

$$x \in Id \quad \ell \in Lab \quad \odot \in \{+, -, \times, \div\} \quad math \in \{\sin, \cos, \tan, \arcsin, \log, \ldots\}$$
$$\textbf{Expr} \ni \mathbf{e} : \mathrm{e} ::= \mathrm{c}\#p \mid x \mid e_1^{\ell_1} \odot e_2^{\ell_2} \mid math(e^{\ell_1}) \mid sqrt(e^{\ell_1})$$
$$\textbf{Cmd} \ni \mathbf{c} : c ::= c_1^{\ell_1}; c_2^{\ell_2} \mid x = e^{\ell_1} \mid \texttt{while}\ b^{\ell_0}\ \texttt{do}\ c_1^{\ell_1} \mid \texttt{if}\ b^{\ell_0}\ \texttt{then}\ c_1^{\ell_1}\ \textit{else}\ c^{\ell_2} \mid \texttt{create\_vector}(v, s) \mid \texttt{create\_matrix}(m, r, c) \mid \texttt{require\_nsb}(x, n)$$

We denote by Id the set of identifiers and by Lab the set of control points of the program used to assign to each element $e \in$ **Expr** and $c \in$ **Cmd** a unique control point $\ell \in Lab$. Fortunately, POPiX is able to handle loops, conditionals and arrays. The declaration of vectors is expressed by the statement `create_vector`(v,s), while s denotes the size of the vector v. The declaration of a matrix m is expressed by the statement `create_matrix` (m, r, c) , while r and c denote respectively the number of rows and columns of the matrix. The statement `require_nsb`(x,n) indicates the minimal nsb n that a variable x must have at a control point. The rest of the grammar is standard. Note that the usual mathematical elementary functions are supported.

Cost functions Cost functions are given as optimization objective to the linear solver. Depending on which cost function is used, different criteria may be considered for the tuning phase. POPiX currently handles the following cost functions: **CF1** Optimizes the sum of the number of significant bits of all the variables at each control point of the program. **CF2** Optimizes the sum of accuracies of only the variables assigned in the program. Compared to **CF1**, this cost function minimizes the size of the variables and the number of bits needed for the operators to store the intermediary results. Let us note that **CF2** is used in the experiments of Section 4. **CF3** Minimizes the maximal accuracy needed in the program, i.e. the worst accuracy at some control point of the tuned program. This function is usefull to make a program fit in a certain format (for example all variables in 16 or 32 bits.) **CF4** Optimizes only the sum of the accuracies of the arithmetic operators and elementary functions. This function is relevant

[7] `https://www.antlr.org/`

from an hardware point of view, for example to limit the size of the operators in FPGAs [7]. **CF5** Minimizes the number of type conversions. Indeed, type conversions introduced by the mixed-precision tuning may slow down the execution of the programs and one may prefer a compromise between memory savings and execution time. This function addresses this problem. Note that these cost functions are modified when dealing with arrays: the tool multiplies the precision by the number of elements and this process is done only once for each array instead of several times for each use of arrays.

4 Experimental Evaluation

In this section, we conduct some experiments to show the effectiveness of our code synthesis method presented in Section 3.

Experimental Metrics In our experiments, our goal is to evaluate the benefits of POPiX in terms of mixed-precision, memory savings and energy consumption which are important metrics to validate our synthesis method. We also measure the time of analysis spent by POPiX and the execution time of the fixed-point progam with respect to the floating-point program in which we assume that all variables are in single precision (32 bits) before the analysis. For our benchmarks, we use applications from FPBench[8], a synthetic benchmark for floating-point performance. We run each program with three accuracy requirements arbitrarily chosen by the user: 4, 8 and 16 bits which bound the relative error of the result. All the results we report in this section where gathered on two machines: Ubuntu 20.04 LTS, with an 2.7GHz i7 core and 16 GB of RAM and Ubuntu 20.04 LTS, with a CPU AMD Ryzen 5 3500u and 5.7 GB of RAM. Let us state that the reason we use the Intel machine is to exploit the Jouleit[9] tool in order to estimate the power consumption of the CPU, RAM and integrated GPU.

Results Analysis Table 1 shows the mixed-precision configurations obtained after analysis in terms of number of variables or operations that we may tune into int8_t, int16_t and int32_t and consequently the memory savings in terms of number of bits. The second left-most column of Table 1 headed "call" refers to the number of elementary functions in the code and the next column headed "op" denotes the number of elementary operations. In this experiment, we assume that 100% is the percentage of all variables initially in single precision. Clearly, the memory savings compared to the initial number of bits for the majority of the original programs is considerable reaching 75% for "CRadius" program for a requirement of 4 bits. For instance, the "carbonGas" program has a total of 13 variables all in single precision before analysis. In the synthesized code we obtain 7 variables tuned into int8_t and 6 variables in int16_t achieving a gain in number of bits of 63,5%. Concerning "azimuth"program, our analysis failed to infer mixed precision for a user accuracy requirement of 16 bits whereas it

[8] `https://fpbench.org/`
[9] `https://github.com/powerapi-ng/jouleit`

			4 bits				8 bits				16 bits			
Program	call	op	8	16	32	%	8	16	32	%	8	16	32	%
azimuth	7	7	1	0	17	**4.2**	0	1	17	**2.8**	-	-	-	**-**
carbonGas	0	7	7	6	0	**63.5**	2	11	0	**53.8**	1	1	11	**9.6**
CRadius	1	3	6	0	0	**75.0**	0	6	0	**50.0**	0	0	6	**0.0**
CTheta	1	3	4	0	3	**42.9**	0	4	3	**28.6**	0	0	7	**0.0**
doppler1	0	7	9	1	0	**72.5**	1	9	0	**52.5**	0	1	9	**5.0**
doppler2	0	7	9	1	0	**72.5**	3	7	0	**57.5**	0	3	7	**15.0**
doppler3	0	7	9	1	0	**72.5**	2	8	0	**55.0**	0	2	8	**10.0**
instantCurrent	3	18	7	14	7	**43.8**	3	18	7	**40.2**	0	3	25	**5.4**
jetEngine	0	29	7	18	5	**47.5**	3	15	12	**32.5**	0	3	27	**5.0**
LeadLagSystem	1	17	2	29	2	**48.5**	0	31	2	**47.0**	0	0	33	**0.0**
LowPassFilter	0	0	0	330	0	**50.0**	0	330	0	**50.0**	0	4	326	**0.6**
CX	1	3	2	0	4	**25.0**	0	2	4	**16.7**	0	0	6	**0.0**
CY	1	3	2	0	4	**25.0**	0	2	4	**16.7**	0	0	6	**0.0**
triangle12	1	9	7	6	0	**63.5**	0	12	1	**46.2**	0	0	13	**0.0**
turbine1	0	14	4	13	0	**55.9**	0	17	0	**50.0**	0	0	17	**0.0**
turbine2	0	10	10	3	0	**69.2**	0	13	0	**50.0**	0	0	13	**0.0**
turbine3	0	14	3	14	0	**54.4**	0	17	0	**50.0**	0	0	17	**0.0**

Table 1: Mixed fixed-point formats in 4, 8 and 16 bits for the synthesized program with the precentages of the number of bits saved.

reaches 4.2% and 2.8% respectively for requirements of 4 bits and 8 bits. Another observation is that for a requirement of only 4 bits, 17 variables are tuned into int32_t. A possible explanation of this result is the call to the elementary functions in the code (call = 7) which can use intermediate variables that request greater precision than the user accuracy requirement.

Figure 4 depicts the energy consumed by the execution of the benchmarks for a requirement of 4 bits. We observe that the fixed-point version codes require significantly less energy than the floating-point codes. For instance, the energy saved on CPU and DRAM reaches $\approx$ 43% for "carbonGas" program and more than 75% for "jetEngine" program. Let us note that this observation is also valid for the remaining user requirements with slight variations of savings. Finally, we present the results in terms of speed for each of our benchmarks in Table 2. We denote respectively by "t_{float}","$t_{synthesis}$" and "t_{fix}" the time of execution of floating-point programs, the total synthesis time of POPiX and the execution time of fixed-point programs all given in milliseconds. We visualize that the time spent by POPiX for the majority of benchmarks is negligible not exceeding 342 ms for the "carbonGas" program ($\approx$ 30 LOCs). Although our synthesis method is fast (few seconds), we observe that the execution time remains the same for the floating and fixed-point codes with negilible slow-down for some benchmarks.

5 Related Work

In recent years, many authors have investigated the possibility of automating fixed-point code synthesis. A similar approach to our work is the TAFFO tool proposed by Cherubin et al. [4]. TAFFO is a static precision tuning tool that

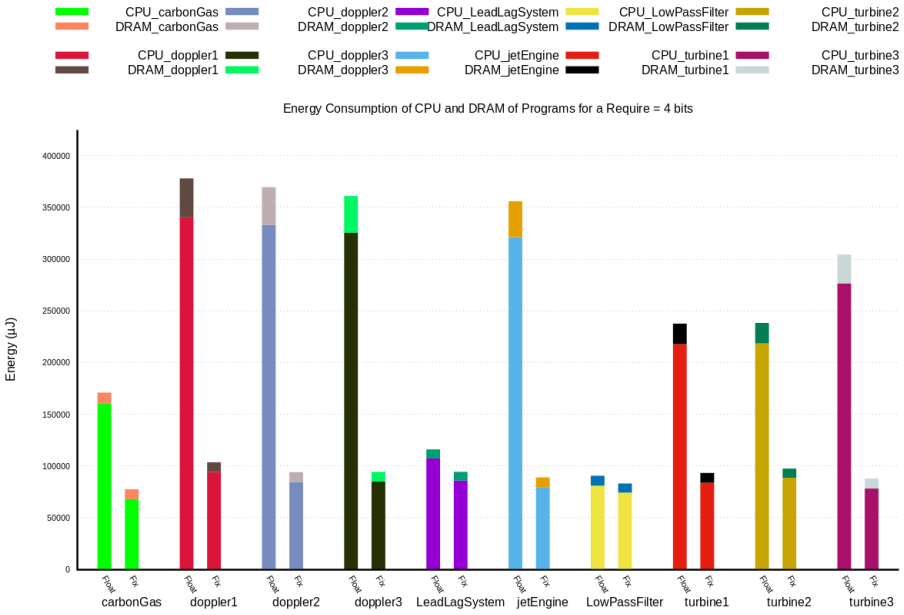

Fig. 4: Measurement of the energy consumption (CPU and DRAM) of the floating-point and fixed-point version of our benchmarks.

converts floating-point computations into a fixed-point version with comparable semantics. The common point between POPiX and TAFFO is that the estimation of the errors is generated by the precision tuning process. Meanwhile, POPiX is much faster and takes only few seconds to synthesize the new fixed-point formats of the program.

Another solution for code conversion was introduced by Cattaneo et al. [3]. Their method relies on a self-contained compiler transformation pass implemented within LLVM to perform the conversion. Their tool was especially dedicated to MIOSIX, a real time operating system targeting embedded system.

The goal of the dissertation of Jha [9] is to give an algorithm for optimal fixed-point expressions synthesis based on inductive synthesis. Two years later, Darulova et al. [5] proposed a fixed-point program synthesis methodology based on expression rewriting and genetic programming. Their algorithm uses abstract interpretation to estimate the error bound of a fixed-point implementation. However, the latter two techniques provide pessimistic bounds for non-linear expressions and are limited to straight-line programs. Coversely, Aslan et al. [2] developed a tool that takes an n-bit fixed-point input and creates an m-bit floating-point output with IEEE754 and custom formats.

In the context of polynomials, linear filters and signal processing algorithms, the members of the DEFIS project [13] presented many approaches for fixed-point code synthesis. To mention a few, the idea described in [6] works by infer-

Program	t_{float}	4 bits		8 bits		16 bits	
		$t_{synthesis}$	t_{fix}	$t_{synthesis}$	t_{fix}	$t_{synthesis}$	t_{fix}
azimuth	1.91	340	1.8	301	2.6	233	3.2
carbonGas	0.17	342	0.29	233	0.31	201	0.46
CRadius	0.10	240	1.31	192	1.33	186	1.36
CTheta	0.38	203	0.56	223	0.57	273	0.57
doppler1	0.16	205	0.53	249	0.54	225	0.57
doppler2	0.24	188	0.28	225	0.29	207	0.30
doppler3	0.17	243	0.30	195	0.32	207	0.34
instantCurrent	1.01	264	1.99	265	2.36	273	2.73
jetEngine	0.34	294	1.18	264	1.24	276	1.82
LeadLagSystem	0.53	352	0.71	394	0.86	346	1.09
CX	0.43	178	0.29	208	0.37	175	0.52
CY	0.39	203	0.41	197	0.55	197	0.68
triangle12	0.17	199	1.57	208	1.63	203	2.53
turbine1	0.24	220	0.31	228	0.35	269	0.49
turbine2	0.28	246	0.49	222	0.61	212	1.01
turbine3	0.48	247	0.47	253	0.49	217	0.83

Table 2: Execution time measurements obtained during the experiments.

ring high-level convolution operations from the original source code, and modeling them as part of the program representation. In addition, Najahi et al. [12] presented an automated approach to synthesize codes in fixed-point arithmetic for some linear algebra basic blocks. They take a mathematical description of the problem as well as the range of the input variables and generate fixed-point code. Lopez [11] addresses the transformation of linear filters and controllers into hardware operators using fixed-point arithmetic. His main contribution is a complete error analysis, with respect to the internal word-lengths and the formulation of the word-length optimization as a convex non-linear integer optimization problem solved using appropriate heuristics. An extension of this work to the full class of linear time invariant algorithms has been proposed in [14].

6 Conclusion

In this article, we have presented a new method for fixed-point code synthesis respecting an accuracy requirement imposed by the user. Our method is based on a static analysis of the code implemented by means of system of constraints which gives the minimal format needed to encode each value. Experimental results show the performance of the codes synthesized in terms of execution time, memory and energy savings on a set of benchmark related to embedded systems.

In the future we would like to validate our method by considering architectures more commonly used in embedded systems. Also, we aim at generating hardware instead of software fixed-point implementations, using FPGAs. Targeting FPGAs has two justifications: this type of hardware is becoming more and more popular today and it presents the advantage of allowing fully custom designs. Finally, for adoption reasons in real-world applications, we aim at extending POPiX in order to handle full C programs via an integration to LLVM.

References

1. Assalé Adjé, Dorra Ben Khalifa, and Matthieu Martel. Fast and efficient bit-level precision tuning. In *Static Analysis - 28th International Symposium, SAS*, volume 12913 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2021.
2. Semih Aslan, Erdal Oruklu, and Jafar Saniie. A high-level synthesis and verification tool for fixed to floating point conversion. In *55th IEEE International Midwest Symposium on Circuits and Systems, MWSCAS*, pages 908–911. IEEE, 2012.
3. Daniele Cattaneo, Antonio Di Bello, Stefano Cherubin, Federico Terraneo, and Giovanni Agosta. Embedded operating system optimization through floating to fixed point compiler transformation. In *21st Euromicro Conference on Digital System Design, DSD*, pages 172–176. IEEE Computer Society, 2018.
4. Stefano Cherubin, Daniele Cattaneo, Michele Chiari, and Giovanni Agosta. Dynamic precision autotuning with TAFFO. *ACM Trans. Archit. Code Optim.*, 17(2):10:1–10:26, 2020.
5. Eva Darulova, Viktor Kuncak, Rupak Majumdar, and Indranil Saha. Synthesis of fixed-point programs. In *Proceedings of the International Conference on Embedded Software, EMSOFT*, pages 22:1–22:10. IEEE, 2013.
6. Gaël Deest, Tomofumi Yuki, Olivier Sentieys, and Steven Derrien. Toward scalable source level accuracy analysis for floating-point to fixed-point conversion. In *The IEEE/ACM International Conference on Computer-Aided Design, ICCAD*, pages 726–733. IEEE, 2014.
7. Xitong Gao and George A. Constantinides. Numerical program optimization for high-level synthesis. In George A. Constantinides and Deming Chen, editors, *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 210–213. ACM, 2015.
8. Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 1737–1746. JMLR.org, 2015.
9. Susmit Jha. *Towards Automated System Synthesis Using SCIDUCTION*. PhD thesis, University of California, Berkeley, USA, 2011.
10. Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 2849–2858. JMLR.org, 2016.
11. Benoit Lopez. *Implémentation optimale de filtres linéaires en arithmétique virgule fixe. (Optimal implementation of linear filters in fixed-point arithmetic)*. PhD thesis, Pierre and Marie Curie University, Paris, France, 2014.
12. Matthieu Martel, Amine Najahi, and Guillaume Revy. Trade-offs of certified fixed-point code synthesis for linear algebra basic blocks. *J. Syst. Archit.*, 76:133–148, 2017.
13. Daniel Ménard, Romuald Rocher, Olivier Sentieys, Nicolas Simon, Laurent-Stéphane Didier, Thibault Hilaire, Benoit Lopez, Eric Goubault, Sylvie Putot, Franck Védrine, Amine Najahi, Guillaume Revy, L. Fangain, Christian Samoyeau, Fabrice Lemonnier, and Christophe Clienti. Design of fixed-point embedded systems (DEFIS) french ANR project. In *Design and Architectures for Signal and Image Processing, DASIP*, pages 1–2. IEEE, 2012.
14. Anastasia Volkova. *Towards reliable implementation of digital filters. (Vers une implémentation fiable des filtres numériques)*. PhD thesis, Pierre and Marie Curie University, Paris, France, 2017.