Automated Synthesis of Target-Dependent Programs for Polynomial Evaluation in Fixed-Point Arithmetic

Christophe Mouilleron, Mohamed Amine Najahi, Guillaume Revy

Automated Synthesis of Target-Dependent Programs for Polynomial Evaluation in Fixed-Point Arithmetic

Christophe Mouilleron[1] Amine Najahi[2,3,4] Guillaume Revy[2,3,4]

[1] ENSIIE, F-91025, Évry, France

[2] Univ. Perpignan Via Domitia, DALI, F-66860, Perpignan, France

[3] Univ. Montpellier II, LIRMM, UMR 5506, F-34095, Montpellier, France

[4] CNRS, LIRMM, UMR 5506, F-34095, Montpellier, France

christophe.mouilleron@ensiie.fr amine.najahi@univ-perp.fr guillaume.revy@univ-perp.fr

***Abstract*—The design of both fast and numerically accurate programs is a real challenge. Thus, the CGPE tool was introduced to assist programmers in synthesizing fast and numerically certified codes in fixed-point arithmetic for the particular case of polynomial evaluation. For performance purposes, this tool produces programs using exclusively unsigned arithmetic and addition/subtraction or multiplication operations, thus requiring some constraints on the fixed-point operands. These choices are well-suited when dealing with the implementation of certain mathematical functions, however they prevent from tackling a broader class of polynomial evaluation problems. In this paper, we first extend the arithmetic model of CGPE to handle signed arithmetic and alignment shifts. Then, in order to make the most out of advanced instructions, we propose an enhancement of this tool based on instruction selection. This allows us to optimize the generated codes according to different criteria, like operation count, evaluation latency, or accuracy. Finally, we illustrate this technique on operation count, and we show that it yields an average reduction of up to 22.3 % of the number of operations in the synthesized codes of some functions.**

Keywords*—*fixed-point arithmetic, automated code synthesis, error analysis, polynomial evaluation

I. Introduction

Polynomials are widely used in computer science, since they can be evaluated using only addition and multiplication, two of the cheapest and most ubiquitous instructions on modern architectures. In computer arithmetic, for example, polynomial evaluation appears frequently as a building block for the floating-point implementation of a mathematical function. The speed and the accuracy of these implementations depend directly on those of the underlying code used to evaluate the polynomial. However, as shown in [1, §6], the number of evaluation schemes of a given polynomial is quite huge, even for small degree univariate polynomials. Moreover these schemes may offer different numerical qualities, evaluation latencies, or instruction-level parallelism (ILP) exposures. Thus the combinatorics makes it difficult to choose a *good* evaluation scheme, since exhaustive testing is not feasible. Hence tools and heuristics are needed to automate the synthesis of polynomial evaluation programs and to validate their numerical quality.

Various projects have been set up in order to assist programmers in writing automatically efficient routines, either in floating-point or fixed-point arithmetic, and not especially for the purpose of evaluating polynomials. The SPIRAL project[1] proposes software and hardware dedicated tools capable of synthesizing fast floating-point codes for DSP algorithms. In [3], the authors deal with code synthesis to evaluate dot-products in fixed-point arithmetic in order to implement IIR filters. A different approach consists of starting from an existing code, and applying a series of transformations either to improve its numerical accuracy in floating-point arithmetic [4] or to minimize the width of integer parts in fixed-point computations [5]. Yet another option is to convert the existing program into an equivalent one but using another arithmetic. This latter technique is investigated in [6] where the authors focus on the conversion of codes from floating-point to fixed-point arithmetic. For the special case of polynomial evaluation, [7] proposes an approach based exclusively on Horner's rule. This method is well-known for minimizing the number of multiplications involved and for having a good numerical quality, especially when the polynomial is evaluated not too close to one of its zeros [8], but it fails at exposing ILP. Finally, [9] suggests a method for the generation of evaluation schemes using only the FMA operator available on the Itanium® architecture, while [10] deals with the generation of evaluation schemes using at best the SIMD instructions of the PlayStation®2 cores.

In this paper, we present an extended version of the CGPE software tool [11], standing for Code Generation for Polynomial Evaluation. This tool relies on heuristics to address the automated generation of fast and accurate codes to evaluate polynomials in fixed-point arithmetic. In fact, even though floating-point arithmetic is more and more used, some architectures are still shipped without any floating-point unit in order to satisfy some area, energy consumption, or conception cost constraints. On these targets, to reach good performance on the generated codes, the original version of CGPE made the following assumptions:

- Only unsigned arithmetic is handled. Thus, sign bits are not stored, which allows to save at least one bit of accuracy on each computed value during the evaluation.
- Addition, subtraction, and multiplication are the only operators used.

[1] See http://www.spiral.net/ and [2].

However, these assumptions reduce the set of problems that can be tackled: the polynomial evaluation must be feasible in unsigned fixed-point arithmetic without inserting shifts to adapt the fixed-point format of intermediate variables. Moreover, by generating codes that rely only on addition, subtraction and multiplication, the tool cannot benefit from all the features of the target hardware. This is particularly true since a greater number of architectures are shipped with advanced instructions such as those combining two binary operations in a single one, like the fused multiply-add (FMA).

This paper makes the following contributions:

- First we extend and formalize the arithmetic model of CGPE to handle signed fixed-point arithmetic, as well as right and left shift operators. This extension clears the path to tackling more general problems, especially since there is no more constraints on the sign and format of the input coefficients and variables.
- Second we add a step based on instruction selection, that works on the DAGs (Directed Acyclic Graphs) computed by CGPE. These DAGs are the intermediate representation of polynomial evaluation schemes inside the tool. The role of this step is to fuse nodes in advanced instructions, and more particularly to choose for each node or set of nodes the instruction to be used to optimize a given criterion, like evaluation latency, operator count, or accuracy. A similar technique has already been investigated by the SPIRAL project to use FMA operators in the implementation of DSP or linear transform algorithms [12], [13].

This paper is organized as follows: After a presentation of the CGPE software tool in Section II, the fixed-point arithmetic model we propose is detailed in Section III. Section IV is devoted to our second contribution, that is, the synthesis of target-dependent codes using instruction selection. Some examples are given in Section V before concluding in Section VI.

II. The CGPE tool

CGPE[2] addresses the automated synthesis of polynomial evaluation programs in fixed-point arithmetic. It focuses on both algorithm speed and accuracy of the generated codes, by adding a systematic certification step. The following of this section presents its input and output, together with its workflow and limitations.

A. Input and output of CGPE

CGPE takes as input a polynomial and a set of criteria and architectural constraints. The polynomial is described in an external XML file, that contains an interval of values and a fixed-point format for each coefficient and variable, as well as a maximum error bound allowed for its evaluation. In addition to this file, CGPE takes a set of command-line parameters, like a bound on the evaluation latency, the latency of each basic operator, and some information on the strategy used during the computation step detailed in Section II-B below. Besides, since the tool was initially intended for use with VLIW processors, additional parameters are provided to set the available level of parallelism, that is, the number of issues on the target.

At the end of the process, CGPE produces a set of C codes evaluating the input polynomial on the given target. First for each synthesized code, it ensures that the evaluation satisfies the latency criterion. Second it guarantees that the evaluation error is less than the maximum error bound, by attaching a Gappa[3] certificate file to each code. The latter tool uses formal verification techniques to prove that the evaluation error entailed in the C code is below a given threshold. Listing 1 shows an example of an automatically produced code

```
//  a0 = +0x7ffec8d0p-30     a1 = -0x7f9bef55p-30
//  a2 = +0x7ab5c54bp-30     a3 = -0x647d671dp-30
//  a4 = +0x379913e9p-30     a5 = -0x0e358cb5p-30
uint32_t func_d5(uint32_t x /* 0.32 */) { // Formats
  uint32_t r0  = mul(x, 0x7f9bef55);      // 2.30
  uint32_t r1  = 0x7ffec8d0 - r0;         // 2.30
  uint32_t r2  = mul(x, x);               // 0.32
  uint32_t r3  = mul(x, 0x647d671d);      // 2.30
  uint32_t r4  = 0x7ab5c54b - r3;         // 2.30
  uint32_t r5  = mul(r2, r4);             // 2.30
  uint32_t r6  = r1 + r5;                 // 2.30
  uint32_t r7  = mul(r2, r2);             // 0.32
  uint32_t r8  = mul(x, 0x0e358cb5);      // 2.30
  uint32_t r9  = 0x379913e9 - r8;         // 2.30
  uint32_t r10 = mul(r7, r9);             // 2.30
  uint32_t r11 = r6 + r10;                // 2.30
  return r11;
}
```

Listing 1. Example of code automatically generated using CGPE.

for the evaluation of a degree-5 polynomial approximant of the function $1/(1+x)$ over $[0,1]$ with an evaluation error bound of $3213 \cdot 2^{-26} \approx 2^{-14.35}$. This code is optimized for a target having 1-cycle addition and subtraction and 3-cycle fully pipelined multiplication, and that can launch 4 instructions each cycle with at most 2 multiplications. The two main goals of CGPE are achieved through this example:

1) A certificate is produced, that proves the evaluation error is no greater than the required evaluation error bound.[4]
2) The generated code in Listing 1 has a latency of 10 cycles on the target processor, which is optimal. In fact, it can be proven, for this example, that a lower latency cannot be achieved even on unbounded parallelism and regardless of the tolerated evaluation error bound.

B. Workflow of CGPE

As illustrated in Figure 1, CGPE works in three steps and has a compiler-like architecture.

First, CGPE starts by a computation step that plays the part of a compiler's front-end. It computes a set of DAGs (Directed Acyclic Graphs), each DAG being the intermediate representation of a given polynomial evaluation scheme inside the tool. During this step, unbounded parallelism is assumed, and only the latency of each basic operator (adder, subtracter, and multiplier) is considered. Figure 2 shows the DAG representing the evaluation code of Listing 1 above. However, unlike a classical compiler's front-end, CGPE computes several DAGs since given an input polynomial, several evaluation schemes

[2]See http://cgpe.gforge.inria.fr and [14], [11].

[3]See http://gappa.gforge.inria.fr and [15].

[4]For the sake of space, the Gappa certificate is available upon request.

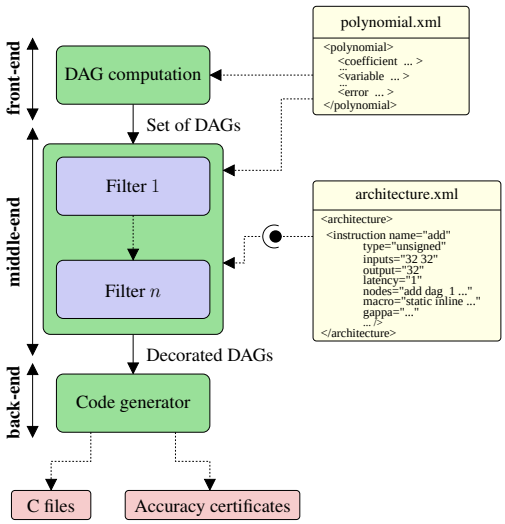


Figure 1. Dataflow path inside CGPE.

are possible. At this step, CGPE ensures that these DAGs reduce the evaluation latency on unbounded parallelism. Various strategies have been developed to produce these DAGs [14, §6][1, §8.1.2]. Second, CGPE goes to a filtering step that can be seen as a compiler's middle-end. At this step, each DAG undergoes a series of filters, each filter being dedicated to a criterion and deciding whether to keep the scheme or to prune it. DAGs that pass all the filters reach CGPE's back-end, which takes care of producing the code and the associated accuracy certificate.

CGPE's architecture makes it easy to optimize the synthesized code for different criteria, by simply adding more constraints in the computation step or new filters. Moreover the computation algorithms are designed to build DAGs using only binary operations, typically addition/subtraction and multiplication. These are already complex, and handling other binary operations or operations with higher arity seems to be unreachable at computation time (front-end), especially since this would require the development of a new DAG computation algorithm each time a new operation is considered. However making the most out of the target architecture is feasible at filtering step, by tweaking only the middle-end. In this context we suggest an extension of CGPE based on instruction selection, and presented in Section IV below.

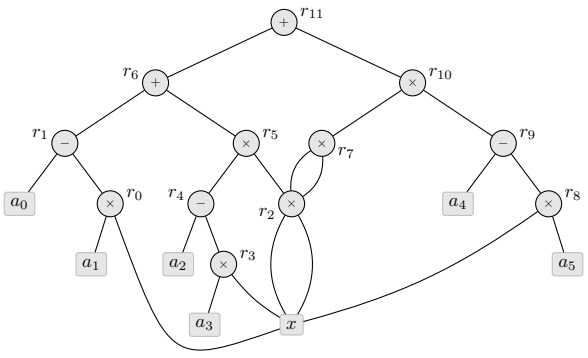


Figure 2. DAG representing the evaluation code in Listing 1.

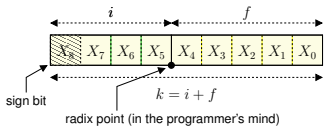


Figure 3. Fixed-point number representation.

III. Fixed-point arithmetic model

Fixed-point arithmetic consists in interpreting an integer as a rational value [16]. In other words, it allows the manipulation of real values by the means of integers coupled with an implicit scaling factor. Here, *implicit* means that it is not encoded into the data of the program, but fixed and only known by the programmer. It follows that each variable appearing in a fixed-point program is supposed to have a fixed-point format. This makes fixed-point programs development quite difficult, more particularly since, unlike floating-point arithmetic, there is no standard guiding the implementation of fixed-point arithmetic. This section presents the fixed-point arithmetic model on which our extension of CGPE is based.

A. Fixed-point number representation

Let X be a k-bit radix-2 integer and $f \in \mathbb{Z}$. Combined with the *implicit* scaling factor f, the integer X represents the real value x defined as follows:

$$x = X \cdot 2^{-f}.$$

We detail here the principle of signed fixed-point arithmetic, unsigned arithmetic being easily deducible from it. In this context, X is usually encoded using two's complement notation. And it follows that:

$$X \in \{-2^{k-1}, \cdots, 2^{k-1} - 1\}$$

and

$$x \in \{X \cdot 2^{-f}\}_{X \in \{-2^{k-1}, \cdots, 2^{k-1}-1\}}.$$

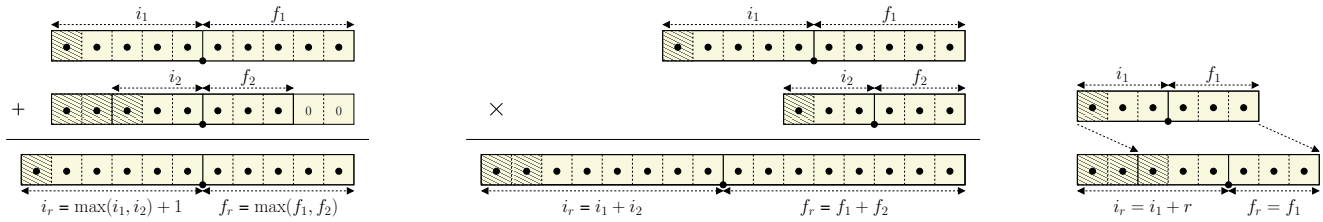
Note that f denotes the number of fraction bits in the binary expansion of x, while the number of bits in the integer part is i, as shown in Figure 3. This value will be encoded in a total of $k = i + f$ bits. Obviously the sign of x depends on the sign of the integer X, and using two's complement notation, it is encoded in the most significant bit of X.

From now on, we will denote by $\mathcal{Q}_{i,f}$ the fixed-point format of a given signed fixed-point value v having an i-bit integer part and a f-bit fractional part. It follows that

$$v \in [-2^{i-1}, 2^{i-1} - 2^{-f}].$$

B. Arithmetic rules

Our extension of CGPE manipulates DAGs, whose nodes are now either addition, subtraction, multiplication, or left or right shift operations. The following of this section presents the semantics of these operations in fixed-point arithmetic. Recall that we consider here signed arithmetic using two's complement notation. Now let v_1 and v_2 be two signed fixed-point values in the formats $\mathcal{Q}_{i_1,f_1}$ and $\mathcal{Q}_{i_2,f_2}$, respectively.



(a) Addition. (b) Multiplication. (c) Right shift of 2 bits.

Figure 4. Fixed-point arithmetic rules for addition, multiplication, and right shift.

1) Addition and subtraction: To be added or subtracted, the two values must be in the same format, that is, with the same integer and fractional part sizes. This *comma alignment* can be done by adding trailing zeros in the binary expansion of the operand with the smallest fractional part, and by extending the sign representation size of the operand with the smallest integer part, as shown in Figure 4(a) for the addition. In this case, the alignment can be seen as a shift operator, described below. It follows that the result of the addition between v_1 and v_2 is in the format $\mathcal{Q}_{i_r,f_r}$ with

$$i_r = \max(i_1, i_2) + 1 \quad \text{and} \quad f_r = \max(f_1, f_2).$$

Notice that the most significant bit of the result is there to prevent overflow issues. If we can ensure that no overflow occurs while adding both values, we can reduce the format of the result, and thus gain one bit of accuracy to the detriment of inserting a shift into the DAG and possibly increasing the critical path. In this case, we have $i_r = \max(i_1, i_2)$. This decision may be made using interval arithmetic.

2) Multiplication: Concerning the multiplication, there is no constraints on the format of the operands. Multiplying v_1 and v_2 results in a fixed-point value in the format $\mathcal{Q}_{i_r,f_r}$ with

$$i_r = i_1 + i_2 \quad \text{and} \quad f_r = f_1 + f_2,$$

as shown in Figure 4(b). Note that the most significant bit is a redundant bit and can be discarded. Nevertheless an exception occurs in signed arithmetic when the operands are both equal to the smallest negative value, that is, in our example $v_1 = -2^{i_1-1}$ and $v_2 = -2^{i_2-1}$. In that case, only the most significant bit represents the sign of the result and it cannot be discarded.

Observe that a multiplication by a power of 2 can be interpreted as a *virtual shift*. A virtual shift simply consists of changing the fixed-point format of the operand, and thus the encoded real value, but it does not imply any true operation during the evaluation on the target. Hence the interest is that a virtual shift is cost-free, and does not impact the evaluation program latency. If we detect that one of the operands is a power of 2, we can replace the effective multiplication by a virtual shift. For example, the multiplication of v_1 by 2^p results in a fixed-point value in the format $\mathcal{Q}_{i_r,f_r}$ with

$$i_r = i_1 + p \quad \text{and} \quad f_r = f_1 - p.$$

3) Left and right shifts: In fixed-point arithmetic, the shift operation implies the change of the format of the operand, according to its direction. Here this change is actually a side effect of the shift performed on the encoding of the integer representation, allowing to align operand mantissas or to remove leading redundant sign bits, for example.

- Shifting the value v_1 of r bit to the right results in a fixed-point value in the format $\mathcal{Q}_{i_r,f_r}$ with

  $$i_r = i_1 + r \quad \text{and} \quad f_r = f_1,$$

  as shown in Figure 4(c) for $r = 2$.
- Shifting the value v_1 of ℓ bit to the left results in a fixed-point value in the format $\mathcal{Q}_{i_r,f_r}$ with

  $$i_r = i_1 \quad \text{and} \quad f_r = f_1 + \ell.$$

In the same way as for the addition, increasing the size of the integer part relies on sign extension, while increasing the fractional part consists in adding trailing zeros.

Let us remark that the format of each intermediate variable can be simply computed by bottom-up scanning the DAG and by systematically applying these arithmetic rules.

C. Error analysis rules

All the previous arithmetic rules are error-free. However fixed-point arithmetic usually relies on integer arithmetic available on the target. Hence, due to finite precision issues, each fixed-point operation may entail a rounding error.

Let $\mathcal{G}$ be a node of a DAG, and $\mathcal{G}_\ell$ and $\mathcal{G}_r$ be its left and right children, respectively. To decide the accuracy of each intermediate variable, CGPE attaches two intervals to each node $\mathcal{G}$: **value**($\mathcal{G}$) that bounds the value resulting from the evaluation of $\mathcal{G}$, and **error**($\mathcal{G}$) that bounds the error between the computed and the mathematical values due to the usage of finite precision [11, §4-B]. Note that if $\mathcal{G}$ is a leaf (that is, a variable or a coefficient), then **value**($\mathcal{G}$) is read from the XML input file and **error**($\mathcal{G}$) $= [0, 0]$. Otherwise the computation of these quantities is done by bottom-up scanning the DAG using multiple-precision interval arithmetic with the MPFI[5] library.

The sequel of this section presents how both intervals are computed in the case of signed arithmetic. Again let v_1 and v_2 be two signed fixed-point values in the formats $\mathcal{Q}_{i_1,f_1}$ and $\mathcal{Q}_{i_2,f_2}$, respectively. Here we consider that the format of each fixed-point value has been computed and the alignment shifts have been inserted in the DAG as presented above. It follows that no overflow occurs during the evaluation.

[5] See http://mpfi.gforge.inria.fr/ and [17].

1) Addition and subtraction: In absence of overflow, fixed-point addition and subtraction are error-free. Hence, if $\mathcal{G}$ represents an operation $\diamond \in \{+, -\}$, we have:

$$\textbf{error}(\mathcal{G}) = \textbf{error}(\mathcal{G}_\ell) \diamond \textbf{error}(\mathcal{G}_r).$$

2) Multiplication: The error entailed by the multiplication depends on the behavior of the underlying multiplication instruction and the fixed-point format of the result. In most cases, the instruction returns the k most significant bits of the exact result, that is, a truncated result. Hence, we have:

$$\begin{aligned}\textbf{error}(\mathcal{G}) \quad = \quad & \textbf{error}_{\text{mul}} \; + \; \textbf{error}(\mathcal{G}_\ell) \cdot \textbf{error}(\mathcal{G}_r) \\ & + \textbf{error}(\mathcal{G}_\ell) \cdot \textbf{value}(\mathcal{G}_r) \\ & + \textbf{value}(\mathcal{G}_\ell) \cdot \textbf{error}(\mathcal{G}_r).\end{aligned} \tag{1}$$

If the truncated result of the multiplication of v_1 and v_2 is in the format $\mathcal{Q}_{i_r,f_r}$, we have in signed arithmetic:

$$\textbf{error}_{\text{mul}} = [-\epsilon, 0] \quad \text{with} \quad \epsilon = 2^{-f_r} - 2^{-(f_1+f_2)}.$$

This can be seen as a rounding downward, that is, toward infinity, since the computed value is always no greater than the mathematical value.

Recall that if the operation is a multiplication by a power of 2, we replace it by a virtual shift, which is cost-free, but also error-free and $\textbf{error}_{\text{mul}} = [0, 0]$. Hence, assuming that the right operand $\mathcal{G}_r$ is 2^p, the error for the node $\mathcal{G}$ depends only on the error of its left operand $\mathcal{G}_\ell$. More particularly:

$$\textbf{value}(\mathcal{G}_r) = [2^p, 2^p] \quad \text{and} \quad \textbf{error}(\mathcal{G}_r) = [0, 0],$$

and from (1), it follows:

$$\textbf{error}(\mathcal{G}) = \textbf{error}(\mathcal{G}_\ell) \cdot 2^p.$$

3) Left and right shifts: Since we assume that no overflow occurs, the left shift is error-free as it just increases the size of the fractional part: thus $\textbf{error}_{\text{shift}} = [0, 0]$. Concerning the right shift, the error can be deduced in a way similar to the multiplication. As mentioned above, shifting the left child $\mathcal{G}_\ell$ of $\mathcal{G}$, representing the value v_1, of r bits to the right results in a value in the format $\mathcal{Q}_{i_r,f_r}$. Hence:

$$\textbf{error}_{\text{shift}} = [-\epsilon, 0] \quad \text{with} \quad \epsilon = 2^{-f_r} - 2^{-f_1}.$$

and we have:

$$\textbf{error}(\mathcal{G}) = \textbf{error}(\mathcal{G}_\ell) + \textbf{error}_{\text{shift}}.$$

Remark that these bounds may be pessimistic in some cases. However if we can determine that, for a given node $\mathcal{G}$, $\textbf{value}(\mathcal{G}_\ell)$ or $\textbf{value}(\mathcal{G}_r)$ are point intervals, we can compute a tighter error interval for the node $\mathcal{G}$. For example, if the value of the left operand of a right shift is a point interval, the error on that shift can be simply computed as follows:

$$\textbf{error}(\mathcal{G}) = \textbf{value}(\mathcal{G}_\ell) - \textbf{value}(\mathcal{G}).$$

D. How to use fixed-point arithmetic in practice?

All the previous arithmetic rules were given in a general fashion, without any constraints on the way used to implement them. In hardware, for example when dealing with FPGA-like architectures, we can choose the total length of each intermediate variable of an implementation, and these rules can thus be applied directly.

In our context, we target C software implementations of fixed-point arithmetic. Hence, the length of our variables has to fit both the format lengths available in the C language, and the hardware available on the architecture. Therefore, for any variable of a given program in the format $\mathcal{Q}_{i,f}$, we will typically have $i + f = k$ with $k \in \{8, 16, 32, 64\}$. In practice we mostly used $k = 32$.

IV. Target-dependent programs synthesis

This section presents our approach based on instruction selection. It starts by some background and motivation before presenting the formalism we propose to describe architecture instructions. Then it details this approach, allowing us to synthesize target-dependent codes optimized for different criteria.

A. Definition and motivation

Instruction selection is a well-known process in compilation theory [18, §8.9]. Given a set of instructions and an intermediate representation of an expression, possibly a DAG, an *instruction selection algorithm*, also called a *tiling algorithm*, produces a subset of instructions necessary to evaluate this expression. To optimize the generated code with respect to some criterion, a cost may be associated to each considered instruction, allowing thus to define a cost function used to evaluate the cost of a DAG. It follows that a good selection is one that minimizes this cost function. Although this problem was proven to be NP-complete, even for simple machines in the case of DAGs [19], various algorithms that perform well in practice have been designed to tackle this problem. Our technique is inspired by the NOLTIS algorithm [19] and more generally by bottom-up rewrite systems that deal with instruction selection on DAGs. It is presented in Section IV-C below.

This work on instruction selection was primarily motivated by the large number of patterns `(a * b) + c` appearing in polynomial evaluation, as well as shift operations induced by the use of fixed-point arithmetic. Indeed most of the modern architectures are shipped with advanced instructions, allowing to fuse at least two operations in a single one. Cite for example the fused multiply-add (FMA), computing `(a * b) + c` in one instruction and with only one final rounding, and now required by the IEEE 754-2008 standard [20] in floating-point processors. When dealing with integer arithmetic, let us cite, the `mulacc` instructions, available on certain architectures, like ARM processors [21], and that computes `(a * b) + c` in a single instruction. Furthermore, CGPE initially targeted the ST231 [22], a processor of the ST200 core family. It is shipped with a shift-and-add instruction performing a left shift of 1 up to 4 positions followed by an addition, that is, the pattern `(a << b) + c` with `b` $\in \{1, \cdots, 4\}$.

This approach is highly flexible since it relieves us from writing a new front-end algorithm each time new instructions

are targeted. Note also that this work was a valuable occasion to test for new patterns, even ones that were not available as instructions on any hardware target. This may also help us in giving feedbacks to processor architects. For instance, one of the first conclusions we drew was that a shift-and-add operation based on right shift would have been quite valuable. Indeed, on the ST231, the shift-and-add instruction (with left shift) may be quite useful when dealing with signed fixed-point arithmetic, since it may appear to remove redundant sign bits [14, §5.1.1]. However, in practice, dealing with fixed point arithmetic induced a lot of shifts to align operands before addition. Most of these are right shifts that cannot be embraced into such a shift-and-add operation.

B. Architecture description

For the sake of modularity, we chose a structure where the core of the tool is not aware of the instructions available for use. In our context, an instruction may be either a hardware instruction or a basic block of instructions. Our architecture is described in an external XML file, that contains all the instructions that may be used. Listing 2 gives an example of an entry for the $32\times32\rightarrow32$-bit unsigned addition.

```
1  <instruction
2    name="add"
3    type="unsigned"
4    inputs="32 32"
5    output="32"
6    latency="1"
7    macro="static uint32_t __name__(uint32_t a, uint32_t b)
8           {
9             return (a + b);
10          }"
11   gappa="_r_ fixed<-_Fr_,dn>= _1_ + _2_;
12          _Mr_ = _M1_ + _M2_;"
13   nodes="add dag 1 dag 2"
14 />
```

Listing 2. Example of an entry of the architecture file.

Precisely, for each instruction entry, this file contains the following parameters: the *name*, its *type*, that is, signed or unsigned, the size of its *inputs* and *output*, and its *latency* in cycles. If the entry matches an instruction available in hardware, this is its latency on the target. Otherwise, if this represents a basic block, we consider that this is its latency on unbounded parallelism, that is, without any resource constraint. In addition to these parameters, the description contains a C macro, allowing us to emulate the instruction in software if it is not available on the target, and a piece of Gappa script. The latter appears when computing the error entailed by the evaluation of the instruction in fixed-point arithmetic. The semantic used is that of Gappa. In Listing 2, line 11 returns the value `_r_` computed in fixed-point arithmetic in the program. In this example, it corresponds to the computed value `_1_ + _2_` (addition between the first and second parameter of the instruction) rounded downward (`dn`) with `_Fr_` fractional bits. Line 12 returns the mathematical value `_Mr_ = _M1_ + _M2_`, computed as if all the previous computations had been done exactly. Once these two values are computed, we are able to deduce the error entailed by the evaluation of this instruction by simply subtracting `_Mr_` from `_r_`. Finally the attribute *nodes* gives the description of the pattern matched by the instruction in terms of atomic operations. The entry of Listing 2 is solely composed of an addition, whose children are both any DAGs and corresponding to the first and second parameters of the instruction, respectively. The node description can be used to match any binary tree of depth at most 4 (a self imposed limit to ease the step). It is determined by traversing this tree in left-to-right breadth first order. For instance, the shift-and-add instruction of the ST231 is represented as follows.

```
nodes="add shift dag 2 dag 1 value [-4,-1]"
```

Here "`value [-4, -1]`" corresponds to the right child of the shift operator, and indicates that it is a numerical value in $[-4,-1]$. Since a negative value means that it is a left shift, thus it is a left shift of 1 to 4 positions.

C. Filter based on instruction selection

As mentioned earlier, once a fast scheme is computed, it undergoes a series of filters. Instruction selection intervenes as a filter : it produces a tiling of the DAG that minimizes a cost function while making the best of the targeted architecture. Depending on the underlying cost function used for the selection of instructions, this filter may generate schemes that are optimized for different criteria.

We have adapted the NOLTIS algorithm (Near-Optimal Linear Time Instruction Selection) introduced by Koes and Goldstein [19]. It appeared to be well-suited for our context, since it allows to tackle the problem of instruction selection on DAGs. NOLTIS proceeds in three major steps:

1) The first step traverses the DAG and it assigns to each node the instruction that minimizes the cost function.
2) The second step handles the case of nodes that are covered by more than a tile: either it leaves the tiles as they are, or cut the sub-DAG rooted at this shared node and mark it as resolved for the rest of the algorithm.
3) The last step consists in another round of instruction selection. It differs from the first step only since it does not try to tile sub-DAGs marked as resolved by step 2.

First let us remark that the second step of NOLTIS may lead to an increase in the the number of operators in the synthesized codes. Second, as shown for instance in [1, § 7.1.1], minimizing the evaluation latency on unbounded parallelism relies on minimizing the maximum of this latency on all operands. Thus no improvement can be expected by running step 2. Finally this step might be used only for marking shared nodes, that is, nodes computing powers of x. Assuming that an operation fusing several multiplications is at least as accurate as the combination of the single multiplication instructions, for accuracy purpose, this step seems to be useless. For all these reasons, we have mainly benefited from the first part of NOLTIS, even if the second part is necessary to correctly synthesize codes on architectures providing instructions like the one computing `(a * b) * c`. Hence we present an adaptation of NOLTIS based on its first step only. This is mainly what is done to tile trees, which is easier than tiling DAGs since there is no shared node handling.

Formally let $\mathcal{G}$ be a node of the DAG and $\mathcal{T}$ be the set of tiles that match this node and that can be used to evaluate it.

For a given tile $t \in \mathcal{T}$, children(t) denotes the set of nodes of the DAG being children of t. The sequel of this section presents the three cost functions we have implemented.

1) Operator count: The original NOLTIS algorithm relies on a cost function yielding the code size optimization. First this can be easily adapted to optimize the number of instructions in the output code. Hence the minimal operator count $\mathcal{C}(\mathcal{G})$ is:

$$\mathcal{C}(\mathcal{G}) = \min_{t \in \mathcal{T}} \Big(\mathcal{C}_t(\mathcal{G}) \Big) \quad \text{with} \quad \mathcal{C}_t(\mathcal{G}) = 1 + \sum_{n \,\in\, \text{children}(t)} \mathcal{C}(n).$$

2) Evaluation latency: Our second adaptation of NOLTIS aims at optimizing the evaluation latency on unbounded parallelism. Reducing the evaluation latency of $\mathcal{G}$ relies on the reduction of the maximum latency of its children. Hence the minimal evaluation latency $\mathcal{L}(\mathcal{G})$ is as follows:

$$\mathcal{L}(\mathcal{G}) = \min_{t \in \mathcal{T}} \Big(\mathcal{L}_t(\mathcal{G}) \Big)$$

$$\text{with} \quad \mathcal{L}_t(\mathcal{G}) = \text{latency}(t) + \max_{n \,\in\, \text{children}(t)} \mathcal{L}(n).$$

3) Accuracy optimization: Our third adaptation aims at reducing the evaluation error. As shown in Section III with our fixed-point arithmetic model, reducing this error on both children of $\mathcal{G}$ yields the reduction of the error entailed by the evaluation of $\mathcal{G}$. Our accuracy optimization is mainly based on this remark. Hence to decide which tile t leads to the tightest evaluation error, we generate the Gappa script for the sub-DAG rooted at $\mathcal{G}$, and for each tile $t \in \mathcal{T}$, we compute the evaluation error $\mathcal{E}_t(\mathcal{G})$ to keep the best one.

V. Experimental results

This section presents some experimental results in order to validate our approach.

A. Impact on the number of operations

In this first example, we consider a set of polynomials of degree 5 up to 12 that approximate the functions $\cos(x)$ and $\sin(x)$ over $[-1, 1]$, and $\log_2(1 + x)$ over $[-0.5, 0.5]$. These polynomials were computed using the `fpminimax` function of the software tool Sollya.[6] For each polynomial of each degree, we have synthesized a set of 50 programs, each of them being optimized for the use of a particular instruction among the following: an `add-add` computing `(a + b) + c`, a `mulacc` computing `(a * b) + c`, a `shift-and-add` left and right computing `(a << b) + c` and `(a >> b) + c`, respectively, with `b` $\in [1, 4]$. For each function and each advanced instruction below, Figure 5 shows the average number of operations in the output code. This figure also shows the number of operations when no advanced instruction is used as well as when all of them are available for use.

From these results, we can observe that thanks to our technique, we reduce the number of instructions in the generated codes of 13.85 % up to 22.3 % depending on the function when all the advanced instructions are considered, and of 8.3 % up to 11.7 % otherwise. For example, the evaluation of the $\sin(x)$ function illustrates the interest of the shift-and-add operator

[6]See http://sollya.gforge.inria.fr and [23].

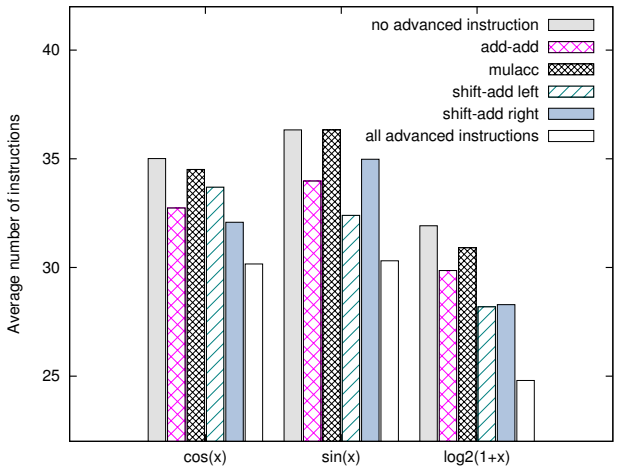


Figure 5. Average number of instructions in the synthesized codes, for the evaluation of polynomials of degree 5 up to 12 for various elementary functions.

available on ST200 family architectures, since, in this case, it is the most valuable operator and leads to a gain of 10.8 %.

As mentioned in Section IV-A, an operator similar to the ST231's shift-and-add, but with a right shift instead of a left shift, would be of great use for evaluating polynomials in signed fixed-point arithmetic. Indeed, in the examples of $\cos(x)$ and $\log_2(1 + x)$ in Figure 5, this is the most relevant instruction since it leads to a reduction of the operation count of 8.4 % and 11.4 %, respectively.

B. Evaluation latency reduction for $\cos(x)$

On this second example, let us focus on a degree-7 polynomial approximating the function $\cos(x)$ over $[0, 2]$. Here we consider 1-cycle addition, subtraction, and shift-and-add operator (as specified in the ST231), and 3-cycle multiplication and `mulacc` operation. We have synthesized codes to

	Without tiling	With tiling	Speed-up
Horner's rule	41	34	≈ 17.07 %
Estrin's rule	16	14	≈ 12.5 %
Best scheme generated by CGPE	15	13	≈ 13.33 %

Table I. Latency in # cycles on unbounded parallelism, for various schemes, with and without tiling.

evaluate $\cos(x)$ using Horner's and Estrin's rules, two classical evaluation schemes, as well as code generated automatically by CGPE Table I gives the latency on unbounded parallelism of these codes, before and after tiling. We can observe that the code after tiling has a lower latency than before tiling, and that the speed-up may be up to ≈ 17 % for Horner's rule.

Obviously, it is of great interest to provide hardware support for instructions fusing several operations, and having a lower latency than the sum of the latencies of all fused operations.

C. Accuracy optimization for a degree-3 polynomial

Now let $a(x)$ be a degree-3 polynomial approximant of $\exp(x)$ over $[0, 1]$, evaluated using Estrin's rule:

$$a(x) = (a_0 + a_1 \cdot x) + (x \cdot x) \cdot (a_2 + a_3 \cdot x),$$

with $a_0 = 4192309 \cdot 2^{-22}$, $a_1 = 65 \cdot 2^{-6}$, $a_2 = 6947 \cdot 2^{-14}$, and $a_3 = 18255 \cdot 2^{-16}$, and in the fixed-point format $\mathcal{Q}_{8,24}$, $\mathcal{Q}_{24,8}$, $\mathcal{Q}_{16,16}$, and $\mathcal{Q}_{16,16}$, respectively. We still consider 1-cycle addition and subtraction and 3-cycle multiplication. Listing 3 shows the fixed-point code evaluating this polynomial in 9 cycles on unbounded parallelism, with a certified evaluation error bound of $\approx 2^{-5.44}$. In this third example, we define a basic

```
uint32_t func_0(uint32_t x) {          //  (+)Q[1.31]
  uint32_t r0  = 0x00ffe0d4 >> 17;     //  (+)Q[25.7]
  uint32_t r1  = mul(x, 0x00000104);   //  (+)Q[25.7]
  uint32_t r2  = r0 + r1;              //  (+)Q[25.7]
  uint32_t r3  = mul(x, x);            //  (+)Q[2.30]
  uint32_t r4  = 0x00006c8c >> 1;      //  (+)Q[17.15]
  uint32_t r5  = mul(x, 0x0000474f);   //  (+)Q[17.15]
  uint32_t r6  = r4 + r5;              //  (+)Q[17.15]
  uint32_t r7  = mul(r3, r6);          //  (+)Q[19.13]
  uint32_t r8  = r7 >> 6;              //  (+)Q[25.7]
  uint32_t r9  = r2 + r8;              //  (+)Q[25.7]
  return r9;
}
```

Listing 3. Synthesized code to evaluate $\exp(x)$.

block called `fx_fma` and behaving in fixed-point arithmetic like FMA in floating-point arithmetic, and computing `(a * b) + (c >> n)` (with `n` $\in \{0, \cdots, 31\}$) with only one final truncation. The code of Listing 3 could be evaluated using 3 `fx_fma`. Using our technique based on selection for accuracy improvement, we find automatically that fusing the first three operations (`r0`, `r1`, and `r2`) into a `fx_fma` yields an evaluation error bound of $\approx 2^{-5.99}$, that is, a code half a bit more accurate, which can be quite useful in some contexts.

VI. Conclusion and future work

This paper extends the CGPE software tool, dealing with automated synthesis of fast and accurate programs for evaluating bivariate polynomials in fixed-point arithmetic. First, we adapt the fixed-point arithmetic model of CGPE by providing a support for signed arithmetic. Thus, we are now able to handle polynomials of degree up to 20 in few minutes, and without any constraint on the sign of the input and output. In addition, we propose a new filter based on instruction selection, allowing us to synthesize codes efficiently for a given target and optimized for different criteria, like operator count, evaluation latency, or numerical accuracy. Regarding operator count, on some examples, this leads to a reduction of up to 22.3 % of the number of operations in the synthesized codes.

The future work direction is threefold: First we will extend the instruction selection based on a single criterion to a multi-criteria selection. Indeed as a natural extension, we could define a cost for a node $\mathcal{G}$ as a linear combination of the three errors defined Section IV-C, that is, with $a, b, c \in \mathbb{R}^+$:

$$\mathcal{C}ost(\mathcal{G}) = \min_{t \in \mathcal{T}} \Big(a \cdot \mathcal{C}_t(\mathcal{G}) + b \cdot \mathcal{L}_t(\mathcal{G}) + c \cdot \mathcal{E}_t(\mathcal{G}) \Big).$$

Second, all this work has been done on unbounded parallelism. A further direction will consist in tweaking the scheduler of CGPE and our architecture formalism, to check if our synthesized codes remain efficient under resource constraints. Third we will extend this tool to tackle other problems than polynomial evaluation. This simply relies on the design of new DAG computation algorithms, while all the work on instruction selection can be used directly by just plugging these new algorithms on the top of the middle-end of CGPE.

References

[1] C. Mouilleron, "Efficient computation with structured matrices and arithmetic expressions," Ph.D. dissertation, Univ. de Lyon - ENS de Lyon, 2011.

[2] M. Püschel, F. Franchetti, and Y. Voronenko, *Encyclopedia of Parallel Computing*, 1st ed. Springer, 2011, ch. Spiral.

[3] B. Lopez, T. Hilaire, and L.-S. Didier, "Sum-of-products evaluation schemes with fixed-point arithmetic, and their application to IIR filter implementation," in *Conf. on Design and Architectures for Signal and Image Processing (DASIP)*, 2012.

[4] M. Martel, "Enhancing the Implementation of Mathematical Formulas for Fixed-Point and Floating-Point Arithmetics," in *Journal of Formal Methods in System Design*, vol. 35. Springer, 2009, pp. 265–278.

[5] M. Martel, "Program transformation for numerical precision," in *PEPM'09*. ACM Press, 2009.

[6] D. Menard, D. Chillet, and O. Sentieys, "Floating-to-fixed-point conversion for digital signal processors," in *EURASIP Journal on Applied Signal Processing*, 2006, pp. 1–15.

[7] D.-U. Lee and J. D. Villasenor, "Optimized Custom Precision Function Evaluation for Embedded Processors," *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 46–59, 2009.

[8] S. Boldo, "Preuves formelles en arithmétiques à virgule flottante," Ph.D. dissertation, École Normale Supérieure de Lyon, 2004.

[9] J. Harrison, T. Kubaska, S. Story, and P. Tang, "The computation of transcendental functions on the IA-64 architecture," *Intel Technology Journal*, vol. 1999-Q4, pp. 1–7, 1999.

[10] R. Green, "Faster Math Functions." *Tutorial at Game Developers Conference*, 2002.

[11] C. Mouilleron and G. Revy, "Automatic Generation of Fast and Certified Code for Polynomial Evaluation," in *Proc. of the 20th IEEE Symposium on Computer Arithmetic (ARITH'20)*, Tuebingen, Germany, 2011.

[12] Y. Voronenko and M. Püschel, "Automatic generation of implementations for DSP transforms on fused multiply-add architectures," in *International Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 5, 2004, pp. V–101.

[13] Y. Voronenko and M. Püschel, "Mechanical derivation of fused multiply-add algorithms for linear transforms," *IEEE Transactions on Signal Processing*, vol. 55, no. 9, pp. 4458–4473, 2007.

[14] G. Revy, "Implementation of binary floating-point arithmetic on embedded integer processors - polynomial evaluation-based algorithms and certified code generation," Ph.D. dissertation, Univ. de Lyon - ENS Lyon, 2009.

[15] G. Melquiond, "De l'arithmétique d'intervalles à la certification de programmes," Ph.D. dissertation, ÉNS Lyon, 2006.

[16] R. Yates, *Fixed-Point Arithmetic: An Introduction*, Digital Signal Labs, 2009.

[17] N. Revol and F. Rouillier, "Motivations for an arbitrary precision interval arithmetic and the MPFI library," *Reliable Computing*, vol. 11, no. 4, pp. 275–290, 2005.

[18] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques and tools*. Addison Wesley, 1986.

[19] D. R. Koes and S. C. Goldstein, "Near-optimal instruction selection on DAGs," in *International Symposium on Code Generation and Optimization (CGO'08)*. Washington, DC, USA: IEEE Computer Society, 2008.

[20] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, 2008.

[21] *ARM Architecture Reference Manual*, 2009.

[22] *ST231 core and instruction set architecture – Reference manual*, 2008.

[23] C. Lauter, "Arrondi correct de fonctions mathématiques - fonctions univariées et bivariées, certification et automatisation," Ph.D. dissertation, Univ. de Lyon - ÉNS Lyon, 2008.