Automatic Generation of Fast and Certified Code for Polynomial Evaluation

Christophe Mouilleron, Guillaume Revy

Automatic Generation of Fast and Certified Code for Polynomial Evaluation

Christophe Mouilleron[1, 2] Guillaume Revy[3]

[1]ENS de Lyon [2]Université de Lyon [3]Université de Perpignan Via Domitia

Laboratoire LIP, École normale supérieure de Lyon — 46 allée d'Italie, 69364 Lyon cedex 07, France

Laboratoire ELIAUS-DALI, Université de Perpignan Via Domitia — 52 avenue Paul Alduy, 66860 Perpignan cedex 9, France

Abstract

Designing an efficient floating-point implementation of a function based on polynomial evaluation requires being able to find an accurate enough evaluation code, exploiting at most the target architecture features. This article introduces CGPE, a tool dealing with the generation of fast and certified codes for the evaluation of bivariate polynomials. First we discuss the issue underlying the evaluation scheme combinatorics before giving an overview of the CGPE tool. The approach we propose consists in two steps: the generation of evaluation schemes by using some heuristics so as to quickly find some of low latency; and the selection that mainly consists in automatically checking their scheduling on the given target and validating their accuracy. Then, we present on-going development and ideas for possible improvements of the whole process. Finally, we illustrate the use of CGPE on some examples, and show how it allows us to generate fast and certified codes in a few seconds and thus to reduce the development time of libms like FLIP.

Keywords: *polynomial evaluation schemes, code generation, automatic accuracy certification, floating-point implementation.*

1 Introduction

The floating-point implementation of a function in software often relies on the evaluation of an accurate enough polynomial that approximates this function on a small interval. In that case, this evaluation remains usually the most expensive part of the whole implementation, and the key point is to make it as efficient as possible, while being accurate enough. The development time for such hand-written implementations may be quite long [15, p.197], tedious and error-prone. Also, a new implementation has to be designed each time a new target comes out or a new format is required. Hence, it is highly desirable to automate and certify this process. So today's challenge is to design methodologies and tools to help in automatically writing efficient and accurate floating-point function implementations. We can

cite several generators, like FloPoCo[1] for hardware or Sollya[2] and Metalibm[3] for software. The SPIRAL project[4] also aims at generating fast codes (in both hardware and software) for DSP algorithms. Since the generation process can usually be described as a sequence of very distinct specific tasks, another approach, embraced by the LEMA project [17], is to develop some language, expressive enough to cover all the process, as well as a library with support for many external dedicated tools, thus enabling to design easily an appropriate toolchain. Finally, one could also start from some existing code and try to improve it. For instance, [18] discusses code transformation for increasing the numerical accuracy of a floating-point computation, and [19] extends it to minimize the number of bits needed for the integer part in a computation with fixed-point arithmetic.

This work takes mainly part in the context of the development of the library FLIP,[5] a floating-point operator library optimized for the ST231, a 4-issue 32-bit VLIW integer processor, and where some operators are implemented using fast as well as accurate enough polynomial evaluation, for ensuring correct rounding of the underlying implementation, in the sense of the IEEE 754-2008 standard [9, § 2.1]. Therefore, the main goal of the tool presented here, called CGPE (for Code Generation for Polynomial Evaluation), is to automate the design of fast and certified code for the evaluation of polynomial in fixed-point arithmetic. Hence, unlike what is done within the SPIRAL project for example, we do not focus on the algorithm speed only, but also on their accuracy, by adding a systematic certified numerical analysis phase to the generation process.

The motivation for such tools is first to speed up both polynomial evaluation codes and their design, and provide certificates on their speed and numerical accuracy. Also, it would make possible to explore quickly a significant part of the space of the evaluation schemes, to study compromises between speed and accuracy for various targets, as in [14].

The main contributions of this work are first an algorithm for generating all the possible schemes for evaluating bivariate polynomials using only additions and multiplications, second some heuristics to speed up the search for some schemes reducing evaluation latency on unbounded parallelism, and third a set of filters for selecting schemes satisfying some given criteria: speed on the target architecture and numerical accuracy.

This article is organized as follows. After some background on polynomial evaluation and its combinatorics in Section 2, the software tool CGPE is presented in Section 3, and some improvements are discussed in Section 4. Then, some experimental results are reported in Section 5, before concluding in Section 6.

2 Background

In the 60's, multiplication being much slower than addition [12], some evaluation schemes have been built, for reducing the number of involved multiplications. We can cite, for example, Knuth and Eve [4, 13] or Paterson and Stockmeyer [24] algorithms. These methods are based on the *precomputation* of new coefficients done once before all the evaluations. Nevertheless, they remain ill-adapted for our context (fixed-point arithmetic), since we may lose too much accuracy during the precomputation phase.

[1] See `http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/`.

[2] See `http://sollya.gforge.inria.fr/` and [15].

[3] See `http://lipforge.ens-lyon.fr/www/metalibm/`.

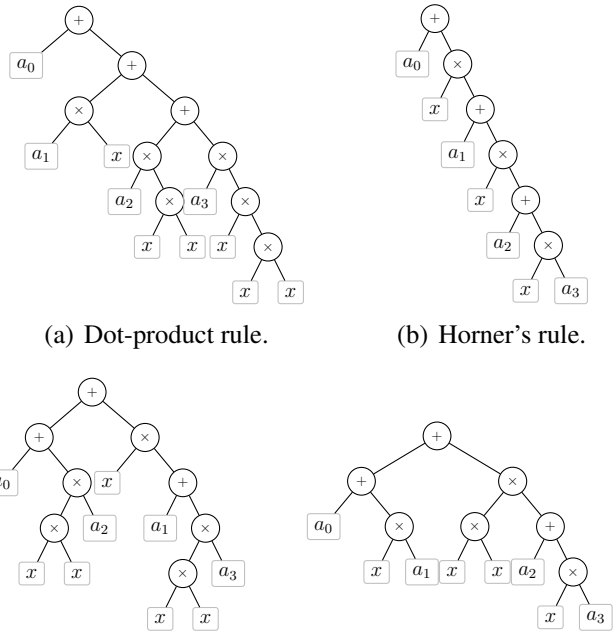
[4] See `http://www.spiral.net/` and [26].

[5] See `http://flip.gforge.inria.fr/` and [11].

So, hereafter, let us focus on methods based on parenthesization modifications. Even in that framework, various schemes may be used for evaluating polynomials. This section is organized as follows. First, in Section 2.1, we recall some classical schemes for evaluating univariate polynomials, that can be extended to bivariate polynomials. Then, in Section 2.2, we formally define what we call *evaluation scheme*, before giving some elements on the combinatorics of such schemes in Section 2.3.

2.1 Classical evaluation schemes

Let $a(x)$ be a univariate degree-n polynomial. The evaluation of $a(x)$ requires exactly n additions, whatever the scheme in use. Hence, in the following of this section, we focus on the number of required multiplications. Let us now present some classical evaluation schemes.



(a) Dot-product rule. (b) Horner's rule.

(c) Second-order Horner's rule. (d) Estrin's rule.

Figure 1. Classical rules for degree-3 univariate polynomial evaluation.

Dot-product rule. One of the most naive ways for evaluating $a(x)$ consists in computing each power x^i, evaluating each monomial $a_i \cdot x^i$, and adding all these terms with n additions. This first scheme is shown in Figure 1(a) for $n = 3$. Even if an efficient way is used for computing the x_i's, this would require exactly $2n - 1$ multiplications, and would be inefficient for implementing fast polynomial evaluation.

Horner's rule. This is one of the most commonly used schemes for evaluating polynomial in operator floating-point implementation. Its interest lies in its good numerical stability, especially when x is not too close to a zero of $a(x)$ [1]. It consists in n multiplications and n additions as shown in Figure 1(b), and is uniquely optimal in term of the number of multiplications involved [23]. However, this sequential scheme does not expose any instruction-level parallelism (ILP) and thus gets inefficient as soon as parallelism is available.

Second-order Horner's rule. This third rule extends Horner's rule in order to expose some ILP. It consists in splitting up $a(x)$ into its odd and even parts, evaluating both parts using Horner's rule, and finally combining both intermediate results using a last Horner's iteration [13, § 4.6.4]. It requires exactly $n+1$ multiplications, as shown in Figure 1(c). Remark that it uses at most two ways of the architecture, and gets inefficient as soon as more parallelism is available.

Estrin's rule. This last rule is based on the divide-and-conquer paradigm, and consists in splitting up $a(x)$ into its low and high parts. Then, both parts are evaluated in a recursive way, until getting degree-1 polynomials, as shown in Figure 1(d). Its implementation tends to expose more ILP than the previous rules, but to the detriment of an increase of the number of multiplications, since it requires about $n + \log(n+1) - 1$ multiplications.

All these schemes can be adapted for evaluating bivariate polynomials, as shown in [2, 25] for Horner's rule. The problem is now to find some efficient polynomial evaluation schemes, depending on some architectural constraints, like the number of ways or the nature of the available operators.

2.2 Formal definition of evaluation scheme

Our goal is to evaluate univariate or bivariate polynomials using only additions and multiplications (eventually replaced with squarings or shift operations, depending on the operands). Thereafter, let us call *expression* the mathematical object corresponding to the polynomial $a(x, y)$ to be evaluated. A *subexpression* of $a(x, y)$ will then be any mathematical polynomial $q(x, y)$ such that $a(x, y) = r(x, y) + x^i \cdot y^j \cdot q(x, y)$ for some polynomial $r(x, y)$ and some integers i, j. Intuitively, the subexpressions are all the polynomials that may appear when evaluating $a(x, y)$.

By fixing some implicit rules, like precedence of $\times$ over $+$ plus left-to-right parenthesization, we can deduce one *parenthesization* for a given expression. All the *parenthesizations* are then obtained by applying one or more of the following mathematical properties of operators $+$ and $\times$: commutativity of $+$ and $\times$, associativity of $+$ and $\times$, and distributivity of $\times$ over $+$ and factorization. The latter will be of great use as it is the one responsible for the increase of parallelism. All these parenthesizations can be represented as ordered binary trees (like the ones in Figure 1), and correspond to various mathematically equivalent ways to perform the evaluation of our initial expression with binary additions and multiplications.

Computation will be carried out using a standard fixed-point or floating-point arithmetic so that only commutativity for $+$ and $\times$ still holds [21, §2.4]. Hence we define the set of *evaluation schemes* as the equivalence classes of the parenthesizations modulo commutativity (i.e. modulo swaps of sons in our trees). Thus, the set of evaluation schemes represents all the potentially numerically distinct implementations of a given expression. For instance, for a univariate polynomial of degree 2, we can find 208 parenthesizations from which we get the 7 following evaluation schemes:

$$\begin{array}{ll} (a_0 + (a_1 \cdot x)) + (a_2 \cdot (x \cdot x)) & (a_0 + (a_1 \cdot x)) + ((a_2 \cdot x) \cdot x)) \\ (a_0 + (a_2 \cdot (x \cdot x))) + (a_1 \cdot x) & (a_0 + ((a_2 \cdot x) \cdot x)) + (a_1 \cdot x) \\ a_0 + ((a_1 \cdot x) + (a_2 \cdot (x \cdot x))) & a_0 + ((a_1 \cdot x) + ((a_2 \cdot x) \cdot x))) \\ a_0 + (a_1 + (a_2 \cdot x)) \cdot x & \end{array}$$

2.3 Combinatorics of evaluation schemes

The number of evaluation schemes has been studied for several classes of arithmetic expressions. The most complete results have been obtained for the two following cases:

- the sum $x_1 + \cdots + x_n$ of n variables, for which the number of evaluation schemes is exactly[6]

$$(2n-3)!! = \prod_{i=1}^{n-2} (2i+1),$$

- the power x^n, for which the number of evaluation schemes is the nth Wedderburn-Etherington number.[7] When n tends to infinity, the asymptotic equivalent is

$$\frac{\eta\,\xi^n}{n^{3/2}}, \quad \text{with} \quad \begin{cases} \xi \approx 2.48325 \\ \eta \approx 0.31877 \end{cases} \quad \text{(see [22] or [5, §5.6]).}$$

These two cases are included in our problem of polynomial evaluation. Indeed, one way to evaluate a univariate polynomial is to proceed like in the dot-product scheme mentioned above: First compute x^i for $2 \leq i \leq n$, then perform all the multiplications $a_i \cdot x^i$. The remaining step can finally be seen as a sum of $n+1$ variables. As both of theses subcases are at least exponential with respect to n, we could expect a quite huge number of schemes for univariate polynomials, and even worse for the special bivariate polynomials $a(x,y) = \alpha + y \cdot p(x)$ we are interested in.

We have computed the number of schemes for univariate polynomials[8] of degree n, and for special bivariate polynomials.[9] We have found 1304066578 schemes for a univariate degree-6 polynomial, and 122657263474 for $a(x,y)$ when $\deg p! = 5$. Therefore, aggressive heuristics will be necessary to tackle problems with degrees higher than 5.

3 The CGPE tool

This section presents CGPE[10] (standing for Code Generation for Polynomial Evaluation), the tool we have implemented for automatically writing fast and certified C codes for evaluating univariate and bivariate polynomials in fixed-point arithmetic by using as much as possible the features of the target. Hereafter, by fast, we mean that reduces the evaluation latency on a given target, while by certified we mean that we can bound the error entailed by its evaluation.

Given a polynomial, we have seen so far that different evaluation schemes may be used for its evaluation, exposing more or less ILP. Recall that we want to compute schemes using only additions and multiplications (or shifts, on integer architectures), and without precomputing any new coefficients. Hence it turns out that decreasing the evaluation latency on unbounded parallelism implies increasing the number of multiplications to expose much more ILP, the number of addition remaining the same.

After a reminder of some related works in Section 3.1, we describe precisely the input and output of our problem in Section 3.2, before giving a global description of CGPE in Section 3.3. Then, our heuristics are presented in Sections 3.4 and 3.5.

[6] `http://www.research.att.com/~njas/sequences/A001147`.

[7] `http://www.research.att.com/~njas/sequences/A001190`.

[8] `http://www.research.att.com/~njas/sequences/A169608`.

[9] `http://www.research.att.com/~njas/sequences/A173157`.

[10] Available at `http://gforge.inria.fr/projects/cgpe/`.

3.1 Related work

Some work has already been done about code generation for function implementation based on polynomial evaluation. For example in [3, 16], the implementation is done using Horner's rule. We have already seen that when parallelism is available, we can speed up the evaluation by using a scheme that exposes more ILP than Horner's rule. In [7], an approach is presented for generating optimal evaluation schemes for univariate polynomial on the Itanium® processor by using only the `fma` operator ; another proposal is done in [6] where a brute force approach (inspired from the previous one) is used for generating polynomial evaluation schemes using at best SIMD instructions, for the implementation of faster mathematical functions for the PlayStation®2.

In our context, we have only addition and multiplication, and no `fma`. Also, brute force method may not be well adapted since, in the long term, our goal is to generate schemes at compile-time: we cannot generate all the evaluation schemes and choose the one we want to keep according to the parameters, especially for high degree bivariate polynomial. Hence, we need some heuristics for getting rid of "bad" schemes as soon as possible during the generation.

3.2 Statement of our problem

Let us now detail the input and output of CGPE, as well as the architectural constraints to be considered.

Input of CGPE. CGPE takes as input a polynomial given by its support, that is, the list of its non-zero coefficients. For each coefficient, the user may provide a value, a fixed-point format and some information like being a power of 2, so that a multiplication by this coefficient on integer arithmetic can be replaced with a shift operation (usually less expensive). The user may also give an interval of values for each variable.

This work has been highly guided by the implementation of the library FLIP on the ST231, where some operator implementations are based on the evaluation of special bivariate polynomials. In this context, the actual value of one of the two variables is obtained a few cycles after the other [10]. This delay can also be given to CGPE.

Finally, CGPE takes a set of criteria to be achieved, like a maximum error bound for the evaluation, or a bound on latency (one can asks for the lowest latency as well).

Architectural constraints. For tuning the program to be efficient on a given target, CGPE has to know some architectural features, like the cost of each operators or the degree of parallelism (number of available issues). Our experiments have been done on the ST231, a 4-issue 32-bit VLIW integer processor, with only two $32 \times 32 \rightarrow 32$-bit multipliers. All the operations (addition, subtraction, and shift) have a latency of 1 cycle, but the multiplication, which has a latency of 3 cycles.

Output of CGPE. At the end of the process, CGPE produces a set of C codes that implement the evaluation of the given polynomial on the given architecture, and whose latency on this target satisfies the latency constraint. Also, CGPE attaches an accuracy certificate to each C code, which ensures that the evaluation error entailed by the program is less than the given maximum error bound.

3.3 Global architecture of CGPE

Code generation process using CGPE works in two steps. Its general architecture is shown in Figure 2(a). First, it computes a set of evaluation schemes for the polynomial given in input, and then it checks each schemes in order to keep only the ones satisfying both speed and accuracy constraints. We have seen so far that schemes can be represented with trees. Actually, CGPE manipulates DAGs (Directed Acyclic Graphs) for representing the computed evaluation schemes. This is mainly motivated by the fact that common subexpressions are thus not duplicated. Moreover, we do not need an explicit phase of common subexpression elimination before any treatment (like error bound computation). Indeed, during the depth-first traversal of a DAG, we can easily detect whether an operation has already been processed or not. In the following of this section, we briefly describe the two steps.

Computation of DAGs. At this point, we assume unbounded parallelism and consider only the costs of each available operator for being able to compute the latency of each DAG on unbounded parallelism. The algorithm we have implemented in CGPE, detailed in [28, §6.1], works as an iterative process that computes DAGs in a bottom-to-top way, starting with the coefficients and variables. At iteration i, it computes all the evaluation schemes for all the subexpressions of total degree i from those of subexpressions of degree less than i. Since the number of such DAGs is getting huge as soon as $n \geq 6$, we have implemented some heuristics, detailed in Section 3.4, to reduce the number of produced DAGs.

DAG selection. At this second point, we take into account all the characteristics of our problem that have been neglected during the first step, that is, bounded parallelism (depending on the parallelism available on the target architecture) and the behavior of each available operators (mainly for computing error bounds). In this step, we perform a succession of tests on each generated DAG in order to determine those which really are solution of our problem. These tests are presented in Section 3.5.

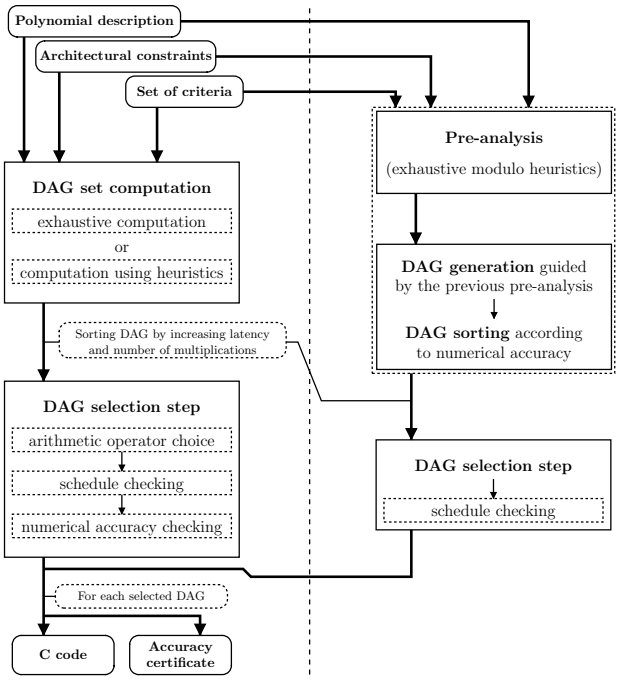
3.4 Heuristics in DAG set computation

As seen in Section 2.3, the number of evaluation schemes grows extremely fast with respect to the total degree n, so that exhaustive search cannot be performed as soon as $n \geq 6$. Therefore, we have implemented two heuristics to restrict the search space. The first heuristic consists in discarding, as soon as possible, DAGs that cannot fulfill the latency requirement on unbounded parallelism, while the second one aims at restricting the search space by considering only some specified subset of all the evaluation schemes.

Early elimination of DAGs. The goal of the first heuristic is to compute DAGs of low latency on unbounded parallelism. For doing this, we first compute a target latency, denoted by τ. The key point is then to decide all along the DAG set computation if the encountered DAGs can be used to evaluate the input polynomial with a latency at most τ. Thus, intermediate DAGs with a latency on unbounded parallelism greater than τ are discarded.

If at the end of the computation process we have not found any DAG fulfilling the speed requirement (on unbounded parallelism), we increase the target latency and restart the computation. Therefore, it is important to define a target latency as close as possible to the minimal latency: neither smaller to avoid running the process uselessly, nor larger to get only DAGs of merely optimal latency (at least on unbounded parallelism). We propose to start with:

$$\tau_{\text{static}} = \lceil \log_2(d_x + d_y + 1) \rceil \cdot \mathcal{C}_\times + \mathcal{C}_+,$$



(a) Current architecture. (b) Possible improvements.

Figure 2. Architecture of CGPE.

with $\mathcal{C}_+$ and $\mathcal{C}_\times$ the addition and multiplication costs respectively. In fact, τ_{static} corresponds to the best latency for evaluating $a_{0,0} + a_{d_x,d_y} \cdot x^{d_x} \cdot y^{d_y}$ on unbounded parallelism: a complete tree of products for the right part plus a final addition. Assuming $a_{0,0} \neq 0$ (it is always the case in our applications), $a_{0,0} + a_{d_x,d_y} \cdot x^{d_x} \cdot y^{d_y}$ is a subexpression of the polynomial $a(x, y)$ given in input. As we need at least to evaluate this subexpression in order to evaluate $a(x, y)$, τ_{static} is a lower bound of the latency for $a(x, y)$. Moreover, it is very close to the actual latency on unbounded parallelism, since it was chosen to be the best latency of the critical part of $a(x, y)$ (in terms of number of operations).

When the user provides a delay, meaning that the actual values for x and y are not available at the same time, τ_{static} is not so relevant anymore. In this case, we would rather use a dynamically computed target latency τ_{dynamic}, obtained by considering each way to evaluate the subexpression $a_{0,0} + a_{d_x,d_y} \cdot x^{d_x} \cdot y^{d_y}$ and finding out the best achievable latency on unbounded parallelism. This computation is a special case of Algorithm 1 presented thereafter in Section 4.1.

Optimized search. The cost for exhaustive DAG set computation is prohibitive, even if we discard DAGs with too high latency on unbounded parallelism like we did above. The main reason lies in the fact that a given polynomial has an exponential number of subexpressions with respect to the size of its support, all of these having to be considered during the DAG set computation. Hence, we have designed a recursive top-to-bottom procedure whose goal is to select only a specified part of the divide-and-conquer decompositions of the given polynomial. For instance, if we have a degree-n univariate polynomial $a(x) = \sum_{i=0}^{n} a_i\, x^i$, we will only consider the evaluations based on a factorization by some power of x or on a splitting into low and high parts:

$$a(x) = \left(a_0 \ +_1 \cdots +_{\text{i-1}} a_{i-1}\, x^{i-1}\right) +_{\text{i}} \left(a_i\, x^i +_{\text{i+1}} \cdots +_{\text{n}} \ a_n\, x^n\right),$$

with $1 \leq i \leq n$. This strategy is executed recursively until:

- either we have reached a given recursion depth,
- or the considered polynomial has a support (number of coefficients) no greater than a given parameter.

In these cases, exhaustive search is launched instead of this recursive-based approach. Typical choice for recursion depth is 2 or 3. As for the parameter limiting the size of the support, it has to be no greater than 5.

Despite the use of the two heuristics mentioned in this section, the number of schemes may still be too large, so we have added another parameter indicating the number of DAGs to be kept at each step of the process. Combined with the two previous heuristics, it allows us to quickly generate DAGs with a low latency, as we will see in Section 5.

3.5 Filters in the DAG selection step

Now that the DAG set computation step has produced several evaluation schemes, fast on unbounded parallelism, what remains is to check each and every DAG in order to determine whether the evaluation can be performed on the target architecture, and whether it is fast and accurate enough.

Arithmetic operator choice. This filter consists in first determining the fixed-point format of each intermediate variable, and verifying that no comma alignment is required, otherwise it may imply an increase of the evaluation latency of the considered DAG. Then it computes a certified enclosure of its evaluation value (denoted **value**, further in this article), and checks if the scheme represented by the DAG can be evaluated in fixed-point arithmetic using only intermediate variables of constant sign, so that we do not have to store any sign bit. The advantage is twofold.

- Assuming the sign is stored in 1 bit, if we multiply two such numbers, the sign will be represented with 2 bits: at the end of the evaluation these bits "lost" at each iterations may have a significant impact on the accuracy of the evaluation result.
- To compensate the side effect presented in the first point, we may shift the result of each multiplication to keep just one bit for representing the sign. But in that case, each multiplication will have a possible extra cost of 1 cycle (4 instead of 3 cycles on ST231, for example), that may lead to a sizeable growth of the evaluation latency.

This is mainly done using interval arithmetic rules with MPFI,[11] by scanning the DAG from bottom to top.

Schedule on a simplified model of the target architecture. The second filter consists in checking if the DAG can be evaluated on the target without any increase of latency compared to the latency on unbounded parallelism. To do that, we have implemented a scheduler based on the list-scheduling algorithm with backtracking: at each step, we have a list of operations that can be launched. According to the parallelism available on the target, the parameters of the problem (delays, ...), and the architectural constraints, we choose some of them and try to carry on the scheduling. If some operations remain

[11] See `http://gforge.inria.fr/projects/mpfi/` and [27].

after τ cycles, we might have taken a bad branch in the search, thus we go back to the previous state and choose other operations, that is another branch. This work has been mainly guided by the ST231 architecture, and today, this scheduler is parameterized by the number of issues and multipliers available, while the strategy for encoding instructions into bundles memains ST231 dependent.

In the worst case, we scan all the possible schedulings, but at the end, we know exactly whether the DAG can be scheduled or not.

Evaluation error bound checking. It remains now to select the evaluation schemes that satisfy the criterion of accuracy, that is, those for which the evaluation error is less than a given bound. This accuracy checking is done using Gappa,[12] which allows us to compute a certified evaluation error bound entailed by the execution of the program. We observe that this step using Gappa may be quite expensive and may be a bottleneck in the process. Hence, we are currently implementing an approach based on naive interval arithmetic using MPFI, which is intended to be used during the DAG set computation step so that it makes this filter useless. This new approach is presented further in Section 4.2.

4 Future development of CGPE

In Section 3, we have introduced the CGPE tool as it is in its current stable version. While this version already provides a complete framework for generating efficient codes given a polynomial and a set of criteria (latency and accuracy bounds, architectural constraints), there is still room for improvements. We are thus currently working on the next version of the tool. This section gives an overview of what we are planning out, as shown in Figure 2(b): First, we propose to add a new initial step that aims at quickly identifying a set of subexpressions worth considering during the DAG set computation step. Second, we aim at incorporating some accuracy checkings within the phase of early elimination of DAGs. Let us now detail these future improvements.

4.1 Guiding the DAG set computation

As we have seen in Section 3.4, one way to reduce the cost of the DAG set computation step is to limit the number of generated DAGs. The optimized search introduced before does this by restricting the number of decompositions considered for subexpressions. We propose here to perform a precomputation whose purpose is to find out the set of all the subexpressions leading to an optimal final DAG for a given criterion. Thus, it becomes possible to start the DAG set computation and drop on the fly all the evaluation schemes that do not belong to the set obtained by this precomputation. While this approach introduces some extra cost, it has the advantage of isolating optimal schemes with respect to the given criterion, so that the underlying restriction heavily relies on this criterion instead of being somewhat arbitrary as before.

Algorithm 1 illustrates this technique with latency on unbounded parallelism as a criterion. It is a divide-and-conquer function that computes the minimum latency for each subexpression of $a(x, y)$ recursively, saving at the same time all the decompositions leading to this latency. This way, we can forget about complete schemes and really focus on the minimum latency itself. Determining which subexpression will be worth considering during the DAG set computation then consists in looking at each decomposition stored for $a(x, y)$ in h, marking the two corresponding subexpressions, and going on recursively.

[12] See `http://gappa.gforge.inria.fr/` and [20].

Algorithm 1: MinLat

Input: a bivariate polynomial $a(x, y)$.
Data: the costs $\mathcal{C}_+$ for $+$ and $\mathcal{C}_\times$ for $\times$, a table d with the delays for x, y and each $a_{i,j}$, and a mapping h where we will store the good decompositions for each subexpression.
Output: minimal latency to evaluate $a(x, y)$

```
begin
1   r ← ∞
2   if a(x,y) = a_{i,j} or x or y then r ← d[a(x,y)]
3   else
4       foreach (⋄, p1, p2) such that p1 and p2 are subexpressions of a, and a = p1 ⋄ p2 do
5           r1 ← MinLat(p1)
6           r2 ← MinLat(p2)
7           if C⋄ + max{r1, r2} < r then
8               r ← C⋄ + max{r1, r2}
9               h[a(x,y)] ← {(⋄, p1, p2)}
10          else if C⋄ + max{r1, r2} = r then
11              h[a(x,y)] ← h[a(x,y)] ∪ {(⋄, p1, p2)}
12  return r
end
```

In practice, we use memoization in order to be efficient, that is, we save the already computed minimum latencies so as to reuse them later if needed. Thus, the cost for Algorithm 1 can be bounded with the number of subexpressions encountered times the treatment cost for one subexpression. So, if we start with the polynomial $a(x, y) = a_{0,0} + y \cdot p(x)$ with deg$p = n$, we will get $O(2^n)$ subexpressions recursively, and finding the latency for one subexpressions cost at most $O(2^n)$. The total cost is therefore $O(2^{2n})$.

As it may still take too much time for some applications, we can use the restriction on the divide-and-conquer decompositions presented in Section 3.4 in order to speed up this precomputation. When considering only splitting into low and upper parts, we get $O(n^3)$ subexpressions[13] and a cost per subexpression in $O(n)$, which gives us a $O(n^4)$ complexity. Notice that this is polynomial with respect to the degree n, making this approach slightly scalable.

4.2 Early elimination of DAGs based on accuracy checking

We have seen in Section 3.4 how to use the latency of a DAG in order to decide whether we keep it or not for the sequel of the DAG set computation step. This early elimination of DAGs, designed to reduce the number of generated DAGs, was only based on some latency criterion, whereas we also have some constraints on the accuracy. Moreover, we put a limitation on the number of DAGs associated to each subexpression, so that we only keep, for subexpressions admitting lots of DAGs, a limited number of

[13] We have $\binom{n+2}{2} = O(n^2)$ ways to get a contiguous support included in the support of $a(x, y)$, and each of them admits $O(n)$ factorizations.

them. This heuristic allows us to speed up the generation significantly but the choice of the kept DAGs is mainly arbitrary.

One way to improve this elimination of DAGs lies in using some accuracy measurement to help us in choosing better DAGs, at least for the accuracy point of view. Therefore, we propose to attach to each generated DAG a certified enclosure of its evaluation error, denoted by **error**, in addition to an interval enclosing its possible values. Let $\mathcal{G}$ be such a DAG, with $\mathcal{G}_\ell$ and $\mathcal{G}_r$ its left and right children. The evaluation error bound of $\mathcal{G}$ can be computed, using interval arithmetic with the library MPFI, as follows:

- It $\mathcal{G}$ is reduced to one node corresponding to a coefficient or a variable, we have $\mathbf{error}(\mathcal{G}) = [0, 0]$, since coefficients and variables are exactly representable.

- If $\mathcal{G}$ represents an addition of two subexpressions, they have to be in the same fixed-point format. When it is the case and when $\mathbf{value}(\mathcal{G})$ has a constant sign, if no overflow occurs, the addition entails no error so we define
$$\mathbf{error}(\mathcal{G}) = \mathbf{error}(\mathcal{G}_\ell) + \mathbf{error}(\mathcal{G}_r).$$
Otherwise, the current scheme is not suitable in the sense detailed in Section 3.5 for the arithmetic operator choice, and so we discard it.

- If $\mathcal{G}$ represents a multiplication, we first check whether the fixed-point numbers in $\mathbf{value}(\mathcal{G})$ can be stored with f bits for the integer part, where f is the integer part size chosen for the current subexpression. If not, the current scheme is not suitable and thus discarded. Otherwise, the error bound is computed as follows:
$$\begin{aligned}\mathbf{error}(\mathcal{G}) &= \mathbf{error}_{\text{mul}} + \mathbf{error}(\mathcal{G}_\ell) \cdot \mathbf{error}(\mathcal{G}_r) \\ &+ \mathbf{error}(\mathcal{G}_\ell) \cdot \mathbf{mag}\big\{\mathbf{value}(\mathcal{G}_r)\big\} \\ &+ \mathbf{error}(\mathcal{G}_r) \cdot \mathbf{mag}\big\{\mathbf{value}(\mathcal{G}_\ell)\big\},\end{aligned}$$
where $\mathbf{error}_{\text{mul}}$ is the error entailed by the multiplication itself, and $\mathbf{mag}\{I\}$ is the magnitude of I (the maximum between the absolute value of each end-point of I). On the ST231 processor, we have $\mathbf{error}_{\text{mul}} = [0, 2^{f-32}]$.

Notice that with this model, having a smaller error bound for $\mathcal{G}_\ell$ and/or $\mathcal{G}_r$ leads to a better bound for $\mathcal{G}$. Thus, keeping only the best DAGs with respect to accuracy at each step allows us to optimize the evaluation error bounds for the DAGs obtained at the end of the process.

If the user provides some maximal error bound, it is then straightforward to select among the final set of DAGs those satisfying this constraint. Moreover, now that we have these error bounds at the DAG set computation step, and because they are certified thanks to interval arithmetic, a numerical accuracy filter in the selection step becomes useless. When no formal proof is required, we can thus avoid the calls to Gappa, and thus save some time for the whole process.

One can think of other criteria to guide the early elimination of DAGs. For instance, we can adapt the model introduced above so that it fits with some other architecture. We can also sort the current DAGs with respect to their number of multiplications, and keep only the ones with few multiplications, which will more likely pass the scheduling filter. Using this criterion is less effective than considering the accuracy since optimizing the number of multiplications for the evaluation of each subexpression does not guarantee that we will end up with a scheme with a minimal number of multiplications for

the expression (because of common subexpressions). Nevertheless, this is still quite interesting to try to decrease the final number of multiplications in order to have DAGs likely to admit a satisfactory schedule because testing whether a DAG can be scheduled can be costly, especially when the test actually fails (see Section 5.2 for more details).

5 Application examples

This section presents some experimental examples, and results are discussed. Here, we assume that the target architecture is the ST231, a 4-issue 32-bit VLIW integer processor. Recall that addition, subtraction and shift have a latency of 1 cycle, while multiplication costs 3 cycles. Experiments have been carried out on a laptop ThinkPad Duo Core2 2.53GHz, under GNU/Linux environment.

5.1 Impact of our heuristics when dealing with a small-degree polynomial

Let us consider the implementation of the *binary16* square root function, in precision 11 [9, Table 3.5], optimized for a 32-bit architecture like the ST231 processor. Using [28, Script 3.1], we know that it may be implemented with a bivariate polynomial approximant $P(s,t) = 2^{-12} + s \cdot a(t)$, with $s \in \{1, \mathrm{RN}_{31}(\sqrt{2})\}$ and $t \in [0, 1-2^{-10}]$, and $a(t)$ a degree-3 univariate polynomial.[14] In this example, s is known 2 cycles after t. Polynomial coefficients are computed using Sollya: $a_0 = 536914839 \cdot 2^{-29}$, $a_1 = 1067301943 \cdot 2^{-31}$, $a_2 = -115190619 \cdot 2^{-30}$, and $a_3 = 52601099 \cdot 2^{-31}$. To ensure correct rounding, the evaluation error has to be no greater than $2^{-12.92}$.[15]

Exactly 88384 schemes may be used for evaluating such a polynomial. We have computed with CGPE all these schemes and checked which ones satisfy the accuracy constraint. This experiment was handled in about 3h40m. Finally, only 42672 of the 88384 schemes passed the first filter (arithmetic operator choice), but among these ones, all passed the next two filters, and in particular the numerical checking. We observe that the fastest program evaluates the polynomial P in 10 cycles by using only 4 multiplications, with an evaluation error of $\approx 2^{-28.73}$.

Then, we run CGPE with our heuristics, asking for lowest latency, keeping 50 schemes at each step of the computation, and bounding the recursion depth to 2 levels. This second experiment was handled in about 10s. At the end of the process, the 41 computed schemes that passed the first filter were all accurate enough for ensuring correct rounding. Among these 41 schemes, the fastest one evaluates the polynomial P in 10 cycles by using only 6 multiplications, and with an evaluation error of $\approx 2^{-28.73}$. This generated program is presented in Listing 1, where $T = t \cdot 2^{32}$, $S = s \cdot 2^{31}$, and $\mathrm{mul}(A,B) = \lfloor A \cdot B / 2^{32} \rfloor$. Remark that if we keep only 5 schemes at each step, we obtain 5 schemes at the end of the process, in about 1s. Moreover, the best one still evaluates P in 10 cycles, using only 6 multiplications, and with an evaluation error of $\approx 2^{-28.73}$.

These experiments show the impact of our heuristics. In particular, we observe that they allow us to reduce significantly the generation cost, while the generated programs remain as fast as the "best" one, and still accurate enough.

[14] Here $\mathrm{RN}_k(X)$ denotes the RoundTieToEven of X in precision k.

[15] Certified evaluation error bound computed with Sollya:
$874035362139639616487950244196397 55 \times 2^{-129}$.

```
// a0 = +0x8002ae5cp-31, a1 = +0x3f9dbc37p-31
// a2 = -0x0dbb56b6p-31, a3 = +0x0322a10bp-31
// Input formats:  T -> 0.32 and  S -> 1.31
uint32_t binary16sqrt(uint32_t T, uint32_t S)
{                                       // Formats
  uint32_t r0 = mul(T, 0x3f9dbc37);     // 1.31
  uint32_t r1 = 0x8002ae5c + r0;        // 1.31
  uint32_t r2 = mul(S, r1);             // 2.30
  uint32_t r3 = 0x00040000 + r2;        // 2.30
  uint32_t r4 = mul(T, T);              // 0.32
  uint32_t r5 = mul(S, r4);             // 1.31
  uint32_t r6 = mul(T, 0x0322a10b);     // 1.31
  uint32_t r7 = 0x0dbb56b6 - r6;        // 1.31
  uint32_t r8 = mul(r5, r7);            // 2.30
  uint32_t r9 = r3 - r8;                // 2.30
  return r9;
}
```

Listing 1. Generated evaluation program.

	$x^{1/2}$	$x^{-1/2}$	$x^{1/3}$	$x^{-1/3}$	$\log_2(1+x)$	$\frac{1}{\sqrt{1+t^2}}$	$\frac{\exp(1+x)}{1+x}$	$\frac{\sin(1+x)}{1+x}+1$	$\exp(\cos(1+x))$
Degree	(8,1)	(9,1)	(8,1)	(9,1)	(6,0)	(7,0)	(10,0)	(5,0)	(8,0)
Approximation interval	$\{1, 2^{1/2}\}\times[0,1]$		$\{1, 2^{1/3}, 2^{2/3}\}\times[0,1]$		$[0.5, 1]$	$[0, 0.5]$	$[0, 1]$	$[0, 1]$	$[0, 1]$
Target / Minimal latency	13 / 13	13 / ?	16 / 16	16 / 16	10 / 11	10 / 11	13 / 13	10 / 10	13 / 13
Achieved latency	13	14	16	16	11	11	13	10	13
Scheme computation	195ms	73ms	26s	25s	17ms	10ms	40ms	1ms	205ms
	[50]	[50]	[50]	[50]	[50]	[50]	[50]	[50]	[50]
Arithmetic operator choice	3ms	3ms	7ms	11ms	1ms	2ms	3ms	1ms	4ms
	[35]	[29]	[30]	[26]	[2]	[12]	[27]	[8]	[16]
Scheduling checking	16s	1m33s	43ms	439ms	2ms	64ms	49s	1ms	91ms
	[11]	[1]	[30]	[24]	[1]	[5]	[5]	[8]	[15]
Certification (Gappa)	10s	1s	27s	27s	230ms	1s	7s	1s	9s
	[11]	[1]	[30]	[24]	[1]	[5]	[4]	[8]	[13]
Total time ($\approx$)	27s	1m35s	55s	53s	1s	2s	57s	1s	9s

Table 1. Timings for certified code generation for some functions.

5.2 Timings for each step

Let us now observe the time spent in each phases of the generation. For doing this, we have considered the implementation of various functions. For each of them, we have computed a polynomial approximant and a certified evaluation error bound using Sollya and the framework presented in [28, § 6.4]. Finally, using CGPE, we have generated some evaluation programs. At each step of the generation, we have kept only 50 schemes. As shown in Table 1, the approach we have presented in this article allows us to quickly generate fast and certified programs for implementing various functions. Numbers in brackets represent how many schemes we got at each step.

The two most expensive steps are the scheduling and the certification with Gappa. For the reciprocal square root ($x^{-1/2}$), the 50 computed schemes have a latency of 13 cycles, but none of them can be scheduled in 13 cycles. Actually, our scheduler tries to schedule each DAG in 13 cycles first, before finding a schedule in 14 cycles in a second step: roughly, it does two steps of scheduling. Therefore, in this case, the time spent in scheduling is much longer.

Concerning the certification, it consists in an external call to Gappa, that uses interval arithmetic as well as rewriting rules and theorems to provide tighter bounds than naive interval arithmetic. However, when Gappa checks if the evaluation error is less than a given bound, it may perform some bisections on the intervals enclosing the values of the variables. That is why it can be more costly than MPFI, which only provides an enclosure. Consequently, it explains our interest in implementing the certification step using MPFI, especially since, in our context, accuracy constraints are not so restrictive: there exists enough DAGs for which we can prove that they are accurate enough using only naive interval arithmetic.

One can see the impact of the three filters by looking at the example of $\exp(\cos(1+x))$. It clearly shows that each filter removes successively various invalid schemes.

The last remark concerns the optimality in terms of evaluation latency of some computed evaluation programs. Indeed, let us consider the example of the square root function $x^{1/2}$. The computed programs have a latency of 13 cycles. But the target latency is also of 13 cycles, which means that no scheme has a latency less than 13 cycles. Hence, it implies that the these computed evaluation programs have optimal evaluation latency. The same conclusion holds for various other functions.

6 Conclusions

In this paper, we have introduced CGPE, a tool dedicated to the automatic generation of fast and certified C code for bivariate polynomial evaluation in fixed-point arithmetic. The underlying problem lies in the number of evaluation schemes being too large even for small degrees. Hence the efficiency of this tool relies on the heuristics we have implemented to reduce the combinatorics, and thus to speed up significantly the whole process.

CGPE has been mainly validated within the development of FLIP, a libm optimized for the ST231, a 4-issue 32-bit VLIW integer processor. We estimate that today it helps us to automate the design of about 50 % of this library. It remains now to validate the generated codes for other kinds of architectures.

We are currently implementing some new heuristics, that should reduce the running time of CGPE. In addition to these first improvements, various directions could be explored. The first one would be to extend this work for generating programs for evaluating polynomials in floating-point arithmetic. This would just consist in modifying the arithmetic model in use for the computation of interval ranges and error bounds. A second direction would be to handle other kinds of polynomials, given in other

representations than the monomial basis (Newton, orthogonal bases, factored form, Knuth and Eve, or Paterson and Stockmeyer). Indeed, in some contexts and on some architectures, such polynomials may be accurate enough, and their implementation much faster. More generally, it is interesting to aim at generating efficient codes for evaluating some other kinds of expressions. For instance, we may want to cover polynomials of matricesnn so as to automatically find non-trivial fast schemes such as the one presented in [8, p. 244].

Finally, CGPE enables to solve one specific problem that usually occurs at the end of a toolchain for code generation of mathematical functions. Adding some support for our tool in a library like the one coming alongside the language LEMA would increase its potential. On the other side, we would benefit from the LEMA script language and thus improve the scripts we currently use to get the polynomial coefficients and the error bound we give as an input to CGPE.

References

[1] S. Boldo. *Preuves formelles en arithmétiques à virgule flottante*. PhD thesis, ÉNS Lyon, November 2004.

[2] M. Ceberio and V. Kreinovich. Greedy algorithms for optimizing multivariate Horner schemes. *SIGSAM Bulletin*, 38(1):8–15, 2004.

[3] R. C. C. Cheung, D.-U. Lee, O. Mencer, W. Luk, and P. Y. K. Cheung. Automating custom-precision function evaluation for embedded processors. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 22–31, New York, NY, USA, 2005. ACM.

[4] J. Eve. The evaluation of polynomials. *Numerische Mathematik*, (6):17–21, 1964.

[5] S. R. Finch. *Mathematical Constants*. Cambridge University press, 1994.

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

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

[8] N. J. Higham. *Functions of Matrices: Theory and Computation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008.

[9] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. Aug. 2008.

[10] C.-P. Jeannerod, H. Knochel, C. Monat, and G. Revy. Computing floating-point square roots via bivariate polynomial evaluation. Technical Report RR2008-38, LIP, 2008.

[11] C.-P. Jeannerod, C. Mouilleron, J.-M. Muller, G. Revy, C. Bertin, J. Jourdan-Lu, H. Knochel, and C. Monat. Techniques and tools for implementing IEEE 754 floating-point arithmetic on VLIW integer processors. In *Proc. of the 4th International Workshop on Parallel and Symbolic Comp. (PASCO '10)*, pages 1–9, New York, NY, USA, 2010. ACM.

[12] D. E. Knuth. Evaluation of polynomials by computers. *Communications of the ACM*, 5(12):595–599, 1962.

[13] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Third edition, 1998.

[14] P. Langlois, M. Martel, and L. Thévenoux. Accuracy versus time: a case study with summation algorithms. In *Proc. of the 4th International Workshop on Parallel and Symbolic Computation (PASCO '10)*, pages 121–130, New York, NY, USA, 2010. ACM.

[15] C. Lauter. *Arrondi correct de fonctions mathématiques - fonctions univariées et bivariées, certification et automatisation*. PhD thesis, Univ. de Lyon - ÉNS Lyon, Oct. 2008.

[16] D.-U. Lee and J. D. Villasenor. Optimized Custom Precision Function Evaluation for Embedded Processors. *IEEE Transactions on Computers*, 58(1):46–59, 2009.

[17] V. Lefévre, P. Théveny, F. de Dinechin, C.-P. Jeannerod, C. Mouilleron, D. Pfannholzer, and N. Revol. LEMA: towards a language for reliable arithmetic. *SIGSAM Bull.*, 44(1/2):41–52, 2010.

[18] M. Martel. Enhancing the Implementation of Mathematical Formulas for Fixed-Point and Floating-Point Arithmetics. In *Journal of Formal Methods in System Design*, volume 35, pages 265–278. Springer, 2009.
[19] M. Martel. Program transformation for numerical precision. In *PEPM'09*. ACM Press, 2009.
[20] G. Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, ÉNS Lyon, Nov. 2006.
[21] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.
[22] R. Otter. The number of trees. *The Annals of Mathematics*, 49(3):pp. 583–599, 1948.
[23] V. Y. Pan. Methods of Computing Values of Polynomials. *Russian Mathematical Surveys*, 21(1):105–136, 1966.
[24] M. S. Paterson and L. J. Stockmeyer. On the number of nonscalar multiplications necessary to evalutate polynomials. *SIAM Journal on Computing*, 2(1):60–66, 1973.
[25] J. M. Peña and T. Sauer. On the multivariate Horner scheme. *SIAM Journal on Num. Analysis*, 37(4):1186–1197, 2000.
[26] M. Püschel, F. Franchetti, and Y. Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer, 2011.
[27] N. Revol and F. Rouillier. Motivations for an arbitrary precision interval arithmetic and the MPFI library. *Reliable Computing*, 11(4):275–290, 2005.
[28] G. Revy. *Implementation of binary floating-point arithmetic on embedded integer processors - Polynomial evaluation-based algorithms and certified code generation*. PhD thesis, Univ. de Lyon - ÉNS Lyon, Dec 2009.