A High Throughput Polynomial and Rational Function Approximations Evaluator

Nicolas Brisebarre*, George Constantinides†, Miloš Ercegovac‡, Silviu-Ioan Filip§, Matei Istoan†, and Jean-Michel Muller*
* Univ. Lyon, CNRS, ENS de Lyon, Inria, Univ. Claude Bernard Lyon 1, LIP, France
Email: FirstName.LastName@ens-lyon.fr
† Electrical and Electronic Engineering, Imperial College London, SW7 2AZ, United Kingdom
Email: {g.constantinides,m.istoan}@imperial.ac.uk
‡ Computer Science Department, UCLA, Los Angeles, CA 90095, USA
Email: milos@cs.ucla.edu
§ Univ Rennes, Inria, CNRS, IRISA, F-35000 Rennes, France
Email: silviu.filip@inria.fr

***Abstract*—We present an automatic system to evaluate functions in hardware via polynomial or rational approximations. These approximations are evaluated using Ercegovac's iterative E-method adapted for FPGA implementation. The polynomial and rational function coefficients are optimized such that they satisfy the constraints of the E-method. We present several examples of practical interest. In each case, the most resource-efficient approximation is used.**

I. Introduction

We aim at designing a system able to approximate (in software) and then evaluate (in hardware) any regular-enough function. More precisely, we try to minimize the sup norm of the difference between the function and the approximation in a given interval.

For particular functions, *ad hoc* solutions such as CORDIC [1] or some specific tabulate-and-compute algorithms [2] can be used. For low precision cases, table-based methods [3]–[5] methods are of interest. However, in the general case, piecewise approximations by polynomial or rational functions are the only reasonable solution. From a theoretical point of view, rational functions are very attractive, mainly because they can reproduce function behaviors (such as asymptotes, finite limits at $\pm\infty$) that polynomials do not satisfy. However, for software implementation, polynomials are frequently preferred to rational functions, because the latency of division is larger than the latency of multiplication. We aim at checking if rational approximations are of interest in hardware implementations. To help in the comparison of polynomial and rational approximations in hardware we use an algorithm, due to Ercegovac [6], [7], called the E-method, that makes it possible to evaluate a degree-n polynomial, or a rational function of degree-n numerator and denominator at a similar cost without requiring division.

The E-method solves diagonally-dominant linear systems using a left-to-right digit-by-digit approach and has a simple and regular hardware implementation. It maps the problem of evaluating a polynomial or rational function into a linear system. The linear system corresponding to a given function does not necessarily satisfy the conditions of diagonal dominance. For polynomials, changes of variables allow one to satisfy the conditions. This is not the case for rational functions. There is however a family of rational functions, called E-fractions, that can be evaluated with the E-method in time proportional to the desired precision. One of our aims is, given a function, to decide whether it is better to approximate it by a polynomial or by an E-fraction. Furthermore, we want to design approximations whose coefficients satisfy some constraints (such as being exactly representable in a given format). We introduce algorithmic improvements with respect to [8] for computing E-fractions. We present a circuit generator for the E-method and compare its implementation on an FPGA with FloPoCo polynomial designs [9] for several examples of practical interest. Since FloPoCo designs are pipelined (unrolled), we focus on an unrolled design of the E-method.

A. An Overview of the E-method

The E-method evaluates a polynomial $P_\mu(x)$ or a rational function $R_{\mu,\nu}(x)$ by mapping it into a linear system. The system is solved using a left-to-right digit-by-digit approach, in a radix r representation system, on a regular hardware. For a result of m digits, in the range $(-1,1)$, the computation takes m iterations. The first component of the solution vector corresponds to the value of $P_\mu(x)$ or $R_{\mu,\nu}(x)$. Let

$$R_{\mu,\nu}(x)=\frac{P_\mu(x)}{Q_\nu(x)}=\frac{p_\mu x^\mu+p_{\mu-1}x^{\mu-1}+\cdots+p_0}{q_\nu x^\nu+q_{\nu-1}x^{\nu-1}+\cdots+q_1x+1}$$

where the p_i's and q_i's are real numbers. Let $n=\max\{\mu,\nu\}$, $p_j=0$ for $\mu+1\leqslant j\leqslant n$, and $q_j=0$ for $\nu+1\leqslant j\leqslant n$. According to the E-method $R_{\mu,\nu}(x)$ is mapped to a linear system $L:\mathbf{A}\times\mathbf{y}=\mathbf{b}$:

$$\begin{bmatrix} 1 & -x & 0 & & \cdots & 0\\ q_1 & 1 & -x & 0 & \cdots & 0\\ q_2 & 0 & 1 & -x & \cdots & 0\\ & & \ddots & \ddots & & \vdots\\ & & & & & \\ \vdots & & & & \ddots & 0\\ q_{n-1} & & & & 1 & -x\\ q_n & & & \cdots & 0 & 1 \end{bmatrix}\begin{bmatrix} y_0\\ y_1\\ y_2\\ \vdots\\ \\ \vdots\\ y_{n-1}\\ y_n \end{bmatrix}=\begin{bmatrix} p_0\\ p_1\\ p_2\\ \vdots\\ \\ \vdots\\ p_{n-1}\\ p_n \end{bmatrix} \quad (1)$$

so that $y_0=R_{\mu,\nu}(x)$. Likewise, $y_0=P_\mu(x)$ when all $q_i=0$.

The components of the solution vector $\mathbf{y} = [y_0, y_1, \ldots, y_n]^t$ are computed, digit-by-digit, the most-significant digit first, by means of the following vector iteration:

$$\mathbf{w}^{(j)} = r \times \left[\mathbf{w}^{(j-1)} - \mathbf{A}\mathbf{d}^{(j-1)}\right], \tag{2}$$

for $j = 1, \ldots, m$, where m is the desired precision of the result. The term $\mathbf{w}^{(j)}$ is the vector residual in iteration j with $\mathbf{w}^{(0)} = [p_0, p_1, \ldots, p_n]^t$. The solution $\mathbf{y}$ is produced as a sequence of digit vectors: $\mathbf{d}^{(j-1)} = [d_1^{(j-1)}, \ldots, d_n^{(j-1)}]^t$ – a digit vector obtained in iteration $j-1$ and used in iteration j. After m iterations, $y_k = \sum_{j=1}^{m} d_k^{(j)} r^{-j}$. The digits of the solution components y_0, y_1, …, y_n are computed using very simple scalar recurrences. Note that all multiplications in these recurrences use $m \times 1$ multipliers and that division required by the rational function is not explicitly performed.

$$w_i^{(j)} = r \times \left[w_i^{(j-1)} - q_i d_0^{(j-1)} - d_i^{(j-1)} + d_{i+1}^{(j-1)} x\right], \tag{3}$$

$$w_0^{(j)} = r \times \left[w_0^{(j-1)} - d_0^{(j-1)} + d_1^{(j-1)} x\right] \tag{4}$$

and

$$w_n^{(j)} = r \times \left[w_n^{(j-1)} - d_n^{(j-1)} - q_n d_0^{(j-1)}\right]. \tag{5}$$

Initially, $\mathbf{d}^{(0)} = \mathbf{0}$. The radix-$r$ digits $d_i^{(j)}$ are in the redundant signed digit-set $D_\rho = \{-\rho, \ldots, 0, 1, \ldots \rho\}$ with $r/2 \leqslant \rho \leqslant r-1$. If $\rho = r/2$, $\mathcal{D}_\rho$ is called *minimally redundant*, and if $\rho = r-1$, it is *maximally redundant*. The choice of redundancy is determined by design considerations. The radix of computation is $r = 2^k$ so that internally radix-2 arithmetic is used. The residuals, in general, are in a redundant form to reduce the iteration time. Since the target is an FPGA technology which provides fast carry chains, we have non-redundant residuals. The digits $d_i^{(j)}$ are selected so that the residuals $|w_i^{(j)}|$ remain bounded. The digit selection is performed by rounding the residuals $w_i^{(j)}$ to a single signed digit, following [7], [10]:

$$d_i^{(j)} = S(w_i^{(j)}) = \begin{cases} \text{sign}\ (w_i^{(j)}) \times \left\lfloor \left|w_i^{(j)}\right| + \frac{1}{2} \right\rfloor, & \text{if } \left|w_i^{(j)}\right| \leqslant \rho, \\ \text{sign}\ (w_i^{(j)}) \times \left\lfloor \left|w_i^{(j)}\right| \right\rfloor, & \text{otherwise.} \end{cases}$$

The selection is performed using a low-precision estimate $\widehat{w}_i^{(j)}$ of $w_i^{(j)}$, obtained by truncating $w_i^{(j)}$ to one fractional bit. Since the matrices considered here have 1s on the diagonal, a necessary condition for convergence is $\sum_{j \neq i} |a_{i,j}| < 1$. Specifically,

$$\begin{cases} \forall i, |p_i| \leqslant \xi, \\ \forall i, |x| + |q_i| \leqslant \alpha, \\ |w_i^{(j)} - \widehat{w}_i^{(j)}| \leqslant \Delta/2. \end{cases} \tag{6}$$

where the bounds ξ, α, and Δ satisfy [7]:

$$\xi = \frac{1}{2}(1+\Delta), \quad 0 < \Delta < 1, \quad \alpha \leqslant (1-\Delta)/(2r) \tag{7}$$

for maximally redundant digit sets used here. While the constraints (7) may seem restrictive, for polynomials, scaling techniques make it possible to satisfy them. However, this is not the case for all rational functions. To remove this limitation the authors of [8] have suggested the derivation of rational functions, called *simple E-fractions*, which are products of a power of 2 by a fraction that satisfies (7). In this work we make further improvements to the rational functions based on E-fractions.

B. Outline of the paper

In Section II, we discuss the effective generation of simple E-fractions, whose coefficients are exactly representable in a given format. Section III presents a hardware implementation of the E-method that targets FPGAs. In Section V we present and discuss some examples in various situations. We also present a comparison with FloPoCo implementations.

II. Effective computation of simple E-fractions

We show how to compute a simple E-fraction with fixed-point or floating-point coefficients. A first step (see Section II-A), yields a simple E-fraction approximation with real coefficients to a function f. In [8], linear programming (LP) is used. Here, we use faster tools from approximation theory. This allows us to quickly check how far the approximation error of this E-fraction is from the optimal error of the minimax approximation (obtained using the Remez algorithm [11], [12]), and how far it is from the error that an E-polynomial, with the same implementation cost, can yield. If this comparison suggests that it is more advantageous to work with an E-fraction, we use the Euclidean lattice basis reduction approach from [8] for computing E-fractions with machine-number coefficients. We introduce in Section II-B2 a trick that improves its output.

A. Real approximation step

Let f be a continuous function defined on $[a, b]$. Let $\mu, \nu \in \mathbb{N}$ be given and let $\mathbb{R}_{\mu,\nu}(x) = \{P/Q : P = \sum_{k=0}^{\mu} p_k x^k, Q = \sum_{k=0}^{\nu} q_k x^k, p_0, \ldots, p_\mu, q_0, \ldots, q_\nu \in \mathbb{R}\}$. The aim is to compute a good rational fraction approximant $R \in \mathbb{R}_{\mu,\nu}(x)$, with respect to the supremum norm defined by $\|g\| = \sup_{x \in [a,\, b]} |g(x)|$, to f such that the real coefficients of R (or R divided by some fixed power of 2) satisfy the constraints imposed by the E-method.

As done in [8], we can first apply the rational version of the Remez exchange algorithm [11, p. 173] to get $R^\star$, the best possible rational approximant to f among the elements of $\mathbb{R}_{\mu,\nu}(x)$. This algorithm can fail if $R^\star$ is degenerate or the choice of starting nodes is not good enough.

To bypass these issues, we develop the following process. It can be viewed as a Remez-like method of the *first type*, following ideas described in [11, p. 96–97] and [13]. It directly computes best real coefficient E-fractions with magnitude constraints on the denominator coefficients. If we remove these constraints, it will compute the minimax rational approximation, even when the Remez exchange algorithm fails.

We first show how to solve the problem over X, a finite discretization of $[a, b]$. We apply a modified version (with denominator coefficient magnitude constraints) of the differential correction (DC) algorithm introduced in [14]. It is given by Algorithm 1. System (8) is an LP problem and can be solved in practice very efficiently using a simplex-based LP solver. Convergence of this `EDiffCorr` procedure can be shown

Algorithm 1 E-fraction `EDiffCorr` algorithm

Input: $f \in \mathcal{C}([a,b])$, $\mu, \nu \in \mathbb{N}$, finite set $X \subseteq [a,b]$ with $|X| > \mu+\nu$, threshold $\varepsilon > 0$, coefficient magnitude bound $d > 0$

Output: approximation $R(x) = \dfrac{\sum_{k=0}^{\mu} p_k x^k}{1+\sum_{k=1}^{\nu} q_k x^k}$ of f over X s.t. $\max_{1\leqslant k\leqslant \nu} |q_k| \leqslant d$

// Initialize the iterative procedure ($R = P/Q$)

1: $R \leftarrow 1$

2: **repeat**

3: $\quad \delta \leftarrow \max_{x\in X} |f(x) - R(x)|$

4: $\quad$ find $R_{\text{new}} = P_{\text{new}}/Q_{\text{new}} = \dfrac{\sum_{k=0}^{\mu} p'_k x^k}{1+\sum_{k=1}^{\nu} q'_k x^k}$ such that the expression

$$\max_{x\in X}\left\{\frac{|f(x)Q_{\text{new}}(x) - P_{\text{new}}(x)| - \delta Q_{\text{new}}(x)}{Q(x)}\right\} \tag{8}$$

$\quad$ subject to $\max_{1\leqslant k\leqslant \nu} |q'_k| \leqslant d$, is minimized

5: $\quad \delta_{\text{new}} \leftarrow \max_{x\in X} |f(x) - R_{\text{new}}(x)|$

6: $\quad R \leftarrow R_{\text{new}}$

7: **until** $|\delta - \delta_{\text{new}}| < \varepsilon$

Algorithm 2 E-fraction Remez algorithm

Input: $f \in \mathcal{C}([a,b])$, $\mu, \nu \in \mathbb{N}$, finite set $X \subseteq [a,b]$ with $|X| > \mu+\nu$, threshold $\varepsilon > 0$, coefficient magnitude bound $d > 0$

Output: approximation $R^\star(x) = \dfrac{\sum_{k=0}^{\mu} p^\star_k x^k}{1+\sum_{k=1}^{\nu} q^\star_k x^k}$ of f over $[a,b]$ s.t. $\max_{1\leqslant k\leqslant \nu} |q^\star_k| \leqslant d$

// Compute best E-fraction approximation over X using a
// modified version of the differential correction algorithm

1: $R^\star \leftarrow \texttt{EDiffCorr}(f, \mu, \nu, X, \varepsilon, d)$

2: $\delta^\star \leftarrow \max_{x\in X} |f(x) - R^\star(x)|$

3: $\Delta^\star \leftarrow \max_{x\in[a,b]} |f(x) - R^\star(x)|$

4: **while** $\Delta^\star - \delta^\star > \varepsilon$ **do**

5: $\quad x_{\text{new}} \leftarrow \operatorname{argmax}_{x\in[a,b]} |f(x) - R^\star(x)|$

6: $\quad X \leftarrow X \cup \{x_{\text{new}}\}$

7: $\quad R^\star \leftarrow \texttt{EDiffCorr}(f, \mu, \nu, X, \varepsilon, d)$

8: $\quad \delta^\star \leftarrow \max_{x\in X} |f(x) - R^\star(x)|$

9: $\quad \Delta^\star \leftarrow \max_{x\in[a,b]} |f(x) - R^\star(x)|$

10: **end while**

using an identical argument to the convergence proofs of the original DC algorithm [15], [16].

To address the problem over $[a,b]$, Algorithm 2 solves a series of best E-fraction approximation problems on a discrete subset X of $[a,b]$, where X increases at each iteration by adding a point where the current residual term achieves its global maximum.

Our current experiments suggest that the speed of convergence for Algorithm 2 is linear. We can potentially decrease the number of iterations by adding to X more local extrema of the residual term at each iteration. Other than its speed compared to the LP approach from [8], Algorithm 2 will generally converge to the best E-fraction approximation with real coefficients over $[a,b]$, and not on a discretization of $[a,b]$.

Once $R^\star$ is computed, we determine the least integer s such that the coefficients of the numerator of $R^\star$ divided by 2^s fulfill the first condition of (6). It gives us a decomposition $R^\star(x) = 2^s R_s(x)$. R_s is thus a rescaled version of R. We take $f_s = 2^{-s} f$ to be the corresponding rescaling of f. The magnitude bound d is usually equal to $\alpha - \max(|a|,|b|)$, allowing the denominator coefficients to be valid with respect to the second constraint of (6).

Both Algorithm 1 and 2 can be modified to compute weighted error approximations, that is, work with a norm of the form $\|g\| = \max_{x\in[a,b]} |w(x)g(x)|$, where w is a continuous and positive weight function over $[a,b]$. This is useful, for instance, when targeting relative error approximations. The changes are minimal and consist only of introducing the weight factor in the error computations in lines 3, 5 of Algorithm 1, lines 2, 3, 5, 8, 9 of Algorithm 2 and changing (8) with

$$\max_{x\in X}\left\{\frac{w(x)\,|f(x)Q_{\text{new}}(x) - P_{\text{new}}(x)| - \delta Q_{\text{new}}(x)}{Q(x)}\right\}.$$

The weighted version of the DC algorithm is discussed, for instance, in [17].

B. Lattice basis reduction step

Our goal is to compute a simple E-fraction

$$\widehat{R}(x) = \frac{\sum_{j=0}^{\mu} \widehat{p}_j x^j}{1+\sum_{j=1}^{\nu} \widehat{q}_j x^j},$$

where $\widehat{p}_j$ and $\widehat{q}_j$ are fixed-point or floating-point numbers [10], [18], that is as close as possible to f_s, the function we want to evaluate. These unknown coefficients are of the form $M2^e$, $M \in \mathbb{Z}$:

- for fixed-point numbers, e is implicit (decided at design time);
- for floating-point numbers, e is explicit (*i.e.,* stored). A floating-point number is of precision t if $2^{t-1} \leqslant M < 2^t - 1$.

A different format can be used for each coefficient of the desired fraction. If we assume a target format is given for each coefficient, then a straightforward approach is to round each coefficient of R_s to the desired format. This yields what we call in the sequel a *naive rounding* approximation. Unfortunately, this can lead to a significant loss of accuracy. We first briefly recall the approach from [8] that makes it possible to overcome this issue. Then, we present a small trick that improves on the quality of the output of the latter approach. Eventually, we explain how to handle a coefficient saturation issue appearing in some high radix cases.

1) Modeling with a closest vector problem in a lattice: Every fixed-point number constraint leads to a corresponding unknown M, whereas each precision-t floating-point number leads to two unknowns M and e. A heuristic trick is given in [19] to find an appropriate value for each e in the floating-point case: we assume that the coefficient in question from $\widehat{R}$ will have the same order of magnitude as the corresponding one from R_s, hence they have the same exponent e. Once e is set, the problem is reduced to a fixed-point one.

Then, given $u_0, \ldots, u_\mu, v_1, \ldots, v_\nu \in \mathbb{Z}$, we have to determine $\mu+\nu+1$ unknown integers $a_j (= \widehat{p}_j 2^{-u_j})$ and $b_j (= \widehat{q}_j 2^{-v_j})$ such that the fraction

$$\widehat{R}(x) = \frac{\sum_{j=0}^{\mu} a_j 2^{u_j} x^j}{1+\sum_{j=1}^{\nu} b_j 2^{v_j} x^j}$$

is a good approximation to R_s (and f_s), *i.e.*, $\|\widehat{R} - R_s\|$ is small. To this end, we discretize the latter condition in $\mu + \nu + 1$ points $x_0 < \cdots < x_{\mu+\nu}$ in the interval $[a, b]$, which gives rise to the following instance of a closest vector problem, one of the fundamental questions in the algorithmics of Euclidean lattices [20]: we want to compute $a_0, \ldots, a_\mu, b_1, \ldots, b_\nu \in \mathbb{Z}$ such that the vectors

$$\sum_{j=0}^{\mu} a_j \boldsymbol{\alpha}_j - \sum_{j=1}^{n} b_j \boldsymbol{\beta}_j \text{ and } \mathbf{r} \tag{9}$$

are as close as possible, where $\boldsymbol{\alpha}_j = [2^{u_j} x_0^j, \ldots, 2^{u_j} x_{\mu+\nu}^j]^t$, $\boldsymbol{\beta}_j = [2^{v_j} x_0^j R_s(x_0), \ldots, 2^{v_j} x_{\mu+\nu}^j R_s(x_{\mu+\nu})]^t$ and $\mathbf{r} = [R_s(x_0), \ldots, R_s(x_{\mu+\nu})]^t$. It can be solved in an approximate way very efficiently by applying techniques introduced in [21] and [22]. We refer the reader to [8], [19] for more details on this and how the discretization $x_0, \ldots, x_{\mu+\nu}$ should be chosen.

2) A solution to a coefficient saturation issue: While we generally obtain integer a_j and b_j which correspond to a good approximation, the solution is not always guaranteed to give a valid simple E-fraction. What happens is that, in many cases, some of the denominator coefficients in R_s are maximal with respect to the magnitude constraint in (6) (recall that the second line in (6) can be restated as $|q_j| \leqslant \alpha - \max(|a|, |b|)$). In this context, the corresponding values of $|b_j|$ are usually too large. We thus propose to fix the problematic values of b_j to the closest value to the allowable limit that does not break the E-method magnitude constraints.

The change is minor in (9); we just move the corresponding vectors in the second sum on the left hand side of (9) to the right hand side with opposite sign. The resulting problem can also be solved using the tools from [8], [19]. This usually gives a valid simple E-fraction $\widehat{R}$ of very good quality.

3) Higher radix problems: Coefficient saturation issues get more pronounced by increasing the radix r. In such cases, care must also be taken with the approximation domain: the $|q_j|$ upper magnitude bound $\alpha - \max(|a|, |b|)$ can become negative, since $\alpha = (1 - \Delta)/(2r) \to 0$ as $r \to \infty$. To counter this, we use argument and domain scaling ideas presented in [23]. This basically consists in approximating $f(x) = f(2^t y)$, for $y \in [2^{-t}a, 2^{-t}b]$ as a function in y. If $t > 0$ is large enough, then the new $|q_j|$ bound $\alpha - \max(|2^{-t}a|, |2^{-t}b|)$ will be $\geqslant 0$.

III. A HARDWARE IMPLEMENTATION TARGETING FPGAS

We now focus on the hardware implementation of the E-method on FPGAs. This section introduces a generator capable of producing circuits that can solve the system $\mathbf{A} \cdot \mathbf{y} = \mathbf{b}$, through the recurrences of Equations (3)–(5).

The popularity of FPGAs is due to their ability to be reconfigured, and their relevance in prototyping as well as in scientific and high-performance computing. They are composed of large numbers of small look-up tables (LUTs), with 4-6 inputs and 1-2 outputs. They can store the result of any logic function of their inputs. Any two LUTs on the device can communicate, as they are connected through a programmable interconnect network. Results of computations can be stored in registers, usually two of them being connected to the output of each LUT. These features make of FPGAs a good candidate as a platform for implementing the E-method, as motivated even further below.

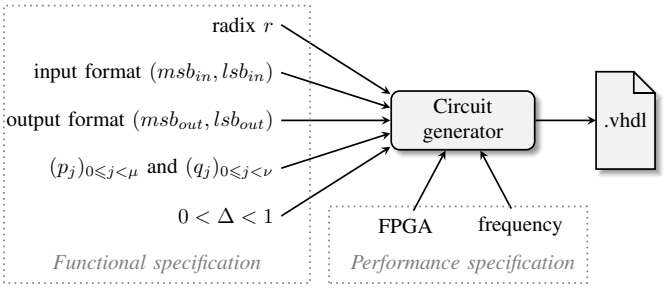

Fig. 1. Circuit generator overview.

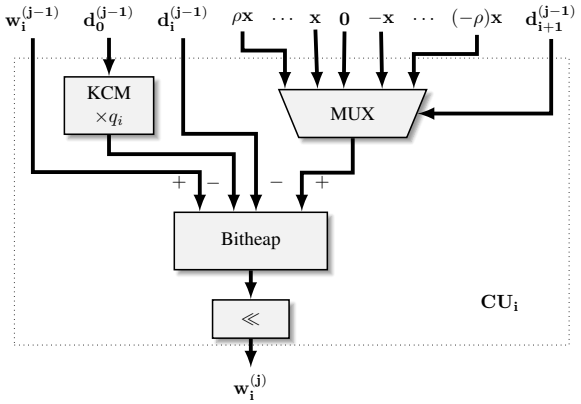

Fig. 2. The basic Computation Unit (CU).

A. A minimal interface

An overview of the generator is presented in Figure 1. Its interface is split according to what a typical user's concerns might be: the *functional* and the *performance* specification. The former consists of the input and output formats, specified as the weights of their most significant (MSB) and least significant (LSB) bits, the coefficients of the polynomials $P_\mu(x)$ and $Q_\nu(x)$, as well as the parameter Δ and the radix r. Having msb_{in} as a parameter is justified by noticing that in the examples of Section V, even though the input x belongs to $[-1, 1]$, the maximum value it is allowed to have is smaller, given by the constraints (6) and (7). It could be argued that msb_{out} can be deduced automatically by the generator. While true, this would involve an unnecessarily complicated analysis, at this stage in the tool flow, so we leave this to the user.

The circuit generator is developed inside the FloPoCo framework [24], which facilitates the support of classical parameters in the performance specification, such as the target frequency of the resulting circuit, or the target device. It also means that we can leverage on the automatic pipelining and test infrastructure present in the framework, alongside the numerous existing arithmetic operators.

B. Implementation details

An overview of the basic iteration, based on Equation (3), is presented in Figure 2. As this implementation is targeted towards FPGAs, several optimizations can be applied. First, the multiplication $d_0^{(j-1)} \cdot q_i$ can be computed using the KCM technique for multiplying by a constant [25], [26], with the optimizations of [27], that extend the method for real constants. Therefore, instead of using dedicated multiplier blocks (or of generating partial products using LUTs), we can directly tabulate the result of the multiplication $d_0^{(j-1)} \cdot q_i$, at the cost of one LUT per output bit of the result. This remains true even for higher radices, as LUTs on modern devices can accommodate functions of 6 Boolean inputs.

A second optimization relates to the term $d_{i+1}^{(j-1)} \cdot x$, from Equation (3). Since $d_{i+1}^{(j-1)} \in \{-\rho, \ldots, \rho\}$, we can compute the products $x \cdot \rho$, $x \cdot (\rho - 1)$, $x \cdot (\rho - 2)$, ..., only once, and then select the relevant one based on the value of $d_{i+1}^{(j-1)}$. The multiplications by the negative values in the digit set come at the cost of just one bitwise operation and an addition, which are combined in the same LUT by the synthesis tools.

Finally, regarding the implementation of the CUs, the multi-operand addition of the terms of Equation (3) is implemented using a *bitheap* [28]. The alignments of the accumulated terms and their varied sizes would make for a wasteful use of adders. Using a bitheap we have a single, global optimization of the accumulation. In addition, managing sign extensions comes at the cost of a single additional term in the accumulation, using a technique from [10]. The sign extension of a two's complement fixed point number `sxx...xx` is performed as:

$$\begin{array}{rl} & \texttt{00...0}\overline{\texttt{s}}\texttt{xxxxxxx} \\ + & \texttt{11...110000000} \\ = & \texttt{ss...ssxxxxxxx} \end{array}$$

The sum of the constants is computed in advance and added to the accumulation. The final shift comes at just the cost of some routing, since the shift is by a constant amount.

Modern FPGAs contain fast carry chains. Therefore, we represent the components of the residual vector w using two's complement, as opposed to a redundant representation. The selection function only requires $\approx 1 + \log_r \delta$ digits of w_i, therefore it can simply be tabulated using LUTs.

Iteration 0, the initialization, comes at almost no cost, and can be done through fixed routing in the FPGA. This is also true for the second iteration, as simplifying the equations results in $w_i^{(1)} = r \times w_i^{(0)}$. The corresponding digits $d_i^{(1)}$ can be pre-computed and stored directly. This not only saves one iteration, but also improves the accuracy, as $w_i^{(1)}$ and $d_i^{(1)}$ can be pre-computed using higher-precision calculations. Going one iteration further, we can see that most of the computations required for $w_i^{(2)}$ can also be done in advance, except, of course, those involving x.

Figure 3 shows an unrolled implementation of the E-method, that uses the CUs of Figure 2 as basic building blocks. At the top of the architecture, a scaling can be applied to the input. This step is optional, and the scale factor (optional parameter

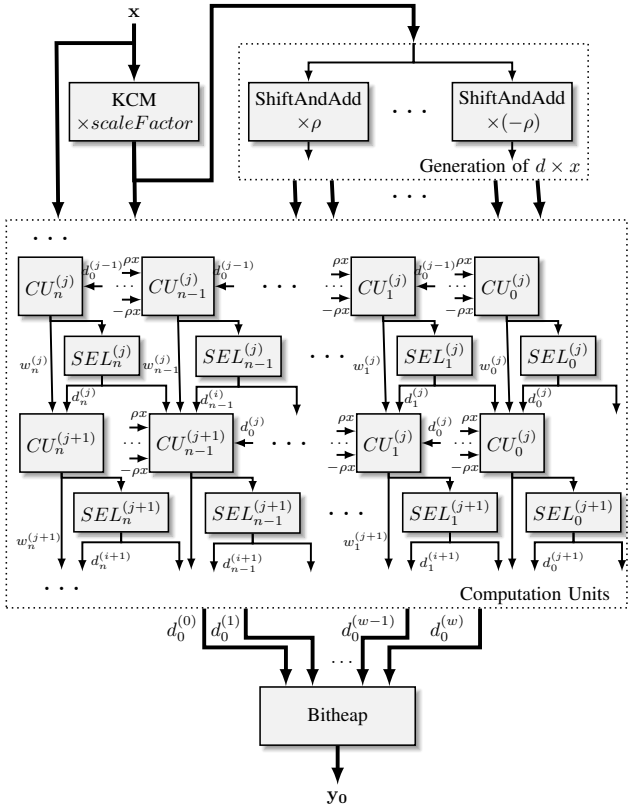

Fig. 3. The E-method circuit generator.

in the design) can either be set by the user, or computed by the generator so that, given the input format, the parameter Δ and the coefficients of P and Q, the scaled input satisfies the constraints (6) and (7). The multiplications between x and the possible values of the digits $d_i^{(j)}$ are done using the classical shift-and-add technique, a choice justified by the small values of the constants and the small number of bits equal to 1 in their representations. At the bottom of Figure 3, the final result y_0 is obtained in two's complement representation. Again, this step is also optional, as users might be content with having the result in the redundant representation.

There is one more optimization that can be done here due to an unrolled implementation. Because only the $d_0^{(j)}$ digits are required to compute y_0, after iteration $m - n$ we can compute one less element of $\mathbf{w}^{(j)}$ and $\mathbf{d}^{(j)}$ at each iteration. This optimization is the most effective when the number of required iterations m is comparable to n, in which case the required hardware is reduced to almost half.

C. Error Analysis

To obtain a minimal implementation for the circuit described in Figure 3, we need to size the datapaths in a manner that guarantees that the output y_0 remains unchanged, with respect to an ideal implementation, free of potential rounding errors. To that end, we give an error-analysis, which follows [6, Ch. 2.8]. For the sake of brevity, we focus on the radix 2 case.

In order for the circuit to produce correct results, we must ensure that the rounding errors do not influence the selection

function: $S(\widetilde{w}_i^{(j)}) = S(w_i^{(j)}) = d_i^{(j)}$, where the tilded terms represent approximate values. In [6], the idea is to model the rounding errors due to the limited precision used to store the coefficients p_j and q_j inside the matrix A as a new error matrix $\mathbf{E_A} = (\varepsilon_{ij})_{n\times n}$. With the method introduced in this paper, the coefficients are machine representable numbers, and therefore incur no additional error. What remains to deal with are errors due to the limited precision of the involved operators. The only one that could produce rounding errors is the multiplication $d_0^{(j-1)} \cdot q_i$. We know that $d_0^{(j-1)} \geqslant 1$ (the case $d_0^{(j-1)} = 0$ is clearly not a problem), so the LSB of $d_0^{(j-1)} \cdot q_i$ is at least that of q_i, if not larger. If the output precision satisfies $lsb_{out} \geqslant lsb_{q_i}$ (which is usually the case), we perform this operation on its full precision, so we do not require any additional guard bits for the internal computations. If this assumption does not hold, based on [6], we obtain the following expression for the rounding errors introduced when computing $\mathbf{w}^{(j)}$ inside Equation (2), denoted with $\boldsymbol{\varepsilon}_{\mathbf{w}}^{(j)}$:

$$\boldsymbol{\varepsilon}_{\mathbf{w}}^{(j)} = 2 \cdot (\boldsymbol{\varepsilon}_{\mathbf{w}}^{(j-1)} + \boldsymbol{\varepsilon}_{const\_mult} + \mathbf{E_A} \cdot \mathbf{d}^{(j-1)}).$$

We can thus obtain an expression for $\boldsymbol{\varepsilon}_{\mathbf{w}}^{(m)}$, the error vector at step m, where m is the bitwidth of y_0 and $\boldsymbol{\varepsilon}_{const\_mult}$ are the errors due to the constant multipliers. Since $\boldsymbol{\varepsilon}_{\mathbf{w}}^{(0)} = 0$,

$$\varepsilon_{w_0}^{(m)} = 2^m \cdot (\varepsilon_{const\_mult} + \|\mathbf{E_A}\| \cdot \sum_{j=1}^{m} d_0^{(j)} \cdot 2^{-j}),$$

where $\|\mathbf{E_A}\|$ is the matrix 2-norm. We use a larger intermediary precision for the computations, with g extra guard bits. Therefore, we can design a constant multiplier for which $\varepsilon_{const\_mult} \leqslant \bar{\varepsilon}_{const\_mult} \leqslant 2^{-m-g}$. Also,

$$\|\mathbf{E_A}\| \leqslant \max_i \sum_{j=1}^{n} |\varepsilon_{ij}| \quad \text{and} \quad \sum_{j=i}^{m} d_0^{(j)} \cdot 2^{-j} < 1,$$

hence we can deduce that for each $w_i^{(m)}$ we have $\varepsilon_{w_i}^{(m)} \leqslant \bar{\varepsilon}_{w_i}^{(m)} \leqslant 2^m(2^{-m-g} + n \cdot 2^{-m-g})$. In order for the method to produce correct results, we need to ensure that $\bar{\varepsilon}_w^{(m)} \leqslant \Delta/2$, therefore we need to use $g \geqslant 2 + \log_2(2(n+1)/\Delta)$ additional guard bits. This also takes into consideration the final rounding to the output format.

IV. Examples, Implementation and Discussion

In this section, we consider fractions with fixed-point coefficients of 24, 32, and 48 bits: these coefficients will be of the form $i/2^w$, with $-2^w \leqslant i \leqslant 2^w$, where $w = 24, 32, 48$. The target approximation error in each case is 2^{-w}, i.e., $\sim 5.96 \cdot 10^{-8}, 2.33 \cdot 10^{-10}, 3.55 \cdot 10^{-15}$ respectively.

Examples. All the examples are defined in the first column of Table I. When choosing them we considered:

- Functions useful in practical applications. The exponential function (Example 2) is a ubiquitous one. Functions of the form $\log_2(1+2^{\pm kx})$ (as the one of Example 3) are useful when implementing logarithmic number systems.The erf functions (Example 4) is useful in probability and statistics, while the Bessel function J_0 (Example 5) has many applications in physics.
- Functions that illustrate the various cases that can occur: polynomials are a better choice (Example 3); rational approximation is better (Examples 1, 2, Example 4 if $r \leqslant 8$ and Example 5 if $r = 2$). We also include instances where the approximating E-fractions are very different from the minimax, unconstrained, rational approximations with similar degrees in the numerator and denominator (Examples 1 and 2).

All the examples start with a radix 2 setting after which higher values of r are considered. Table I displays approximation errors in the real coefficient and fixed-point coefficient E-fraction cases. Notice in particular the lattice-based approximation errors, which are generally much better than the naive rounding ones. We also give some complementary comments.

Example 1. The type $(4, 4)$ rational minimax unconstrained approximation error is $4.59 \cdot 10^{-16}$, around 5 orders of magnitude smaller than the E-fraction error. A similar difference happens in case of Example 2, where the type $(3, 3)$ unconstrained minimax approximation has error $2.26 \cdot 10^{-16}$.

Example 2. In this case, we are actually working with a rescaled input and are equivalently approximating $\exp(2x), x \in [0, 7/128]$. Also, for $r = 8$, the real coefficient E-fraction is the same as the E-polynomial one (the magnitude constraint for the denominator coefficients is 0).

Example 3. Starting with $r = 8$, we have to scale both the argument x and the approximation domain by suitable powers of 2 for the E-method constraints to continue to hold (see end of Section II-A).

Example 4. As with the previous example, for $r = 16, 32$ we have to rescale the argument and interval to get a valid E-polynomial.

Example 5. By a change of variable, we are actually working with $J_0(2x - 1/16), x \in [0, 1/16]$. If we consider $r \geqslant 16$, the 48 bits used to represent the coefficients were not sufficient to produce an approximation with error below 2^{-48}.

Implementation. We have generated the corresponding circuits for each of the examples, and synthesized them. The target platform is a Xilinx Virtex6 device xc6vcx75t-2-ff484, and the toolchain used is ISE 14.7. The resulting circuit descriptions are in an easily readable and portable VHDL. For each of the examples we have compared against a state of the art implementation created using the FloPoCo generator, as presented in [9]. FloPoCo [24] is an open-source arithmetic core generator and library for FPGAs. It is, to the best of our knowledge, one of the only alternatives capable of producing the functions chose for comparison. Table II presents the results.

At the top of Table II, for Example 1, we show the flexibility of the generator: it can easily accommodate for various latencies and target frequencies. The examples show how the frequency can be scaled from around 100MHz to 300MHz, at the expense of a deeper pipeline and an increased number of registers.

Also, the number of registers approximately doubles each time the circuit's period is reduced by a factor 2. This very predictable behavior should help the end user make an accept-

TABLE I
APPROXIMATION ERRORS IN THE REAL COEFFICIENT AND FIXED-POINT COEFFICIENT E-FRACTION CASES

	Function Type of error	Δ	r	(μ,ν)	w	Real coefficient E-fraction error	Naive rounding error	Lattice-based error
Ex. 1	$\sqrt{1+(9x/2)^4}, x\in[0,1/32]$ absolute	$\frac{1}{8}$	2	$(4,4)$	32	$5.22\cdot10^{-11}$	$1.11\cdot10^{-9}$	$5.71\cdot10^{-11}$
			4			$6.32\cdot10^{-11}$	$4.93\cdot10^{-10}$	$7\cdot10^{-11}$
			8			$8.25\cdot10^{-11}$	$1.78\cdot10^{-9}$	$1.11\cdot10^{-10}$
Ex. 2	$\exp(x), x\in[0,7/64]$ relative	$\frac{1}{8}$	2	$(3,3)$	32	$1.64\cdot10^{-10}$	$3.24\cdot10^{-10}$	$1.94\cdot10^{-10}$
			4	$(4,4)$		10^{-12}	$1.91\cdot10^{-11}$	$1.11\cdot10^{-12}$
			8	$(5,0)$		$1.16\cdot10^{-12}$	$1.74\cdot10^{-11}$	$1.39\cdot10^{-12}$
Ex. 3	$\log_2(1+2^{-16x}), x\in[0,1/16]$ absolute	$\frac{1}{2}$	2	$(5,5)$	24	$1.98\cdot10^{-8}$	$4.37\cdot10^{-7}$	$2.33\cdot10^{-8}$
			4,8,16	$(5,0)$		$2.04\cdot10^{-8}$	$4.22\cdot10^{-7}$	$2.64\cdot10^{-8}$
Ex. 4	$\mathrm{erf}(x), x\in[0,1/32]$ absolute	$\frac{1}{8}$	2	$(4,4)$	48	$2.92\cdot10^{-17}$	$1.67\cdot10^{-16}$	$3.43\cdot10^{-17}$
			4	$(4,4)$		$3.44\cdot10^{-17}$	$1.13\cdot10^{-16}$	$4.23\cdot10^{-17}$
			8,16,32	$(5,0)$		$1.34\cdot10^{-15}$	$2.7\cdot10^{-15}$	$1.64\cdot10^{-15}$
Ex. 5	$J_0(x), x\in[-1/16,1/16]$ relative	$\frac{1}{2}$	2	$(4,4)$	48	$2.15\cdot10^{-17}$	$2.49\cdot10^{-15}$	$2.37\cdot10^{-15}$
			4,8	$(6,0)$		$1.23\cdot10^{-17}$	$2.53\cdot10^{-15}$	$2.37\cdot10^{-15}$

able trade-off in terms of performance to required resources. The frequency cap of 300MHz is not something inherent to the E-method algorithm, neither to the implementation; instead it comes from current limitations of the bitheap framework inside the FloPoCo generator. We expect that once this issue is fixed, our implementations will be capable of reaching much higher target frequencies.

Discussion. Examples 1 and 2 illustrate that for functions where classical polynomial approximation techniques, like the one used in FloPoCo, manage to find solutions of a reasonably small degree, the ensuing architectures also manage to be highly efficient. This shows, as implementations produced by FloPoCo (with polynomials of degree 6 in both cases) are twice (if not more) as efficient in terms of resources.

However, this is no longer the case when E-fractions can provide a better approximation. This is reflected by Examples 3 to 5, where we obtain a more efficient solution, by quite a large margin in some cases.

For Example 5, Table II does not present any data for the FloPoCo implementation as they do not currently support this type of function.

There are a few remarks to be made regarding the use of a higher radix in the implementations of the E-method. Example 4 is an indication that the overall delay of the architecture reaches a point where it can no longer benefit from increasing the radix. The lines of Table II marked with an asterisk were generated with an alternative implementation for the CUs, which uses multipliers for computing the $d_{i+1}^{(j-1)}\cdot x$ products. This is due to the exponential increase of the size of multiplexers with the increase of the radix, while the equivalent multiplier only increases linearly. Therefore, there is a crossover point from which it is best to use this version of the architecture, usually at radix 8 or 16. Finally, the effects of truncating the last iterations become the most obvious when the maximum degree n is close to the number of required iterations m in radix r. This effect can be observed for Example 3 and 4, where there is a considerable drop in resource consumption between the use of radix 8 and 16, and 16 and 32, respectively.

V. EXAMPLES, IMPLEMENTATION AND DISCUSSION

In this section, we consider fractions with fixed-point coefficients of 24, 32, and 48 bits : these coefficients will be of the form $i/2^w$, with $-2^w \leqslant i \leqslant 2^w$, where $w = 24, 32, 48$.

The choice of the examples was influenced by the following:

- Functions useful in practical applications. The exponential function (Example 2) is a ubiquitous one. Functions of the form $\log_2(1+2^{\pm kx})$ (as the one of Example 3) are useful when implementing logarithmic number systems: in such systems, numbers are represented by their logarithms, and addition is implemented using the formula
$$\log_2(a+b) = \log_2(a) + \log_2\left(1+2^{\log_2(b)-\log_2(a)}\right).$$
Function erf (Example 4) is useful in Probability and Statistics, and Bessel function J_0 (Example 5) has many applications in Physics.
- Examples that illuminate the various cases that can occur: polynomials are a better choice (Example 3); rational approximation is better (Examples 1, 2, and Example 4 if $r \leqslant 8$); including a case for which the approximating E-fraction is very different from the minimax, unconstrained, rational approximation with similar degrees in the numerator and denominator (Example 1).

All the examples start with a radix 2 setting after which higher values of r are considered.

Example 1. Consider function $x \in [0,1/32] \mapsto \sqrt{1+(9x/2)^4}$, approximated by a $(4,4)$ rational function. The E-method parameters are $\Delta = 1/8, \alpha = 7/32$ and $\xi = 9/16$. For a degree-4 E-polynomial approximation, the error is $3.41\cdot10^{-10}$. The real coefficient E-fraction has error $5.22\cdot10^{-11}$. For 32-bit coefficients, our lattice-based approach gives an E-fraction with error $5.71\cdot10^{-11}$. Without denominator coefficient constraint, the minimax rational approximation error is $4.59\cdot10^{-16}$.

TABLE II
SYNTHESIS RESULTS FOR A XILINX VIRTEX6 DEVICE

Design	Approach	radix	Resources		Performance
			LUT	reg.	cycles@period(ns)
Ex. 1	Ours	2	7,880	0	1@94.3
			7,966	1,523	11@9.6
			7,299	2,689	17@5.7
			6,786	5,202	36@3.7
		4	4,871	0	1@57.9
			4,768	988	7@12.3
			4,600	1,583	11@6.9
			4,853	3,106	22@3.8
		8	4,210	0	1@44.4
			3,875*	0	1@62.2*
			5,307*	309	5@18.4*
			5,184*	499	8@10.4*
			4,707*	1,027	15@5.8*
	FloPoCo	–	994	0	1@29.5
			1,032	138	7@6.7
			1,147	335	19@5.3
Ex. 2	Ours	2	6,820	0	1@88.5
		4	6,356	0	1@68.0
		8	5,042	0	1@39.0
	FloPoCo	–	3,024	0	1@41.1
Ex. 3	Ours	2	2,944	0	1@67.0
		4	2,742	0	1@35.1
		8	2,582	0	1@33.1
		16	2,856	0	1@31.2
			1,565*	0	1@29.0*
	FloPoCo	–	3,622	0	1@55.7
Ex. 4	Ours	2	19,564	0	1@139.6
		4	23,052	0	1@92.5
			21,179*	0	1@131.5*
		8	15,388*	0	1@250.7*
		16	12,878*	0	1@76.9*
		32	3,909*	0	1@86.7*
	FloPoCo	–	20,494	0	1@139.9
Ex. 5	Ours	2	19,423	0	1@368.1
		4	13,642	0	1@70.3
		8	18,653	0	1@58.6
	FloPoCo	–	–	–	–

If $r = 4$, the real coefficient E-fraction approximation error increases slightly to $6.32 \cdot 10^{-11}$, whereas the lattice-based error is $7 \cdot 10^{-11}$ (rounding error is $4.93 \cdot 10^{-10}$). For radix $r = 8$, the real coefficient E-fraction approximation error is $8.25 \cdot 10^{-11}$, whereas the lattice-based error is $1.11 \cdot 10^{-10}$ (rounding error is $1.78 \cdot 10^{-9}$). For each r, we take $\alpha = (1 - \Delta)/(2r)$.

Example 2. Consider $\exp(x)$ on $[0, 7/64]$, implemented as $\exp(2x)$ on $[0, 7/128]$, with the same parameters as Example 1, but with respect to the relative error. We get, for $\nu = 3$, a polynomial minimax error of $4.66 \cdot 10^{-8}$, while the type $(3, 3)$ real coefficient E-fraction has error $1.64 \cdot 10^{-10}$. Taking 32-bit fixed-point coefficients with the method of Section II-B gives an error $1.94 \cdot 10^{-10}$ (the rounding error is $3.24 \cdot 10^{-10}$). The minimax rational approximation error is smaller, $2.26 \cdot 10^{-16}$.

For radix $r = 4$, the denominator coefficients in an optimized type $(3, 3)$ real coefficient E-fraction get saturated and the corresponding approximation error becomes too large to satisfy the 2^{-32} accuracy requirement. A type $(4, 4)$ approximation is more than enough: the real coefficient E-fraction error is $1 \cdot 10^{-12}$, with the lattice-based error only $1.13 \cdot 10^{-12}$ (rounding error $1.91 \cdot 10^{-11}$). The polynomial $\nu = 4$ approximation barely misses the target: we have minimax error $2.54 \cdot 10^{-10}$ (slightly larger than 2^{-32}). With radix $r = 8$ the real coefficient E-fraction is the same as the E-polynomial one (the magnitude constraint for the denominator coefficients is 0). This means that a degree $\nu = 5$ polynomial is needed. The real coefficient E-polynomial error is $1.16 \cdot 10^{-12}$, whereas the lattice-based error is $1.39 \cdot 10^{-12}$ (rounding gives the error $1.74 \cdot 10^{-11}$).

Example 3. We deal with $x \in [0, 1/16] \mapsto \log_2\left(1 + 2^{-16x}\right)$ and consider the absolute error approximation. The E-method parameters are $\Delta = 1/2, \alpha = 1/8$ and $\xi = 3/4$. The polynomial Remez algorithm, for target degree $\nu = 5$, gives an error $2.04 \cdot 10^{-8}$. The lattice-based error when targeting 24-bit coefficients is $2.64 \cdot 10^{-8}$ (rounding error is $4.22 \cdot 10^{-7}$).

The type $(5, 5)$ simple real coefficient E-fraction gives the error $2.05 \cdot 10^{-8}$ (and $2.44 \cdot 10^{-8}$ for the lattice-based discretization), so not a lot to gain by using it.

For $r = 4,\ 8,\ 16$, E-polynomials are still more interesting. Starting with $r = 8$, we have to scale both the argument x and the approximation domain by suitable powers of 2 for the E-method constraints to continue to hold [23].

Example 4. Consider $x \in [0, 1/32] \mapsto \mathrm{erf}(x)$ with $\Delta = 1/8$. If we target 48-bit coefficients and absolute approximation error $< 2^{-48}$, we can take a type $(4, 4)$ E-fraction. The real coefficient approximation error is then $2.92 \cdot 10^{-17}$, while the lattice-based optimized one is $3.43 \cdot 10^{-17}$ (the rounding error increases here to $1.67 \cdot 10^{-16}$). To obtain a similar error with an E-polynomial we need at least degree $\nu = 6$, which gives a real coefficient error $9.53 \cdot 10^{-17}$ ($\nu = 5$ suffices though for the error to fall below 2^{-48}).

For $r = 4$, the real coefficient E-fraction error is $3.44 \cdot 10^{-17}$, while the lattice-based one is $4.23 \cdot 10^{-17}$ (rounding error $1.13 \cdot 10^{-16}$). Starting from $r = 8$, degree $\nu = 5$ polynomials are more interesting than E-fractions. The real coefficient error is $1.34 \cdot 10^{-15}$, whereas the lattice-based one is $1.64 \cdot 10^{-15}$ (rounding error is $2.7 \cdot 10^{-15}$). For $r = 16, 32$ we have to rescale the argument and interval to get a valid E-polynomial.

Example 5. Consider $x \in [0, 1/16] \mapsto J_0(2x - 1/16)$, where J_0 is a Bessel function of the first kind and $\Delta = 1/2$. Again, take 48-bit coefficients. We are constructing a relative error approximation. If we are using polynomials, we need a degree $\nu = 6$ approximation to get a lattice-based error $2.37 \cdot 10^{-15}$. By contrast, a rational E-fraction of type $(4, 4)$ is sufficient to get the same lattice-based error $2.37 \cdot 10^{-15}$ (the type $\nu = 4$ polynomial error is only $8.08 \cdot 10^{-13}$).

For $r \geqslant 4$, rational approximations are not more interesting than polynomial ones, so we consider the degree $\nu = 6$ approximation. The real coefficient error is $1.23 \cdot 10^{-17}$, whereas the lattice-based error is $2.37 \cdot 10^{-15}$ (rounding error

is $2.53 \cdot 10^{-15}$). For $r \geqslant 16$, the 48 bits used to represent the coefficients were not sufficient to produce an approximation with error below 2^{-48}.

Implementation. We have generated the corresponding circuits for each of the examples, and synthesized them. The target platform is a Xilinx Virtex6 device xc6vcx75t-2-ff484, and the toolchain used is ISE 14.7. The resulting circuit descriptions are in an easily readable and portable VHDL. For each of the examples we have compared against a state of the art implementation created using the FloPoCo generator, as presented in [9]. Table II presents the results.

At the top of Table II, for Example 1, we show the flexibility of the generator: it can easily accommodate for various latencies and target frequencies. The examples show how the frequency can be scaled from around 100MHz to 300MHz, at the expense of a deeper pipeline and an increased number of registers.

Also, the number of registers approximately doubles each time the circuit's period is reduced by a factor 2. This very predictable behavior should help the end user make an acceptable trade-off in terms of performance to required resources. The frequency cap of 300MHz is not something inherent to the E-method algorithm, neither to the implementation; instead it comes from current limitations of the bitheap framework inside the FloPoCo generator. We expect that once this issue is fixed, our implementations will be capable of reaching much higher target frequencies.

Examples 1 and 2 illustrate that for functions where classical polynomial approximation techniques, like the one used in FloPoCo, manage to find solutions of a reasonably small degree, the ensuing architectures also manage to be highly efficient. This shows, as implementations produced by FloPoCo (with polynomials of degree 6 in both cases) are twice (if not more) as efficient in terms of resources.

However, this is no longer the case where E-fractions can provide a better approximation. This is reflected by Examples 3 to 5, where we obtain a more efficient solution, by quite a large margin in some cases.

For Example 5, Table II does not present any data for the FloPoCo implementation as they do not currently support this type of function.

There are a few remarks to be made regarding the use of a higher radix in the implementations of the E-method. Example 4 is an indication that the overall delay of the architecture reaches a point where it can no longer benefit from increasing the radix. The lines of Table II marked with an asterisk were generated with an alternative implementation for the CUs, which uses multipliers for computing the $d_{i+1}^{(j-1)} \cdot x$ products. This is due to the exponential increase of the size of multiplexers with the increase of the radix, while the equivalent multiplier only increases linearly. Therefore, there is a crossover point from which it is best to use this version of the architecture, usually at radix 8 or 16. Finally, the effects of truncating the last iterations become the most obvious when the maximum degree n is close to the number of required iterations m in radix r. This effect can be observed for Example 3 and 4, where there is a considerable drop in resource consumption between the use of radix 8 and 16, and 16 and 32, respectively.

VI. Summary and Conclusions

A high throughput system for the evaluation of functions by polynomials or rational functions using simple and uniform hardware is presented. The evaluation is performed using the unfolded version of the E-method, with a latency proportional to the precision. An effective computation of the coefficients of the approximations is given and the best strategies (choice of polynomial vs rational approximation, radix of the iterations) investigated. Designs using a circuit generator for the E-method inside the FloPoCo framework are developed and implemented using FPGAs for five different functions of practical interest, using various radices. The results paint a clear picture: the E-method is generally more efficient as soon as the rational approximation is significantly more efficient than the polynomial one. From a hardware standpoint, the results show it is desirable to use the E-method with high radices, usually at least 8. The method also becomes efficient when we manage to find a balance between the maximum degree n of the polynomial or E-fraction and the number of iterations required for converging to a correct result, which we can control by varying the radix. A complete open-source implementation of our approach will soon be available online.

Acknowledgments

This work was partly supported by the FastRelax project of the French Agence Nationale de la Recherche, EPSRC (UK) under grants EP/K034448/1 and EP/P010040/1, the Royal Academy of Engineering and Imagination Technologies.

References

[1] J. E. Volder, "The CORDIC computing technique," *IRE Transactions on Electronic Computers*, vol. EC-8, no. 3, pp. 330–334, 1959, reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[2] W. F. Wong and E. Goto, "Fast hardware-based algorithms for elementary function computations using rectangular multipliers," *IEEE Trans. Comput.*, vol. 43, no. 3, pp. 278–294, Mar. 1994.

[3] D. D. Sarma and D. W. Matula, "Faithful bipartite ROM reciprocal tables," in *Proceedings of the 12th IEEE Symposium on Computer Arithmetic (ARITH-12)*, Knowles and McAllister, Eds. IEEE Computer Society Press, Los Alamitos, CA, Jun. 1995, pp. 17–28.

[4] M. J. Schulte and J. E. Stine, "Approximating elementary functions with symmetric bipartite tables," *IEEE Trans. Comput.*, vol. 48, no. 8, pp. 842–847, Aug. 1999.

[5] F. de Dinechin and A. Tisserand, "Multipartite table methods," *IEEE Trans. Comput.*, vol. 54, no. 3, pp. 319–330, Mar. 2005.

[6] M. D. Ercegovac, "A general method for evaluation of functions and computation in a digital computer," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, Urbana-Champaign, IL, 1975.

[7] ——, "A general hardware-oriented method for evaluation of functions and computations in a digital computer," *IEEE Trans. Comput.*, vol. C-26, no. 7, pp. 667–680, 1977.

[8] N. Brisebarre, S. Chevillard, M. D. Ercegovac, J.-M. Muller, and S. Torres, "An Efficient Method for Evaluating Polynomial and Rational Function Approximations," in *19th IEEE Conference on Application-specific Systems, Architectures and Processors. ASAP'2008*, Leuven (Belgium), 2–4 July 2008, pp. 233–238.

[9] F. de Dinechin, "On fixed-point hardware polynomials," Tech. Rep., Oct. 2015, working paper or preprint. [Online]. Available: https://hal.inria.fr/hal-01214739

[10] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann Publishers, San Francisco, CA, 2004.

[11] E. W. Cheney, *Introduction to Approximation Theory*, 2nd ed. AMS Chelsea Publishing. Providence, RI, 1982.

[12] M. J. D. Powell, *Approximation theory and methods*. Cambridge University Press, 1981.

[13] R. Reemtsen, "Modifications of the first Remez algorithm," *SIAM J. Numer. Anal.*, vol. 27, no. 2, pp. 507–518, 1990.

[14] E. W. Cheney and H. L. Loeb, "Two new algorithms for rational approximation," *Numer. Math.*, vol. 3, no. 1, pp. 72–75, 1961.

[15] S. N. Dua and H. L. Loeb, "Further remarks on the differential correction algorithm," *SIAM J. Numer. Anal.*, vol. 10, no. 1, pp. 123–126, 1973.

[16] I. Barrodale, M. J. D. Powell, and F. K. Roberts, "The differential correction algorithm for rational ℓ_∞-approximation," *SIAM J. Numer. Anal.*, vol. 9, no. 3, pp. 493–504, 1972.

[17] D. Dudgeon, "Recursive filter design using differential correction," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 22, no. 6, pp. 443–448, 1974.

[18] 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, 2010.

[19] N. Brisebarre and S. Chevillard, "Efficient polynomial L^∞ approximations," in *ARITH '07: Proceedings of the 18th IEEE Symposium on Computer Arithmetic, Montpellier, France*. IEEE Computer Society, 2007, pp. 169–176.

[20] P. Q. Nguyen and B. Vallée, Eds., *The LLL Algorithm - Survey and Applications*, ser. Information Security and Cryptography. Springer, 2010. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02295-1

[21] A. K. Lenstra, H. W. Lenstra, and L. Lovász, "Factoring polynomials with rational coefficients," *Math. Ann.*, vol. 261, no. 4, pp. 515–534, 1982.

[22] L. Babai, "On Lovász' lattice reduction and the nearest lattice point problem," *Combinatorica*, vol. 6, no. 1, pp. 1–13, 1986.

[23] N. Brisebarre and J.-M. Muller, "Functions approximable by E-fractions," in *Proc. 38th IEEE Conference on Signals, Systems and Computers*. IEEE, Nov. 2004.

[24] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Des. Test Comput.*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[25] K. Chapman, "Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner)," *EDN magazine*, no. 10, p. 80, May 1993.

[26] M. J. Wirthlin, "Constant coefficient multiplication using look-up tables," *VLSI Signal Processing*, vol. 36, no. 1, pp. 7–15, 2004.

[27] A. Volkova, M. Istoan, F. De Dinechin, and T. Hilaire, "Towards Hardware IIR Filters Computing Just Right: Direct Form I Case Study," Tech. Rep., 2017. [Online]. Available: http://hal.upmc.fr/hal-01561052/

[28] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes, and B. Popa, "Arithmetic core generation using bit heaps," in *Proceedings of the 23rd Conference on Field-Programmable Logic and Applications*, Sep. 2013, pp. 1–8.