Table-based polynomials for fast hardware function evaluation

Jérémie Detrey, Florent de Dinechin
LIP, École Normale Supérieure de Lyon
46 allée d'Italie
69364 Lyon cedex 07, France
E-mail: {Jeremie.Detrey, Florent.de.Dinechin}@ens-lyon.fr

Abstract

Many general table-based methods for the evaluation in hardware of elementary functions have been published. The bipartite and multipartite methods implement a first-order approximation of the function using only table lookups and additions. Recently, a single-multiplier second-order method of similar inspiration has also been published. This paper extends such methods to approximations of arbitrary order, using adders, small multipliers, and very small ad-hoc powering units. We obtain implementations that are both smaller and faster than previously published approaches.

This paper also deals with the FPGA implementation of such methods. Previous work have consistently shown that increasing the approximation degree lead to not only smaller but also faster designs, as the reduction of the table size meant a reduction of its lookup time, which compensated for the addition and multiplication time. The experiments in this paper suggest that this still holds when going from order 2 to order 3, but no longer when using higher-order approximations, where a tradeoff appears.

1. Introduction

Many applications require the evaluation in hardware of a numerical function: Trigonometric functions for DSP algorithms, reciprocal and inverse square root for providing seed values to the Newton-Raphson or Goldschmidt algorithms for division and square root [10, 3, 2, 12], exponential and logarithm for some scientific computing applications or for the logarithm number system (LNS) [5], etc. When a compound function (such as $\log_2(1 + 2^x)$ for instance) is needed, it is often more efficient to implement it as one operator instead of a combination of successive operators for each function (here an exponential, an addition and a logarithm).

Specific methods exist for implementing most of the elementary functions. For example, the CORDIC algorithm and its derivatives implement trigonometric and exp/log functions. With some work, this is probably also true of most useful compound function (see for example the literature about LNS arithmetic for methods dedicated to evaluating $\log_2(1 + 2^x)$). However these specific methods usually have their constraints. For instance, the CORDIC derivative leads to small but slow operators. Besides they may require a lot of non-reusable work to get a functional implementation.

An alternative is to use a general implementation method which may be tailored easily to any function. The simplest of these methods is, of course, to tabulate all the values that the function takes in the needed discrete range. The drawback is then the hardware cost, as the size of the table increases exponentially with the size (in bits) of its input argument.

One may reduce this cost by using polynomial or piecewise polynomial approximations of the function, at the expense of one or more multipliers which increase the latency. See for example [3, 15, 9] for first-order (linear) approximations, [7, 13, 1] for order 2, and [8] for order 3, among others. The present paper may be considered as an improvement on these previous approaches: It generalizes them and minimizes the negative impact of the multipliers by a very careful examination of the architectural tradeoffs with respect to size, latency and precision.

This paper may also be considered as an extension of previous work on table-and-addition methods [2, 14, 11, 4] which use a first-order Taylor approximation of the function. In these methods, the product terms are themselves tabulated, leading to an architecture composed of table lookups and additions, and therefore very fast. Recently, we proposed a method allowing a second order approximation using only one small multiplier [6]. Here "small" means that its input and output sizes are much smaller than the input and output precision of the function to be evaluated. Both methods can be applied to any function, elementary or compound, that fulfills basic continuity requirements. This means that they lend themselves to the implementation of automatic operator generators: We have programs that take an arbitrary function with an input and output precision, and compute the optimal implementation of this function as a

Proceedings of the16th International Conference on Application-Specific Systems, Architecture and Processors (ASAP'05)
1063-6862/05 $20.00 © 2005 IEEE

hardware operator, according to some predefined optimality criterion. The size and speed of the operator depends on the input and output precision, but also on the function. These generators output circuit descriptions in the VHDL language, suitable for synthesis.

These methods may target Application-Specific Integrated Circuits (ASICs) or Field-Programmable Gate Arrays (FPGAs). The metrics of ASICs and FPGAs are very different: Tables may be implemented in various ways in both worlds, the same holds for arithmetic operators, and the relative size and speed of arithmetic and tables are different. For practical reasons we mostly studied FPGA implementation. In both cases, an operator generator will try and synthesize a few candidates to optimality before giving a definitive result.

With the FPGA metrics, an interesting result of previous work on table-based methods was that a multipartite implementation was a win-win situation compared to a simple table: Although the former has one table lookup and several additions on the critical path, it is faster than the latter which has only a table lookup. The reason is simply that the tables are so much reduced in the multipartite implementation that the lookups are much faster, which compensates the addition time. More surprising was the fact that going to second-order approximation and to architectures with multipliers on the critical path was again a win-win move [6]. This is a motivation to study higher-order methods in this paper. The other motivation is, of course, to obtain better implementations of hardware function evaluators.

This paper first presents in Section 2 a general framework for the hardware implementation of arbitrary polynomials. Methods using the Horner evaluation scheme have been studied, but their iterative nature leads to implementations with long latency. The approach studied here is to use a developed form of the polynomial, where each monomial is evaluated in parallel. Each monomial may then be implemented by multipliers and powering units, or table-based methods, or a combination of both. The philosophy is here to carry out a careful error computation, not only to guarantee faithful correct rounding of the result, but also to build blocks which are never more accurate than strictly needed, as exposed in Section 3. The architectures obtained are depicted in Section 4. A detailed error analysis is presented in Section 5. It gives explicit formulae for the various error terms, which can be used to implement rapid design exploration heuristics.Last, speed and area estimations of the operators are studied in Section 6 and compared to results obtained using other methods.

2. Presentation of the method

2.1. Function evaluation

The problem of hardware function evaluation can be expressed as follows: Given a function f defined on a finite input interval $\mathcal{I} \subset \mathbb{R}$, and two positive integers w_I and w_O which specify the length in bits of the input and output words respectively, build a hardware circuit which will compute an approximation $\tilde{f}$ of the function f on the interval $\mathcal{I}$. Without loss of generality, we can take $\mathcal{I} = [0; 1)$ and scale f such as $f(\mathcal{I}) = [0; 1)$. Therefore, we will write the input word X as $X = .x_1x_2 \dots x_{w_I}$, and similarly the output word $Y = \tilde{f}(X) = .y_1y_2 \dots y_{w_O}$. We also want our evaluation operator to guarantee the accuracy of the result. As rounding to nearest is impractical because of the table-maker's dilemma [10], we choose to ensure faithful rounding: $\epsilon = \max_{X\in\mathcal{I}} |f(X) - \tilde{f}(X)| < 2^{-w_O}$.

2.2. Piecewise polynomial approximation

The method we present here is based on a piecewise polynomial approximation: The input interval $\mathcal{I} = [0; 1)$ is regularly split in several sub-intervals $\mathcal{I}_i = [i \cdot 2^{-\alpha}; (i+1) \cdot 2^{-\alpha})$. These sub-intervals are addressed by the α most significant bits of X, and we approximate f on each of them by a degree n polynomial P_i. Each polynomial P_i is computed using a minimax scheme [10], and therefore minimizes the maximum error entailed by this approximation.

We partition the input interval into sub-intervals of the same size. In [9] Lee *et al.* have developed a method for hierarchical segmentation of the input interval which improves the quality of the operators for some functions. Our work could easily be adapted to such a partition.

As the sub-intervals are addressed by α bits from X, we can split the input word in two sub-words $A = .a_1a_2 \dots a_\alpha$ and $B = .b_1b_2 \dots b_\beta$ of length α and $\beta = w_I - \alpha$ respectively. This gives: $X = A + B \cdot 2^{-\alpha}$.

Thus, to compute $\tilde{f}(X)$, we need to evaluate the polynomial $P_A(B \cdot 2^{-\alpha})$, which we will write $P(A, B \cdot 2^{-\alpha})$ to simplify the notations. We expand the polynomial to obtain:

$$P(A, B \cdot 2^{-\alpha}) = K_n(A) \cdot B^n \cdot 2^{-n\alpha} + \dots + K_1(A) \cdot B \cdot 2^{-\alpha} + K_0(A). \quad (1)$$

The method presented in this article evaluates separately each term (or monomial) $T_k(A, B) = K_k(A) \cdot B^k \cdot 2^{-k\alpha}$ for k ranging from 0 to n. A final summation of all the terms then effectively computes the approximated function $\tilde{f}(X)$.

2.3. Computing the terms

There are several methods to evaluate a term $T_k(A, B)$, and we chose to implement two of them in our work, as described in the following paragraphs.

2.3.1. Simple ROM

The first and the simplest method is to extensively compute all the possible values for the term and tabulate these values in a table addressed by A and B, or only by A for $T_0(A, B) = K_0(A)$.

2.3.2. Power-and-multiply

A second method consists in first computing B^k by using a powering unit, and then multiplying the result by $K_k(A)$.

Yet several implementation choices remain. The powering unit can either be a simple table addressed by B where all the possible values of B^k are stored, or a specialized *ad-hoc* unit which first generates then adds all the partial products required to compute B^k (such as in [13] for $k = 2$).

Moreover, the product of B^k by $K_k(A)$ can be spread on several multipliers by splitting the word $B^k = .p_1p_2 \dots p_{k\beta}$, which is of length $k\beta$, in m_k sub-words $S_{k,j}$, as in the multipartite method [4]. Spreading the product will allow us to optimize separately each multiplier as detailed in Section 3.2.

We get $B^k = S_{k,1} + S_{k,2} \cdot 2^{-\rho_{k,2}} + \ldots + S_{k,m} \cdot 2^{-\rho_{k,m_k}}$, where $S_{k,j} = .p_{\rho_{k,j}+1}p_{\rho_{k,j}+2} \dots p_{\rho_{k,j}+\sigma_{k,j}}$, for j ranging from 1 to m_k, is the sub-word of B^k starting at bit $\rho_{k,j}$ and of length $\sigma_{k,j}$. As it is a partition, we also have the natural conditions on the $\rho_{k,j}$'s and the $\sigma_{k,j}$'s: $\rho_{k,1} = 0$, $\rho_{k,j+1} = \rho_{k,j} + \sigma_{k,j}$ for $1 \leq j < m_k$, and $\sum_{j=1}^{m_k} \sigma_{k,j} = \rho_{k,m_k} + \sigma_{k,m_k} = k\beta$.

We can therefore rewrite the k-th term as :

$$T_k(A, B) = \left(\sum_{j=1}^{m_k} K_k(A) \cdot S_{k,j} \cdot 2^{-\rho_{k,j}} \right) \cdot 2^{-k\alpha}. \quad (2)$$

Finally, another choice raised by this method is, for each product $Q_{k,j}(A, S_{k,j}) = K_k(A) \cdot S_{k,j}$, whether to use a table addressed by A and a multiplier, or a single but larger table addressed by A and S_j. Here the $m_{k:M}$ first products will be implemented with multipliers, whereas the $m_{k:T} = m_k - m_{k:M}$ last ones will have their values tabulated. This is motivated by the error analysis below.

2.4. Exploiting symmetry

A change of variable in Eq. 1 gives a new expression for the polynomial approximation $P(A, B \cdot 2^{-\alpha})$, such that all the terms are symmetric with respect to the middle of the sub-interval $\mathcal{I}(A)$: $P(A, B \cdot 2^{-\alpha}) = K'_n(A) \cdot (B - \Delta)^n \cdot 2^{-n\alpha} + \ldots + K'_1(A) \cdot (B - \Delta) \cdot 2^{-\alpha} + K'_0(A)$, where $\Delta = \frac{1}{2}(1 - 2^{-\beta})$. This transformation allows us to use a trick from Schulte and Stine [14] to compute the terms only on one half of the sub-interval and deduce the values for the other half by symmetry at the expense of a few XOR gates.

Remark : To avoid overloading too much the notations, we will continue to write the terms $T_k(A, B) = K_k(A) \cdot B^k \cdot 2^{-k\alpha}$, even when symmetry is implied.

3. Decreasing accuracy

In this method so far, the only error is entailed by the initial polynomial approximation of the function. However, we can see from Eq. 1 that, because of the different power-of-two factors, the terms do not have the same weight in the final addition and thus some of them are computed with too much accuracy when compared to others.

In order to simplify the tables, and consequently gain in area and latency for our operator, we can therefore decrease the accuracy of those terms that are relatively too accurate.

3.1. Terms as simple ROMs

When considering a term $T_k(A, B)$ implemented as a table addressed by A and B, the idea is to decrease the size of the address word. Decreasing the size of A by using only its α_k most significant bits to address the table means that we will use a same value of the coefficient K_k for $2^{\alpha-\alpha_k}$ consecutive intervals. Decreasing the size of B by using only its β_k most significant bits to address the table means that less values will be computed for each interval.

We can therefore refine the splitting of the input word and use only $A_k = .a_1a_2 \dots a_{\alpha_k}$ and $B_k = .b_1b_2 \dots b_{\beta_k}$ to address the table of term $T_k(A_k, B_k)$.

3.2. Terms as power-and-multiply units

In this case, the first idea is also to decrease the size of A and B. But here, as the product is spread over m_k multiplications, we have $m_{k:M}$ tables addressed by A, and $m_{k:T}$ tables addressed by A and $S_{k,j}$ (with the notations of 2.3.2). Once again, according to Eq. 2, those tables have different relative accuracies due to the $2^{-\rho_{k,j}}$ factors. We can therefore address them with sub-words of A of different sizes: The table used by $Q_{k,j}(A, S_{k,j}) = K_k(A) \cdot S_{k,j}$ will be addressed with only the $\alpha_{k,j}$ most significant bits of A.

Yet, from Eq. 2 we can see that the relative weight of $Q_{k,j}$ decreases as j increases. This gives the following constraint on the $\alpha_{k,j}$'s: $\forall j, j' \in [1; m]$, if $j < j'$ then $\alpha_{k,j} \geq \alpha_{k,j'}$. Moreover, we can also use $B_k = .b_1b_2 \dots b_{\beta_k}$ instead of B.Thus, the length of B_k^k will be only $k\beta_k$, which also implies smaller $S_{k,j}$'s. In fact, we can be even more general and suppose that the powering unit will generate only the λ_k most significant bits of B_k^k (with $\lambda_k \leq k\beta_k$).

This way, the $S_{k,j}$'s are much smaller, and consequently so are the product (both multiplier-based and table-based) units.

3.3. A few words about the *ad-hoc* powering units

If generating only λ_k bits of B_k^k is not a problem for the table-based powering units, the *ad-hoc* powering units, on the other hand, will entail a larger error if only the partial products of weight greater than λ_k are computed then added. To solve this problem without having the unit to compute all the $k\beta_k$ bits of B_k^k, we introduce another parameter μ_k which specifies the minimal weight of the partial products considered internally by the operator, before truncating the result to λ_k bits.

4. Architecture

The overall architecture of the operators designed by the proposed method is quite simple, as it is directly derived from Eq. 1: All the terms T_k are computed in parallel and then added.

Still, a few points have to be detailed. First, it is obvious that the order 0 term T_0 does not depend on B, and therefore will be implemented as a simple ROM. Concerning the term T_n of degree n, one can notice that the accuracy required for this term is very low, due to the $2^{-n\alpha}$ factor. We can then decrease α_n and β_n to only a few bits, and therefore implement also this term with a simple ROM. The same argument sometimes holds for lower order terms such as T_{n-1}. On the other hand, the other terms need to be computed with a larger accuracy, and will usually be implemented with slower but smaller power-and-multiply units.

4.1. Term as a simple ROM

The architecture for evaluating a term T_k using a simple ROM is also quite straightforward. The most significant bit b_1 of the input word B_k selects if B_k is in the first or the second half of the sub-interval $\mathcal{I}(A)$. A row of XOR gates is used to compute the symmetric of B_k. The table lookup is addressed by the α_k bits of A_k, and the $\beta_k - 1$ bits $B'_k = b'_2 b'_3 \dots b'_{\beta_k}$ from the XOR gates. If k is odd, a last row of XOR gates controlled by b_1 computes if necessary the opposite of the value given by the ROM. If k is even, we do not need these XOR gates.

4.2. Term as a power-and-multiply unit

The architecture of a power-and-multiply unit is given in Fig. 1. As for table-based terms, the most significant bit b_1 of B_k controls a row of XOR gates used to take the symmetric value of B_k. The resulting $\beta_k - 1$ bits $B'_k = b'_2 b'_3 \dots b'_{\beta_k}$ are then given to the powering unit, which outputs the λ_k most significant bits of B'^k_k. This word is split in m_k subwords. Each of these words $S_{k,j}$ is then multiplied by $K_k(A_{k,j})$, either using a normal multiplier or a lookup table. In both cases, we again exploit the symmetry of the product, and use some rows of XOR gates. Note that the last row of XORs is controlled by both b_i and $p_{\rho_{k,j}}$ when k is odd.

4.3. Table-based powering unit

A table-based powering unit is simply a lookup table, addressed by the $\beta_k - 1$ bits of B'_k, which contains the λ_k most significant bits of B'^k_k.

4.4. *Ad-hoc* powering unit

The architecture of an *ad-hoc* powering unit is straightforward. The first part of the operator generates all the partial products that are required to compute the μ_k most significant bits of B'^k_k. Then, these partial products are added, and finally the result is truncated to λ_k bits.

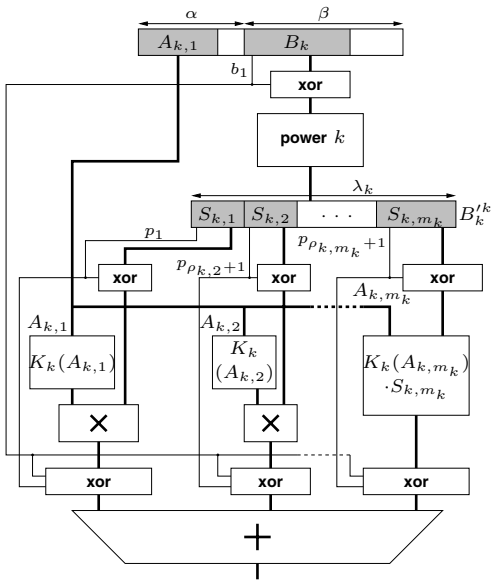

Figure 1. Architecture of the term T_k implemented as a power-and-multiply unit.

5. Error analysis

In this section we briefly describe how to keep track of all the errors entailed by the presented method, and therefore how to guarantee faithful rounding for the final result.

This method can be adapted to any error bound $\epsilon_{\max} > 2^{-w_O-1}$, but in this paper we only consider the case of $\epsilon_{\max} = 2^{-w_O}$. For readability we do not detail all the equations involved in this error analysis. The interested reader will find all the details in the reference implementation available from `http://www.ens-lyon.fr/LIP/Arenaire/`.

5.1. Polynomial approximation: ϵ_{poly}

The polynomial approximation of the function on each sub-interval yields an error bounded by ϵ_{poly}. The Remez algorithm [10] that we use to compute the minimax polynomials gives us the value of ϵ_{poly} for each sub-interval $\mathcal{I}(A)$.

5.2. Decreasing accuracy: $\epsilon_{\text{method}} = \epsilon_{\text{tab}} + \epsilon_{\text{pow}}$

5.2.1. Reducing table input size (ϵ_{tab})

Reducing the number of bits used to address a table means in fact using a constant value for several entries of the table.

For instance, considering a term $T_k(A, B) = K_k(A) \cdot B^k \cdot 2^{-k\alpha}$ implemented as a ROM, decreasing the word

Proceedings of the16th International Conference on Application-Specific Systems, Architecture and Processors (ASAP'05)
1063-6862/05 $20.00 © 2005 IEEE

length of A to α_k bits means that the same value of the coefficient K_k will be used for $2^{\alpha-\alpha_k}$ consecutive sub-intervals. To minimize the error, we can use for this value $K_k(A_k)$ the average of the extremal values of $K_k(A)$ for the sub-intervals. The maximum error is thus half the distance between those extremal values. Similarly, reducing B to β_k bits means that a constant value of B^k will be used for $2^{\beta-\beta_k}$ successive values of B. Taking the average of the extremal values of B^k also yields the minimum error.

Applying this technique to all the tables of the architecture, it is possible to compute exactly the sum of these approximation errors, which we note ϵ_{tab}. Note that these are errors of the ideal real values that the tables should hold, before rounding which is considered below.

Remark: Although the symmetry trick allows us to use only $\beta_k - 1$ bits of B to address the table, it entails absolutely no additional error.

5.2.2. *Ad-hoc* powering units (ϵ_{pow})

Using only the β_k most significant bits of B when computing B^k produces a quantifiable error, as $B_k = B - .b_{\beta_k+1}b_{\beta_k+2}\dots b_\beta \cdot 2^{-\beta_k}$ and $0 \leq .b_{\beta_k+1}b_{\beta_k+2}\dots b_\beta < 1$. To center this error around 0, we add an implicit half-ulp (unit in the last place) to B_k before computing B_k^k.

Moreover, the error made when reducing the number of partial products taken into account in the computation of B_k^k can also be bounded in advance, as we already know the number and the weight of the partial products that are ignored. We can then compute the sum s of those partial products, as the error will be in the interval $[0; s]$. Adding $s/2$ to the sum of the partial products computed by the unit will center the error around 0 as much as possible. We note $\epsilon_{\mathrm{pow},k}$ this error for each *ad-hoc* powering unit in the design. The errors yielded by each *ad-hoc* powering unit are then suitably multiplied by the corresponding K_k coefficient, and added to obtain the error term $\epsilon_{\mathrm{pow}} = \sum_{k=2}^{n} \epsilon_{\mathrm{pow},k} \cdot \max_A K_k(A)$.

5.3. Rounding considerations: ϵ_{rt}, ϵ_{rf}

The tables cannot be filled with results rounded to the target precision w_O: Each table would entail a maximum error of 2^{-w_O-1}, exceeding the total error budget $\epsilon_{\max} = 2^{-w_O}$. This argument also applies to multipliers, whose result cannot be rounded to the target precision. We therefore need to extend the internal precision of our operator by g guard bits. The values stored in the tables will then be rounded to the precision $w_O + g$, thus yielding a maximum rounding error of 2^{-w_O-g-1}. Similarly, the result of the multipliers will be rounded to $w_O + g$ bits, by truncating and adding a half-ulp, to ensure here also a maximum error of 2^{-w_O-g-1}. The sum of all these errors is noted ϵ_{rt}.

The final summation is also performed on $w_O + g$ bits, and is then truncated to the target precision w_O. A trick by Das Sarma and Matula [2] allows us to bound this rounding error by $\epsilon_{\mathrm{rf}} = 2^{-w_O-1}(1 - 2^{-g})$.

5.4. Putting it all together

Summing all the errors described previously, we have the following constraint to ensure faithful rounding: $\epsilon = \epsilon_{\mathrm{poly}} + \epsilon_{\mathrm{method}} + \epsilon_{\mathrm{rt}} + \epsilon_{\mathrm{rf}} < \epsilon_{\max}$. Since we have explicit formulae for all the error terms, we can then expand the values of ϵ_{rt} and ϵ_{rf} to obtain an inequation that we can solve to find the smallest number of required guard bits g. In fact, as we have added the maximum errors for each term, the total error may be overestimated, and a smaller g could be enough. We therefore apply a simple trial-and-error method to find the smallest acceptable g.

6. Results

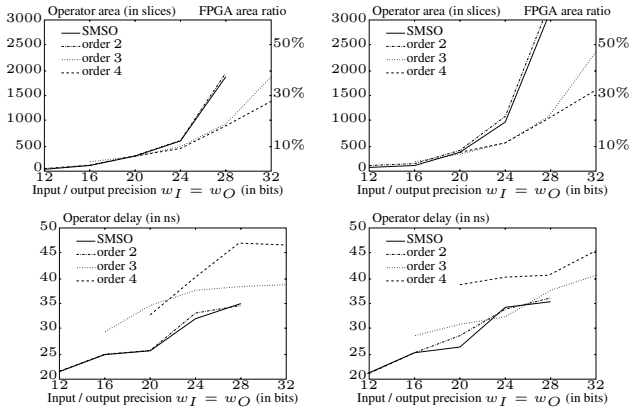

Figure 2. Operator area (top) and delay (bottom) for the $\sin x$ (left) and $\log_2(1+x)$ (right) functions.

This section presents synthesis results obtained for the presented method. We have successfully implemented order 2, 3 and 4 approximations of the functions $\sin x$ (on the interval $[0; \pi/4)$) and $\log_2(1 + x)$ (on $[0; 1)$), and we compare them for various precisions with the SMSO (single-multiplier second-order) method from [6] in terms of estimated area and delay of the operators in Fig. 2. SMSO has already been proven to be always better than the best available multipartite methods [6] and than the order 2 method from [13]. Those estimations were obtained using the Xilinx design suite ISE 5.2 along with the Xilinx synthesizer XST, for a Virtex-II XC2V-1000-4 FPGA. However, we chose not to implement multipliers using the Virtex-II 18×18 multipliers, to allow a more accurate comparison with other works.

7. Conclusion and future work

This article presents a general method, implemented in a functional tool, to build fast (combinatorial) implementations of arbitrary functions. The method leads to faster and smaller implementations than those previously published. As a rule of thumb, a second order approximation is optimal for precisions up to 16 bits and leads to operators which consume only a few percent of even the smallest FPGAs. For 24-bit precision, an order 3 approximation is optimal (order 4 is not smaller, but slower). For 32 bits, a precision out of reach of previous table-based methods, we have a tradeoff between order 3 and order 4, one being 30% smaller, the other being faster. Besides, all these architectures are very regular and easy to pipeline.

Future work

We now have to study the application of this method to ASIC synthesis, where the metrics are very different. Since the architectures involve the sum of many terms, the intermediate results should probably be expressed in carry-save notation, with only one fast adder in the architecture. Therefore, there is some work to do on the VHDL backend. We also should study the metrics (the relative cost of implementing a table as ROM or logic, the relative cost of a squarer unit, etc), and probably placement considerations.

Moreover, even though we have considered error bounds as constant on the interval $\mathcal{I}$ in section 5, the very same scheme can be applied when considering the bounds on $\epsilon_{\mathrm{method}}$ as piecewise polynomials, as this gives a much finer approximation of the method error bound. The idea here is to gradually decrease the accuracy of the tables when $\epsilon_{\mathrm{method}}$ is small compared to its extremal values by using a non-constant number of input bits to address the table. This strategy will be interesting when synthesizing the tables as logic, as logic minimization will apply. We have already implemented this error analysis but we do not take advantage of it yet.

Another question which remains open is the interest of the Horner evaluation method when targeting hardware. In the literature concerning the precisions considered, we are only aware of very naive approaches [3, 9]. To get a fair comparison, the Horner approach should be studied with an effort on the error analysis similar to that described in this paper.

Acknowledgements

The authors would like to thank Arnaud Tisserand for many interesting discussions on this topic, and also for administrating the CAD tool server on which all the synthesis presented in this paper were performed.

References

[1] J. Cao, B. Wei, and J. Cheng. High-performance architectures for elementary function generation. In N. Burgess and L. Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic*, Vail, Colorado, June 2001.

[2] D. Das Sarma and D. Matula. Faithful bipartite ROM reciprocal tables. In S. Knowles and W. McAllister, editors, *12th IEEE Symposium on Computer Arithmetic*, pages 17–28, Bath, UK, 1995. IEEE Computer Society Press.

[3] D. Das Sarma and D. Matula. Faithful interpolation in reciprocal tables. In *13th IEEE Symposium on Computer Arithmetic*, pages 82–91, Asilomar, California, July 1997.

[4] F. de Dinechin and A. Tisserand. Some improvements on multipartite table methods. In N. Burgess and L. Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic*, pages 128–135, Vail, Colorado, June 2001. Updated version of LIP research report 2000-38.

[5] J. Detrey and F. de Dinechin. A VHDL library of LNS operators. In *37th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, USA, Oct. 2003.

[6] J. Detrey and F. de Dinechin. Second order function approximation using a single multiplication on FPGAs. In *14th Intl Conference on Field-Programmable Logic and Applications*, pages 221–230, Antwerp, Belgium, Aug. 2004. LNCS 3203.

[7] P. M. Farmwald. High bandwidth evaluation of elementary functions. In K. S. Trivedi and D. E. Atkins, editors, *Proceedings of the 5th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Los Alamitos, CA, 1981.

[8] V. Jain and L. Lin. High-speed double precision computation of nonlinear functions. In *12th IEEE Symposium on Computer Arithmetic*, pages 107–114, Bath, England, UK, July 1995.

[9] D.-U. Lee, W. Luk, J. Villasenor, and P. Cheung. Hierarchical segmentation schemes for function evaluation. In *IEEE Conference on Field-Programmable Technology*, Tokyo, Dec. 2003.

[10] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhauser, Boston, 1997.

[11] J.-M. Muller. A few results on table-based methods. *Reliable Computing*, 5(3):279–288, 1999.

[12] J. Piñeiro and J. Bruguera. High-speed double-precision computation of reciprocal, division, square root, and inverse square root. *IEEE Transactions on Computers*, 51(12):1377–1388, Dec. 2002.

[13] J. A. Piñeiro, J. D. Bruguera, and J.-M. Muller. Faithful powering computation using table look-up and a fused accumulation tree. In N. Burgess and L. Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic*, pages 40–47, Vail, Colorado, June 2001.

[14] J. Stine and M. Schulte. The symmetric table addition method for accurate function approximation. *Journal of VLSI Signal Processing*, 21(2):167–177, 1999.

[15] N. Takagi. Powering by a table look-up and a multiplication with operand modification. *IEEE Transactions on Computers*, 47(11):1216–1222, Nov. 1998.