New Evaluation Scheme for Software Function Approximation with Non-Uniform Segmentation

Justine Bonnot, Erwan Nogues, Daniel Menard
INSA Rennes, UEB, IETR, UMR 6164
Rennes, France
Email: firstname.lastname@insa-rennes.fr

***Abstract*—Modern applications embed complex mathematical processing based on composition of elementary functions. A good balance between approximation accuracy, and implementation cost, i.e. memory space requirement and computation time, is needed to design an efficient implementation. From this point of view, approaches working with polynomial approximation obtain results of a monitored accuracy with a moderate implementation cost. For software implementation in fixed-point processors, accurate results can be obtained if the segment on which the function is computed I is segmented accurately enough, to have an approximating polynomial on each segment. Non-uniform segmentation is required to limit the number of segments and then the implementation cost. The proposed recursive scheme exploits the trade-off between memory requirement and evaluation time. The method is illustrated with the function $\exp(-\sqrt(x))$ on the segment $[2^{-6}; 2^{5}]$ and showed a mean speed-up ratio of 98.7 compared to the mathematical C standard library on the Digital Signal Processor C55x.**

I. Introduction

The technological progress in microelectronics as well as the Internet of Things era require that embedded applications integrate more and more sophisticated processing. Most embedded applications incorporate complex mathematical processing based on composition of elementary functions. The design challenge is to implement these functions with enough accuracy without sacrificing the performances of the application which are measured in terms of memory usage, execution time and energy consumption. The targeted processors are Digital Signal Processors (DSP). To implement these functions, several solutions can be used. Specific algorithms can be adapted to a particular function [6] as for instance the approximation by Tchebychev polynomials or by convergent series. For a hardware implementation, numerous methods have been proposed. Look-Up Tables (LUT), bi- or multipartite tables [2] are used. These tables contain the values of the function on the targeted segment and are impossible to include in embedded systems because it may take a lot of memory space for a given precision.

For the time being, a few software solutions exist for computing these functions. The C standard library can be used but targets scientific computation. It offers an important accuracy but is slow on the targeted architecture (DSP). The implementation of the CORDIC (COordinate Rotation DIgital Computer) algorithm that computes the values of trigonometric, hyperbolic or logarithmic functions [7], can also be used.

In this work, the software implementation of mathematical functions in embedded systems is considered. Low cost and low power processors are targeted. To achieve cost and power constraints, no floating-point unit is considered available and processing is carried-out with fixed-point arithmetic. Nevertheless, this arithmetic offers a limited dynamic range and precision. Then, the *polynomial approximation* of a function give a very accurate result in a few cycles if the segment I on which the function is computed, is segmented precisely enough to approximate the function by a polynomial on each segment.

To approximate the function, the Remez algorithm is used. The degree of the approximating polynomials can be chosen and impacts the number of segments needed to suit the accuracy constraint as well as the computation time and the memory required. Reducing the polynomial order increases the approximation error. Thus, to obtain a given maximal approximation error, the number of segments increases implying an increasing number of polynomials and then a larger memory footprint. On the contrary, for a given data-path word-length, increasing the polynomial order raises the computation errors in fixed-point coding due to more mathematical and scaling operations. Even though higher polynomial order implies a smaller approximation error and consequently less segments, a higher fixed-point computation error counteracts this benefit. Thus, for fixed-point arithmetic, the polynomial order is relatively low. Consequently, to obtain a low maximal approximation error, the segment size is reduced. Accordingly, non-uniform segmentation is required to limit the number of polynomials to store in memory and to obtain a moderate approximation error. Then, each segment has its own approximating polynomial and the coefficients of each polynomial are stored in a single table $\mathcal{P}$.

Consequently, the challenge of the polynomial approximation method is to find the accurate segmentation of the segment I as well as the fastest computation of the index of the polynomial in table $\mathcal{P}$ corresponding to the input value x. In the proposed method, the computation of the index of the polynomial associated to an input value x is considered. Different non-uniform segmentations [4] associated to a hardware method of indexation of the segmentation have been proposed, but they target only hardware implementations. Moreover, these methods do not provide flexibility in terms of segmentation. The segmentation used in the proposed method is a non-uniform segmentation.

In this paper, a new indexing scheme for software function evaluation, based on polynomial evaluation with non uniform segmentation, is proposed. This recursive scheme enables exploring the trade-off between the memory size and the function evaluation time. Besides, compared to Table-based methods,

978-0-9928-6265-7/16/$31.00 ©2016 IEEE

our approach reduces significantly the memory footprint. The proposed method is compared to an indexing method using only conditional structures and shows a significant reduction in the computation time. The determination of the best non-uniform segmentation is not the scope of this paper.

The rest of the paper is organized as follows. First, the related work is detailed in Section II. The evaluation scheme using a non-uniform segmentation is detailed in Section III. Finally, the experiment results as well as comparisons with indexing method using conditional structures and table-based method are given in Section IV.

II. Related work

To compute the value of a function, iterative methods as the CORDIC algorithm [7] can be a software solution. That algorithm computes approximations of trigonometric, hyperbolic or logarithmic functions with a fixed precision. Nevertheless, that algorithm is composed of a loop whose number of iterations depends on the precision required and may take too long to compute precise values. A table needs to be stored too, whose size is the same as the number of iterations. The asset of that method is the sole use of shifts and additions that makes it particularly adapted to low cost hardware solutions.

Numerous hardware methods have been developed for polynomial approximation. Firstly, the LUT method consists in approximating the function with 0-degree polynomials on the segment I beforehand segmented so that the error criterion is fulfilled on each segment. That method is the most efficient in terms of computation time but the most greedy concerning the memory space required since the segmentation is uniform. Improvements of that method are the bi- or multi-partite methods presented in [2]. The function f is approximated by linear functions on the segment I prior segmented. Two tables need to be stored: a table containing the initial values of each segment obtained by the segmentation, and a table of offsets to compute whichever value belonging to this segment. The multi-partite method exploits the symmetry on each segment and reduces significantly the size of the tables. That method offers quick computations and reduced tables to store but is limited to low-precision requirements and is implemented only for hardware function evaluation.

Non-uniform segmentation followed by polynomial approximation is developed in [4] and [5] for hardware function evaluation. The initial segment is recursively segmented until the error criterion is fulfilled on each segment. The segmentation is done according to 4 predefined segmentation schemes limiting the ability to fit onto the function evolution. Afterwards, AND and OR gates are used to find the segment corresponding to an input value x. LUT are used to store the coefficients of the polynomials. Nevertheless, that method does not allow to control the depth of segmentation of the initial segment. Finally that method targets only hardware implementation.

III. Proposed method

A. Non-uniform Segmentation

The function f is approximated by the Remez algorithm as in the article [4]. That algorithm seeks the minimax, which is the polynomial that approximates the best the function according to the infinite norm on a given segment. The Remez algorithm is based on several inputs: the function f, the segment $I = [a; b]$ on which f is approximated (f has to be continuous on I), and the degree N_d of the approximating polynomials. The Remez algorithm in the proposed method, is called with the Sollya tool [1].

The Remez approximation algorithm has an approximation error ϵ_{app}. That error cannot be controled by the Remez algorithm but can be computed on the segment I. ϵ_{app} is defined as the infinite norm of the difference between the function f and its approximating polynomial P, $\epsilon_{app} = ||f - P||_\infty$. On the segment I, the error of approximation can be greater than the maximal error required by the user. The segmentation of I allows to suit the error criterion provided by the user, on each segment. The coefficients of the approximating polynomials on each segment are saved in table $\mathcal{P}$. Once the segmentation is determined, the challenge is to index efficiently $\mathcal{P}$. To ease the addressing, the bounds of the segments obtained are sums of powers of two.

The segmentation of I is modeled by a tree, as depicted in figure 1. That tree is composed of nodes and edges. The root of that tree contains the bounds of the segment I and the children of the root contain the bounds of the segments obtained in each level of segmentation. The leaves of the tree correspond to the segments to which a polynomial is associated, i.e. the segments on which the error criterion is verified. The depth of the tree is a trade-off between the number of polynomials and the computation time of the index of a polynomial. For instance, the binary tree obtained by subdividing each segment in two equal parts is the deepest: it leads to the minimum number of polynomials but the computation of the index takes the longest. On the contrary, the tree whose depth is 1 has the greatest number of polynomials but the computation of the index is the fastest. If the depth of the binary tree is N, then the number of polynomials in the tree of depth 1 is 2^N.

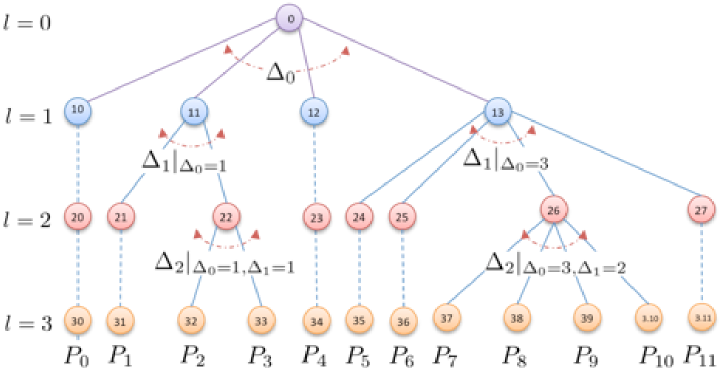

Fig. 1. Example of a tree obtained with a non-uniform segmentation

On the tree in figure 1, the segment I is segmented in 2^2 segments, that leads to 2 segments on which the error criterion is fulfilled (leaves n_{10} and n_{12}). The segment represented by the node n_{11} has an approximating polynomial that does not approximate accurately enough f, then that segment is segmented in 2^1 segments represented by leave n_{21} and node n_{22} that is segmented again in 2^1 segments. The segment represented by the node n_{13} is segmented in 2^2 segments that

$i_0 = 0$

i_0	0
$\mathcal{T}[0][i_0].o$	0
$\mathcal{T}[0][i_0].m$	x[15..14]

$i_1 = \Delta_0$

i_1	0	1	2	3
$\mathcal{T}[1][i_1].o$	0	0	1	1
$\mathcal{T}[1][i_1].m$	0	x[13]	0	x[13..12]

$i_2 = i_1 + o_1 + \Delta_1$

i_2	0	1	2	3	4	5	6	7
$\mathcal{T}[2][i_2].o$	0	0	0	1	1	1	1	4
$\mathcal{T}[2][i_2].m$	0	0	x[12]	0	0	0	x[11..10]	0

$i_3 = i_2 + o_2 + \Delta_2$

Fig. 2. Indexing tables associated to the tree in Figure 1

leads to 3 leaves (leaves n_{24}, n_{25} and n_{27}) and a last node to segment, n_{26}, in 2^2 segments.

B. Evaluation scheme

The main contribution of this paper is a new method to index the polynomial coefficients table $\mathcal{P}$ in a minimum time. Each line of this table represents a polynomial and each column the coefficient of the degree i monomial. The index corresponding to the input value x is the number of the line of that table used to apply the function to x, i.e. the segment in which the value x is. Once the non-uniform segmentation is obtained, polynomial coefficients are saved in $\mathcal{P}$. The evaluation scheme, to approximate the function f, is composed of two parts corresponding to index computation and polynomial computation as presented in figure 3. The step of index computation determines from the w_m most significant bits the index i used to address the polynomial table $\mathcal{P}$. The step of polynomial computation evaluates the polynomial P_i with x.

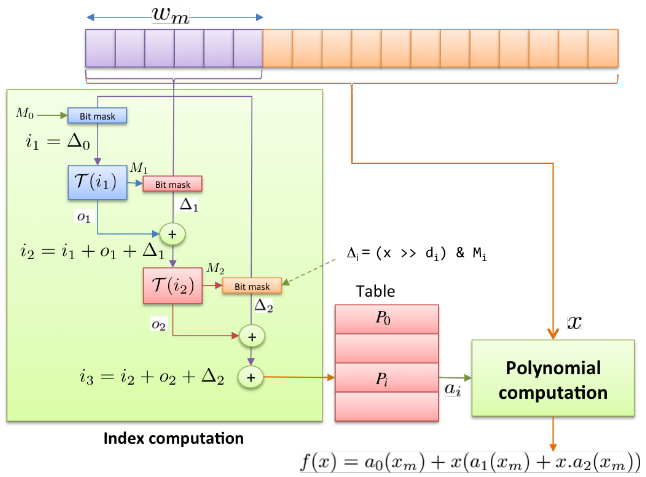

Fig. 3. Evaluation scheme integrating index and polynomial computation. Illustration for a three-level tree

1) Index Computation: The aim of this step is to determine for an input value x the index associated to the segment in which x is located to get the coefficients of the polynomial approximating f on that segment. The problem is to find the path associated to the segment in a minimal time. Timing constraint discards solutions based on conditional structures and requiring comparison, test and jump instructions, as shown in Section IV. The proposed approach is based on the analysis and interpretation of specific bits of x, formatted using fixed-point coding. Thanks to the sum of powers of two segment bounds, masking and shifting operations can be used to align these bits on the LSB and select them.

Since the tree is not well balanced due to the non-uniform segmentation, for any tree level, the number of bits to analyze is not constant and depends on the considered node. Indeed, all the nodes associated to a given level do not necessarily have the same number of children. The indexing method uses a table $\mathcal{T}$ which stores for each level, a structure for each node, containing the mask, the shift and the offset to pass from a level to the following, given the bits of x. A line of the table $\mathcal{T}$ is associated to each tree level. Each line of $\mathcal{T}$ contains N_{nl} elements, where N_{nl} is the number of total nodes in this level l. At each intermediate node n_{lj} (where l is the level and j the node) a mask ($\mathcal{T}[l][j].M$) and a shift ($\mathcal{T}[l][j].s$) are associated and used to select the adequate bits of x to move from node n_{lj} to the next node $n_{l+1j'}$ located at level $l+1$. Moreover, an offset ($\mathcal{T}[l][j].o$) is used to compute the index of the polynomial associated to the considered segment. The number of information to store in the table $\mathcal{T}$ depends on the depth of the tree, the number of polynomials, and the degree of the approximating polynomials.

The pseudo code for the computation of the index of the polynomial corresponding to an input value x, from the tables detailed previously, is presented in Algorithm 1. For each node n_{lj}, an offset o_{lj} is provided and the index is obtained by summing the offsets to the different Δ_l, corresponding to the result of the bit-masking at each level l:

$$i = \sum_{l=0}^{N_l - 1} o_l + \Delta_l \qquad (1)$$

The pseudo code to compute the index is compact and composed of few operations: three memory readings, a shift, a bitwise logic operation and two additions.

Algorithm 1 Indexing of the approximating polynomials

```
$i = 0$;
for $l = 0$ to $N_l - 1$ do
    $o := \mathcal{T}[l][i].o$
    $\Delta := (x >> \mathcal{T}[l][i].s) \& \mathcal{T}[l][i].M$
    $i := i + \Delta + o$
end for
Return $i$
```

The value of the mask $\mathcal{T}[l][j].M$ associated to a node n_{lj} is obtained from the number of children $NbCh(n_{lj})$ of that node with the following expression:

$$\mathcal{T}[l][j].M = NbCh(n_{lj}) - 1 \qquad (2)$$

The value of the shift $\mathcal{T}[l][j].s$ corresponding to a node n_{lj} is obtained from the number of children $\mathrm{NbCh}(n_{lj})$ and from the shift associated to node $n_{l-1,j''}$ corresponding to the parent of node n_{lj} with the following expression

$$\mathcal{T}[l][j].s = \mathcal{T}[l-1][j''].s - \log_2(\mathrm{NbCh}(n_{lj})) \qquad (3)$$

The value of the offset $\mathcal{T}[l][j].o$ associated to a node n_{lj} is obtained from the number of children $\mathrm{NbCh}(n_{ij})$ and from the shift associated to node $n_{l-1,j''}$ corresponding to the parent of node n_{lj} with the following expression

$$\mathcal{T}[l][j].o = \sum_{k=0}^{j-1} \mathrm{NbCh}(n_{lk}) - 1 \tag{4}$$

For the considered example in figure 1, the tree is made-up of 3 levels leading levels in table $\mathcal{T}$. As an example, let us consider, the 16 bits value x with $x[15..10] = 111001b$. In the tree from figure 1, in level 1, the 2 most significant bits $x[15..14]$ are tested. The index $i_1 = \Delta_0 = 3$ and leads to node n_{13}. Then, due to the mask M_1 for node n_{13}, the bits $x[13..12]$ are tested. The index $i_2 = i_1 + o_1 + \Delta_1 = 3 + 1 + 2 = 6$ and leads to n_{26}. Finally, due to the mask M_2 for node n_{26}, the bits $x[11..10]$ are tested. The index $i_3 = i_2 + o_2 + \Delta_2 = 6 + 1 + 1 = 8$ and leads to node n_{38}. The index of the polynomial associated to x is 8. The polynomial associated is P_8.

2) Polynomial Computation: To improve the speed performances, fixed-point arithmetic is used to code the coefficients of the polynomials approximating the function, and they are stored in the bidimensional table $\mathcal{P}[i][j]$. The term i is the index of the segment and the coefficient is the one of the j-degree monomial. Indeed, the computation of the value of $P_i(x)$, where P_i is the approximating polynomial on a segment indexed by i, can be decomposed using the Horner rule, reducing the computation errors. According to the Ruffini-Horner algorithm [8], each polynomial $P_i(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$ can be factored as:

$$P_i(x) = ((((a_n x + a_{n-1})x + a_{n-2})x + ...)x + a_1)x + a_0$$

Using that rule, the calculation scheme can be decomposed in a basic loop kernel presented in Algorithm 2.

Algorithm 2 Computation of the polynomial P_i in fixed-point.

```
IntDP $z$;
$z := \mathcal{P}[i][N_d - 1]$
for $d = N_d$ to 1 do
    $z := ((IntSP)z \times x) >> \mathcal{D}[i][d] + \mathcal{P}[i][d]$
end for
```

Taking into account the loop kernel, shifts are necessary to compute the value of $P_i(x)$ in fixed-point coding because the output of the multiplier can be on a greater format than necessary. By using arithmetic of intervals on each segment, the real number of bits necessary for the integer part can be adjusted with left shift operation.

A right shift operation can be necessary to put the two adder operands on the same format. In our case, a quantification from double (IntDP) to single precision (IntSP) is done and creates a source of error ϵ_{fxp}. When the two shift values have been computed, they can be both added to do a single shift on the output of the multiplier and is stored in the bidimensional table $\mathcal{D}[i][d]$ where d corresponds to the iteration of the loop of Algorithm 2 and i is the index of the segment.

Finally, since parameter N_d is known and constant for all P_i on an approximation, the loop can be unrolled so as to avoid overhead due to loop management.

IV. Experiments

To illustrate the proposed method, the function $\exp(-\sqrt(x))$ is studied on segment $[2^{-6}; 2^5]$ and the DSP C55x from Texas Instruments [3] is considered. The considered maximum error is $\epsilon = 10^{-2}$. The trees for 1- to 3-degree polynomials have been computed with depth varying from the maximal depth (binary tree) to a depth of 2. The evolution of the memory footprints S_{n-poly} and S_{n-tot} is observed in figure 4 where S_{n-poly} is the memory footprint for the polynomial coefficients (table $\mathcal{P}$) and shifts (table $\mathcal{D}$ for fixed-point computations) and $Sn - tot$ is $Sn - poly$ added to the memory footprint for the indexing table $\mathcal{T}$ in figure 4. The trees are computed with a maximal error criterion for the approximation ϵ_{app} equal to half of ϵ ($5 \cdot 10^{-3}$). The data and coefficients are on single-precision (16 bits), the coding of the input is $Q_{6,10}$.

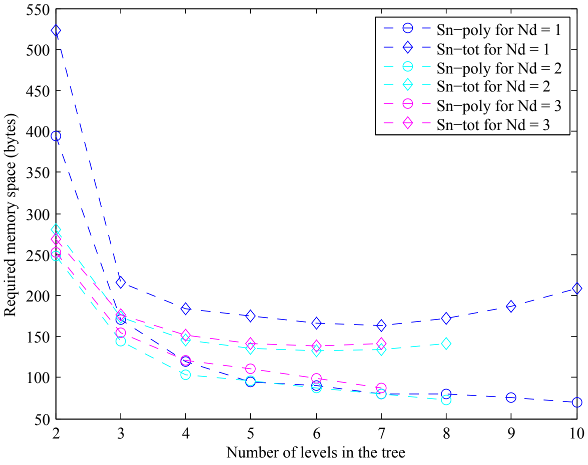

Fig. 4. Evolution of the memory footprints S_{n-poly} (tables $\mathcal{P}$ and $\mathcal{D}$) and S_{n-tot} (tables $\mathcal{P}$, $\mathcal{D}$ and $\mathcal{T}$).

The less levels the tree has, the greatest number of polynomials there is. Consequently, S_{n-poly} is high. Nevetheless, since the tree has a reduced depth, few tables are needed. On the contrary, the more levels the tree has, the less polynomials there are and S_{n-poly} is low. However, the table $\mathcal{T}$ is the biggest.

The total memory space can be characterized as a function of the computation time for a given error. This performance can be used as a Pareto curve during the system design phase to select the configuration leading to a good trade-off. The function $\exp(-\sqrt(x))$ on the segment $[2^{-6}; 2^5]$ can be approximated by a polynomial of degree from 1 to 4 with data coded on 16 bits, requiring a maximal total error ϵ of 10^{-2}. The maximum values of the error of fixed-point coding are presented in table I. A 5-degree polynomial does not suit this approximation since the error obtained with fixed-point coding is greater than ϵ_{fxp}. However, a 5-degree polynomial would fulfill the error criterion with data coded in double-precision but at the expense of a high increase in execution time.

The expression of the computation time t depending on the degree N_d and the number of levels in the tree N_l in single-precision with the provided code is:

$$t = 8 \cdot N_l + 3 \cdot N_d + 9 \tag{5}$$

Degree	ϵ_{fxp}
1	$[-2.8 \cdot 10^{-3}; 0]$
2	$[-2.5 \cdot 10^{-3}; 0]$
3	$[-2.5 \cdot 10^{-3}; 0.3 \cdot 10^{-3}]$
4	$[-2.4 \cdot 10^{-3}; 1.5 \cdot 10^{-3}]$
5	$[-2.7 \cdot 10^{-3}; 35.2 \cdot 10^{-3}]$

TABLE I. RANGE OF THE FIXED-POINT CODING ERROR DEPENDING ON THE POLYNOMIAL DEGREE FOR THE APPROXIMATION OF $exp(-\sqrt(x))$

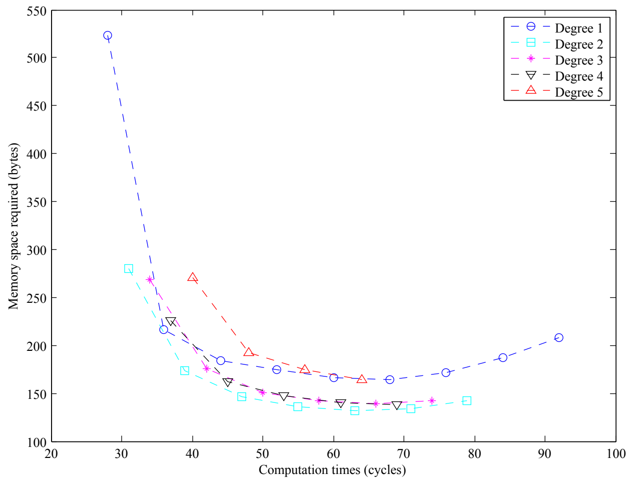

Fig. 5. Pareto curves for approximating $exp(-\sqrt(x))$ on $[2^{-6}; 2^5]$)

Figure 5 provides the evolution of the total memory footprint S_{n-tot} and the computation time t. The tree with a low number of levels implies a high memory footprint but a minimum computation time. Then, the computation time increases with the number of levels in the tree while the required memory decreases until reaching a minimum. Finally, the memory slightly increases with the computation time and the number of levels due to the increase of the table $\mathcal{T}$ size.

The proposed method is compared to the C standard library, to the LUT method and to an indexing method using conditional structures. However, the proposed method cannot be compared to the hardware implementation of bi- or multipartite tables since the method proposed is a software-based method. The computation time obtained with our method shows a mean speed-up of 98.7 compared to the implementation by Texas Instruments of the C standard library on the DSP C55x [3]. Our approach is compared to the method using only conditional structures to index $\mathcal{P}$. To find the segment in which an input value x is, each segment is tested using *if* statements while the right one has not been found. The computation time of the index with that method is not constant and depends on the segment containing x. To take into account this variability, mean execution time is considered. The results are presented for 1 to 3-degree approximating polynomials. Given that our approach provides different trade-off, the minimal, the mean and the maximal speed-ups are considered. The overhead in memory size and execution time of the conditional indexing method compared to our approach are presented in table II. The segmentation and the conditions of approximation are the same as in figure 5. Our approach requires more memory (overhead lower than 1) due to table $\mathcal{T}$ storage, but reduces significantly the execution time (overhead significantly greater than 1).

Finally, the memory space required by the proposed method (Mem_{prop}) is compared in table III to the LUT method (Mem_{tab}) for several approximations. The memory required is given in bytes. Our approach reduces significantly the memory footprint compared to LUT method.

	Memory			Execution time		
Degree	Min	Mean	Max	Min	Mean	Max
1	0.21	0.45	0.66	3.73	23.28	104.82
2	0.11	0.30	0.49	3.93	8.52	17.41
3	0.06	0.21	0.38	3.78	10.24	25.53

TABLE II. OVERHEAD IN MEMORY SIZE AND EXECUTION TIME OF THE CONDITIONAL INDEXING METHOD COMPARED TO OUR APPROACH

Function	ϵ_{tot}	$[a; b]$	Mem_{tab}	Mem_{prop} (mean)
$\exp(-\sqrt(x))$	10^{-2}	$[2^{-6}; 2^5]$	8192	206
$\sqrt(-\log(x))$	0.02	$[2^{-5}; 1]$	256	169
$\sin(x)$	10^{-2}	$[0; \frac{\pi}{2}]$	256	32

TABLE III. MEMORY REQUIREMENTS (BYTES) FOR THE PROPOSED METHOD Mem_{prop} AND THE LUT METHOD Mem_{tab}

V. CONCLUSION

The method proposed in this paper enables system designers to efficiently evaluate the cost of approximating a function. Indeed, Pareto curves giving the memory footprint depending on the computation time allow to choose a trade-off between computation time and required memory space. That trade-off is obtained thanks to the different degrees of the approximating polynomials as well as the depth of the tree storing the segmentation of the segment I on which the function is computed. Besides, the new scheme of indexing the table of polynomials shows a significant reduction in terms of computation time and does not need a significant supplementary memory space compared to an indexing method using only conditional structures. Compared to the C standard library implementation, the proposed method shows significant computation time reduction for low-degree polynomials since the speed-up mean of the proposed method on DSP C55x is 98.7.

REFERENCES

[1] S. Chevillard, M. Joldeş, and C. Lauter. Sollya: An environment for the development of numerical codes. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 28–31, Heidelberg, Germany, September 2010. Springer.

[2] F. de Dinechin and A. Tisserand. Multipartite table methods. *IEEE Transactions on Computers*, 54(3):319–330, March 2005.

[3] Texas Instruments. *C55x v3.x CPU Reference Guide*. Dallas, June 2009.

[4] D.-U. Lee, R.C.C. Cheung, W. Luk, and J.D. Villasenor. Hierarchical segmentation for hardware function evaluation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(1):103–116, January 2009.

[5] Dong-U Lee, Wayne Luk, John Villasenor, and Peter YK Cheung. Non-uniform segmentation for hardware function evaluation. In *Field Programmable Logic and Application*, pages 796–807. Springer, 2003.

[6] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C (2nd ed.): the art of scientific computing*. Cambridge University Press New York, NY, USA, 1992.

[7] Jack E. Volder. The CORDIC trigonometric computing technique. *IRE Transactions on Electronic Computers*, EC-8(3):330–334, September 1959.

[8] Edmund Taylor Whittaker and George Robinson. *The calculus of observations*. Blackie London, 1924.