HAL
open science

Using integer linear programming for correctly rounded multipartite architectures

Orégane Desrentes, Florent de Dinechin

To cite this version:

Orégane Desrentes, Florent de Dinechin. Using integer linear programming for correctly rounded multipartite architectures. FPT 2022 - International Conference on Field Programmable Technology, Dec 2022, Hong Kong, China. hal-03844218

HAL Id: hal-03844218

https://inria.hal.science/hal-03844218

Submitted on 8 Nov 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Using integer linear programming for correctly rounded multipartite architectures

Orégane Desrentes[†], Florent de Dinechin[‡]
[†]École Normale Supérieure de Lyon, France
oregane.desrentes@ens-lyon.fr
[‡]Univ Lyon, INSA Lyon, Inria, CITI, France
florent.de-dinechin@insa-lyon.fr

***Abstract*—This article introduces several improvements to the multipartite method, a generic technique for the hardware implementation of numerical functions. A multipartite architecture replaces a table of value with several tables and an adder tree. Here, the optimization of multipartite tables is formalized using Integer Linear Programming so that generic ILP solvers can be used. This improves the quality of faithfully rounded architectures compared to the state of the art. The proposed approach also enables correctly rounded multipartite architectures, providing errorless table compression. This improves the area by a factor 5 without any performance penalty compared with the state of the art in errorless compression. Another improvement of the proposed work is a cost function that attempts to predict the total cost of an architecture in FPGA architectural LUTs, where most of the previous works only count the size of the tables, thus ignoring the cost of the adder tree.**

I. Introduction: multipartite architectures

The multipartite method for numeric function approximation is a family of methods for the evaluation in hardware of a numerical function $f(x)$. It replaces the tabulation of the values of a function by an architecture that sums the output of several tables indexed by various subsets of the input bits (Figure 1).

Its root is the bipartite method (two tables), invented in the specific case of the sine function [1], and independently re-invented in the specific case of the reciprocal function [2]. A series of improvements generalised this technique to arbitrary functions and to more than two tables [3], [4], [5], [4], [5], [6], [7], [8]. All these methods are founded mathematically on a piecewise linear approximation of the function $f_i(x) \approx a_i + b_i x$, where the two terms a_i and $b_i x$ are themselves approximated by tabulation on each subdomain i. Arbitrary decompositions of the input word have also been explored [9], but the result is very similar to what is obtained by starting with a linear approximation.

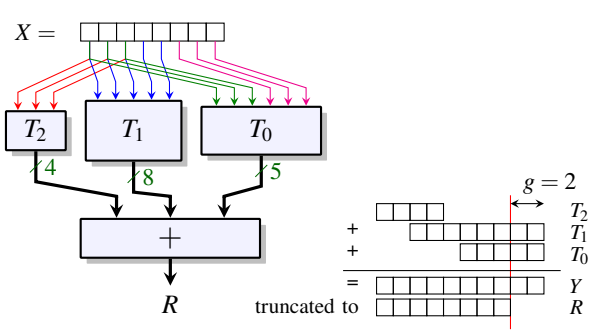


Fig. 1: Example multipartite architecture

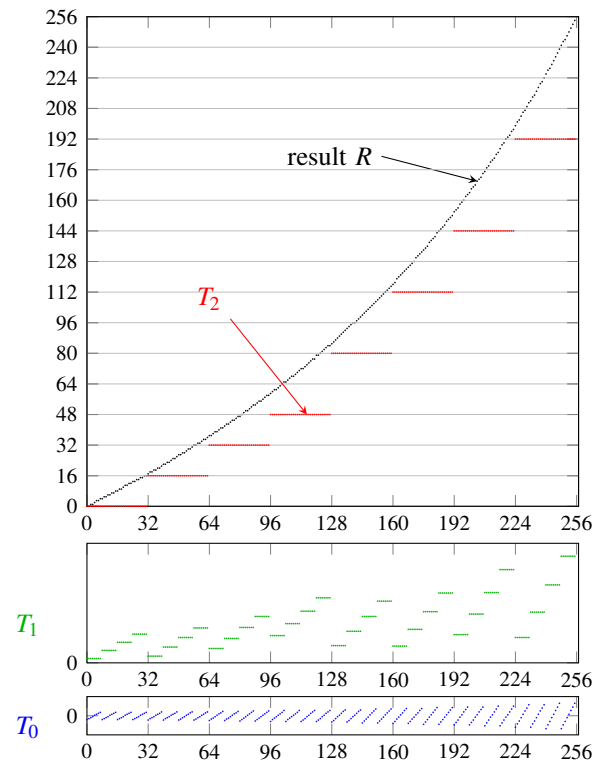


Fig. 2: Tables for an approximation to $(\frac{2}{2-x} - 1)$.

We refer the reader to the literature [6], [7] for the derivation of a multipartite architecture. For the purpose of this article, what matters is the resulting architecture, an example of which is shown in Figure 1, and the content of the tables, plotted in Figure 2. For each point of the horizontal axis of Figure 2, the sum of the content of the three tables that are accessed for this point provides a very good approximation to the function.

These figures illustrate various opportunities to reduce the table size that have been exploited in previous works:

- T_2 only holds 8 distinct values, addressed by the 3 leading bits of X (Figure 1). In Figure 2 for T_2, only one value is stored for each horizontal line. The same holds for T_1, which only holds 32 values (addressed by the 5 leading

bits of X.

- The value stored in T_2 is a multiple of 16/256, which means that its trailing zeroes need not be stored. In the general case, the same can hold for other tables, but it is not exploited in this small example.
- T_0 only holds 8 segements, each segment being repeated 4 times. As each segment contains 8 values, T_0 altogether holds 64 values. This corresponds to inputting to T_0 the 3 leading bits of X (indexing the set of repeated segment) and the 3 LSB bits of X (indexing the value in the segment).
- Since each table incurs an additional approximation, a classical technique is to perform to computation in extended precision ($g = 2$ bits on the example of Figures 1 and 2). The approximation and rounding errors accumulate in these guard bits, which are then simply dropped [6].

In this example, the total number of bits stored in tables is $2^3 \times 4$ (T_2) $+2^5 \times 8$ (T_1) $+2^6 \times 5$ (T_0) $= 608$ bits, much smaller than $2^8 \times 8 = 2048$ bits for a plain tabulation in a 8-bit in, 8-bit out table. This compression ratio improves with the input/output size.

As most of the tables (on our example T_1 and T_0) contain piecewise linear approximations, another table compression opportunity is to exploit the symmetry of each line segment with respect to its center. This opens the possibility to replace one table of Figure 1 with the slightly more complex architecture shown in Figure 3. By removing one input bit to the table, this halves its size, but at the cost of two rows of XORs that implement the negation.

All this defines a fairly large implementation space, but an exhaustive enumeration of this space is tractable [7] for the sizes for which such architectures make sense, ie, for input/output sizes between 8 bits and 24 bits (for larger precision, the tables get really large, and higher-order approximations must be used).

However, this exhaustive exploration of the parameter space is based on a worst-case error analysis that can only guarantee *faithful rounding:* the value returned by the architecture is not always the value nearest to the function value. Sometimes it is the next-closest value, as illustrated by Figure 4. Another equivalent point of view is that the error $\delta(x) = R - f(x)$ induced by the architecture is bounded by one unit in the last place (ulp) in a faithful architecture. Conversely, the error of a correctly rounded architecture is bounded by half an ulp.

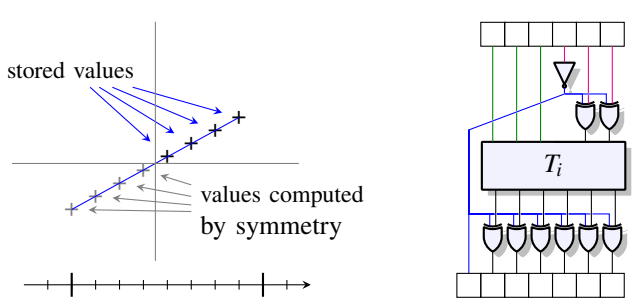


Fig. 3: Using symmetry to trade one table input bit for two rows of XOR gates

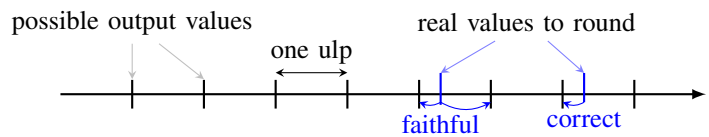


Fig. 4: Faithful versus correct rounding

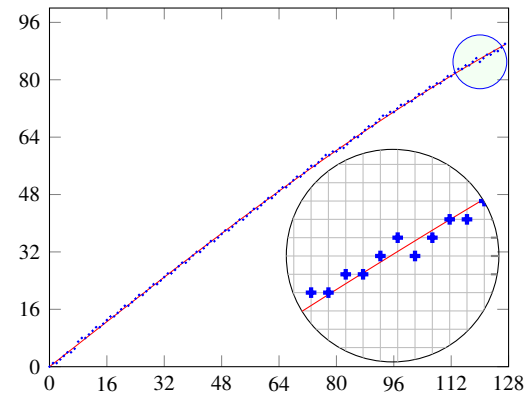


Fig. 5: Non-monotonicity in a faithful approximation of $\sin(\frac{\pi}{4}x)$

The present article introduces several improvements to the family of multipartite methods.

- The main contribution is an ILP model of multipartite architectures, presented in Section II. Inspired by comparable work on multiplier-based piecewise approximation [10], this model replaces analytical considerations on the function, as used in previous works, with a generic global optimization of the table contents. This results in several minor improvements, in particular smaller values of g can be used. In this model, the lossless table compression of the Table of Initial Values (TIV) of the most recent works [6], [7] becomes a special case of Tables of Offsets (TO), which simplifies the global optimization problem.
- A qualitative contribution is that this model also enables correctly rounded multipartite architectures. A correctly rounded architecture is much more expensive than a faithful one, and this is expected. However, another point of view is that multipartite architectures become a new approach for the lossless compression of a table of values in hardware. As such, they improve the state of the art in lossless compression [7], [11] by a very large margin. Also, an issue with faithful architectures is that they are not necessary monotonic [12], as illustrated by Figure 5. This issue disappears with correctly rounded architectures.
- The cost function used in most previous works is to count the number of table bits. This work introduces a more accurate cost function for FPGAs, taking into account the XORs and the addition tree. With it, the symmetry (Fig 3) will only be used when it saves more than it costs.

The benefits of these improvements are demonstrated in Section IV.

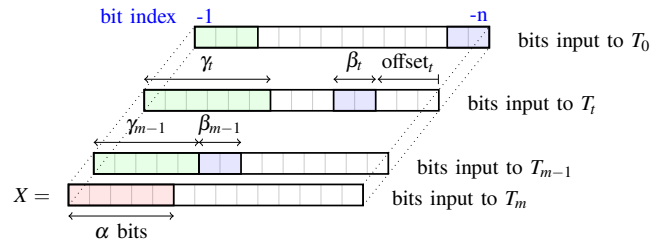


Fig. 6: Multipartite input word decomposition

II. An ILP model of multipartite architectures

A. Notations

The input word decomposition used in this work is shown in Figure 6.

The total number of tables in $m+1$. Each table T_t for $t \in \{0, m-1\}$ is associated with the three integer parameters γ_t, β_t, and offset_t which determine the bits of X input to T_t.

Tables T_m and T_{m-1} are special[1]: T_m has a single parameter α, and inputs the α MSB bits of the input X. T_{m-1} has the constraint that $\gamma_{m-1} = \alpha$. The other γ_t for $t \in \{0, m-1\}$ can be larger or smaller than α – the algorithm will determine them. They are such that $\gamma_0 \leq \gamma_1 \leq \ldots \leq \gamma_{m-2} \leq \gamma_{m-1} + \beta_{m-1}$. Remark that even T_m could be considered a special case of TO with $\gamma_m = \alpha$ and $\beta_m = 0$.

The input X is an n-bit number and the output R is a p-bit number, both in fixed point: without loss of generality, X and R belong to $[0,1)$. Their bits are numbered as their binary weights, *i.e.* the most significant bit of X has index -1 and the least significant bit has index $-n$.

Each table T_t for $t \in \{0, m-1\}$ can use the symmetry of Figure 3, or not. If a table T_t uses symmetry, the address it inputs is the integer:

$$\begin{aligned} A_t(X) &= 2^{\beta_t - 1} X[-1 : -\gamma_t] \\ &+ X[\text{offset}_t - n + \beta_t - 1 : \text{offset}_t - n] \oplus \neg X[\text{offset}_t - n + \beta_t] \end{aligned} \tag{1}$$

where $X[a:b]$ denotes the slice of bits between position a and position b, both included. If a table T_t does not use symmetry, the input address is:

$$\begin{aligned} A_t(X) &= 2^{\beta_t} X[-1 : -\gamma_t] \\ &+ X[\text{offset}_t - n + \beta_t : \text{offset}_t - n] \end{aligned} \tag{2}$$

The output of the table T_t for the input word X is $T_t(A_t(X))$. This will be written as $T_t(X)$ in the sequel to lighten notation.

The other notations used in this article are summarized in Table I.

[1]In the terminology of previous works, T_m and T_{m-1} constitute the compressed Table of Initial Values (TIV). T_m is the TI_{new} of [7], [8] and the T_{ss} of [11]. T_{m-1} is the TI_{diff} of these works. The other tables T_t for $t < m-1$ are called Tables of Offsets (TO) in previous works.

Constant	Meaning
$n \in \mathbb{N}$	Size of the input
$p \in \mathbb{N}$	Size of the output
$\text{is_correct} \in \mathbb{B}$	Ensure correct or faithful rounding
$\text{FPGA} \in \mathbb{B}$	Optimize for LUT count, or for table bit cost
$m \in \mathbb{N}$	$m+1$ is the number of tables
$g \in \mathbb{N}$	Maximum number of guard bits that can be used
$\alpha \in \mathbb{N}$	Size in bits of the input to T_m
For $t \in \{0, \ldots, m-1\}$	
$\beta_t \in \mathbb{N}$	Size of the rightmost chunk of X input to this table(see Figure 6)
$\gamma_t \in \mathbb{N}$	Size of the leftmost chunk of X input to this table (see Figure 6)
$\text{sym}_t \in \mathbb{B}$	Table T_t uses symmetry as of Figure 3
For $t \in \{0, \ldots, m\}$	
$\text{msb}_t \in \mathbb{B}$	A priori estimate of the most significant bit of the output of T_t
Variable	**Meaning**
For $t \in \{0, \ldots, m\}$	
$T_t \in \mathbb{B}_{2^{\text{in_size}_t}, msb_t+p+g}$	The values in the table T_t
$\text{unused_lsb}_t \in \mathbb{B}^{msb_t+p+g}$	Helps counting the trailings zeros of T_t
$\text{trailing_zeros}_t \in \mathbb{N}$	Number of trailing zeros of T_t
$\text{unused_msb}_t \in \mathbb{B}^{msb_T+p+g}$	Helps counting the leading zeros of T_t
$\text{leading_zeros}_t \in \mathbb{N}$	Number of leading zeros of T_t
size_t	Actual size of the output of the table

TABLE I: Notations used in the linear program.

B. Overview of the ILP-based optimization

Ideally we would like to express as an ILP problem the choice of the architecture parameters as well as the choice of values to fill the tables, in such a way that an ILP solver could find the smallest architecture evaluating the function with the required precision. However, an ILP solver can only optimize linear problems. To circumvent this limitation, when a parameter has a non-linear impact on the problem, it will be enumerated in a loop outside of the ILP, so that its value can be considered a constant inside the ILP.

As fixed-point numbers are scaled integers, the ILP formulation may use only the integers. For instance the value $Y \in [0,1)$ before the final rounding (Fig. 1, right) is represented by the integer $2^{p+g}Y$.

The core of the method is to have 2^n constraints (C1), one for each possible input value X in the fixed-point interval $[0,1)$.

$$\forall X \in \{0, \ldots, 2^n - 1\} \quad Y_L(X) \leq R(X) \leq Y_H(X) \tag{C1}$$

In each of these 2^n constraints, the input X is constant. The variable $R(X)$ represents the output of the architecture for the input X. The values $Y_L(X)$ and $Y_H(X)$ depend on the target rounding (Fig. 4). In the case of faithful rounding (C1.a), $Y_L(x)$ and $Y_H(X)$ correspond to the two values the function is allowed to take:

$$\begin{aligned} &\forall X \in \{0, \ldots, 2^n - 1\}, \\ &\quad Y_L(X) = \lfloor f(X) \times 2^p \rfloor, Y_H(X) = \lceil f(X) \times 2^{-p} \rceil \end{aligned} \tag{C1.a}$$

In the case of correct rounding (C1.b), $Y_L(x)$ and $Y_H(X)$ are both equal to the correctly rounded value of $f(X)$:

$$\forall X \in \{0,\dots,2^n-1\}, \quad Y_L(X) = Y_H(X) = \lfloor f(X) \times 2^p \rceil \quad \text{(C1.b)}$$

In both cases, for a constant X, the bounds $Y_L(x)$ and $Y_H(X)$ are also constants.

Unfortunately the value $R(X)$, defined as the truncation of $Y(X)$ (see Figure 1), is not ILP-friendly since truncation is not a linear operation. To address this, constraint (C1) can be replaced by a constraint on the variable $Y(X)$ which is the sum before truncation. The linear constraint (C1.c) ensures that the result $R(X)$ after truncation will fulfill (C1).

$$\forall X \in \{0,\dots,2^n-1\}, \quad Y_L(X) \times 2^g \leq Y(X) \leq Y_H(X) \times 2^g + 2^g - 1 \quad \text{(C1.c)}$$

Then constraint (C2) expresses that $Y(X)$ is the sum of the outputs of the tables $T_t(X)$ for $t \in \{0,\dots,m\}$.

$$\forall X \in \{0,\dots,2^n-1\}, Y(X) = \sum_{t\in\{0,\dots,m\}} T_t(X) \quad \text{(C2)}$$

In the case where symmetry is used, the output of the table need to be xored, which is not straightforward since modular arithmetic is not linear. The way to manage the symmetry is explained in section II-C.

The addressing of each table described in section II-A is highly non linear. However, since the input X of the architecture is a constant, it is not an issue to also treat $A_t(X)$ as a constant, provided that the values for all the parameters involved (m, α, β_t, γ_t, sym_t) are also constant. An external loop is used to enumerate the values of these parameters, leading to multiple calls to the ILP solver. This is similar to the exploration of the decomposition space in the classic multipartite method [6].

The output of each table $T_t(X)$ is represented as a vector of bits $T_t(X)[j]$, which are so many variables of the ILP. Constraint (C3) defines $T_t(X)$ as the weighted sum of those binary variables, and it is linear.

$$\forall X \in \{0,\dots,2^n-1\}, \forall t \in \{0,\dots,m\}, \quad T_t(X) = \sum_{j=-p-g}^{-1} T_t(X)[j] \cdot 2^{j+p+g} \text{ with } T_t(X)[j] \in \mathbb{B} \quad \text{(C3)}$$

Let us now address the formalization of the cost of a table. The number of bits contained in a table is $2^{\text{input_size}} \times$ output_size. The input size is a constant, determined by α, β_t, and γ_t. The output size, on the other hand, depends on the values inside the tables, and we want the ILP to minimize it. To this purpose, the output size may be expressed with a method introduced in [10]. For each table output $T_t(X)$, let us first define (C4.a) a binary variable unused_lsb for each bit of the output, equal to 1 if this bit and all bits of lower weights are equal to 0 (Boolean logic can be transformed into linear constraints, and most ILP solvers therefore support them):

$$\forall X \in \{0,\dots,2^n-1\}, \text{unused_lsb}_{t,X,k} = 1 \Leftrightarrow \forall j \in \{-p-g,\dots,k\}\ T_t(X)[j] = 0 \quad \text{(C4.a)}$$

These variables are synthesized for the full table by (C4.b):

$$\text{unused_lsb}_{t,k} = 1 \Leftrightarrow \forall X, \text{unused_lsb}_{t,X,k} = 1 \quad \text{(C4.b)}$$

This way, the number of unused least significant bits of a table T_t (which we want to maximize in order to minimize the cost of the table) is the sum of those variables (C4.c).

$$\text{trailing_zeros}_t = \sum_{k=-p-g}^{-1} \text{unused_lsb}_{t,k} \quad \text{(C4.c)}$$

Constraints (C5.a), (C5.b), and (C5.c) similarly count the number of leading zeroes.

$$\forall X \in \{0,\dots,2^n-1\}, \text{unused_msb}_{t,X,k} = 1 \Leftrightarrow \forall j \in \{k,\dots,-1\}\ T_t(X)[j] = 0 \quad \text{(C5.a)}$$

$$\text{unused_msb}_{t,k} = 1 \Leftrightarrow \forall X, \text{unused_msb}_{t,X,k} = 1 \quad \text{(C5.b)}$$

$$\text{leading_zeros}_t = \sum_{k=-p-g}^{-1} \text{unused_msb}_{t,k} \quad \text{(C5.c)}$$

This way, the output size of the table can be computed by removing the unused bits from the current size of the output (C6).

$$\text{size}_t = msb_t + p + g - \text{leading_zeros}_t - \text{trailing_zeros}_t + 1 \quad \text{(C6)}$$

Finally, the objective function of the ILP (Obj.Bits) minimizes the total number of stored bits.

$$\text{Minimize: } \text{size}_m \times 2^\alpha + \sum_{t\in\{0,\dots,m-1\}} \text{size}_t \times 2^{\gamma_t+\beta_t-\text{sym}_t} \quad \text{(Obj.Bits)}$$

C. Modelling symmetry

The XOR gates after the table (Fig. 3) implement an approximation of symmetry, using two's complement formula $-A = \neg A + 1$. This formula also shows how to emulate XORs by subtraction: $\neg A = -A - 1$. This corresponds to (C7.a):

$$\forall X \in \{0,\dots,2^n-1\}, \neg T_t(X) = -T_t(X) - \text{symm_one}_t \quad \text{(C7.a)}$$

where variable symm_one_t is a 1 whose weight should be 2^k where k is the smallest number such that $\text{unused_lsb}_t = 0$. It can be computed as (C7.b):

$$\text{symm_one}_t = 1 + \sum_{j=-p-g}^{-1} \text{unused_lsb}_{t,j} \times 2^{j+p+g} \ . \quad \text{(C7.b)}$$

D. Objective function for FPGA

The objective function (Obj.Bits) above minimizes the number of bits in the tables. In the case of a modern FPGA whose microarchitecture is based on 6-input Look-Up Tables (LUT6), an FPGA-oriented objective function should estimate the number of LUT6 instead of only counting bits. Replacing the size in (Obj.Bits) with $\text{size}_t \times 2^{\max(\gamma_t+\beta_t-6,0)}$ is a good approximation (the backend tools will attempt to perform further logic optimization). The max captures that reducing the input size of a table below 6 bits provides no gain.

Besides, the costs of the XORs should also be evaluated when symmetry is used. For recent Xilinx FPGAs, one LUT6 has two outputs and can be used for two XORs. The total cost of XORs (see Figure 3) is therefore well approximated by $0.5 \times (\beta_t + \text{size}_t - 1)$, again linear in size_t. When the tables are sufficiently small, the overhead of the XORs may not be worth the benefit.

The size of the adder tree can also be included in the objective function: we use the state-of-the-art bit heap compression framework of FloPoCo [13], [14] which costs in average 0.55 LUTs per compressed bit. As the number of compressed bits is the sum of the size_t, this contribution to the cost function is also linear in the size_t variables.

Summing up, an FPGA-oriented objective function is:

Minimize:

$$\text{size}_m \times 2^{\max(\alpha-6,0)} + \sum_{t=0}^{m-1} c_t + 0.55 \sum_{t=0}^{m} \text{size}_t$$

with

$$c_t = \begin{cases} 2^{\max(0,\gamma_t+\beta_t-7)}\text{size}_t + 0.5(\beta_t + \text{size}_t - 1) & \text{if } \text{sym}_t = 1 \\ 2^{\max(0,\gamma_t+\beta_t-6)}\text{size}_t & \text{otherwise} \end{cases} \quad \text{(Obj.LUTs)}$$

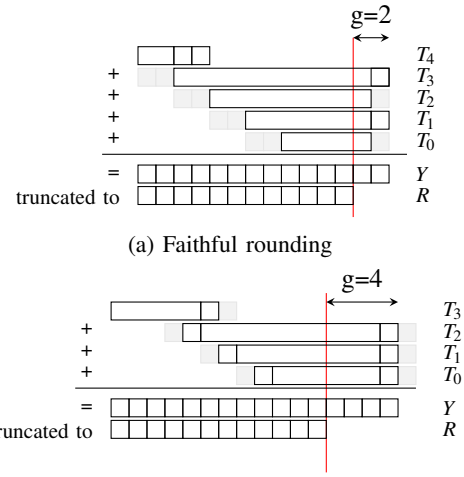
III. Solving the ILP program

So far we have an external loop that enumerates the non-linear parameter space, and for each parameter vector we solve an ILP program that defines the linear parameters and fills the table. This approach provides the optimal solution with respect to the cost model used, but it is quite slow: firstly, the parameter space is large, secondly the ILP problem is large, too, with more variables than there are bits in the tables. In practice this approach does not scale beyond $n = p = 12$.

To speed it up, we now first present two improvements in the model and in the exploration that make them more complex but do not jeopardize optimality, then several heuristics that improve performance but sacrifice the guarantee of optimality.

A. Mixing integers and bit array

To reduce the number of variables in the ILP problem, it is possible to replace several binary variables of the table output with the integer they represent. Fig. 7 show these integers as long blocks. We must however keep the most and least significant bits as binary variables, so that the ILP can optimize them out. In principle the MSB of each table output is known from previous works, as it is essentially a function of f and the decomposition $(\alpha, \beta_t, \gamma_t)$ [6]. However, it is interesting to give the freedom to the ILP to use one bit more or one bit less, as it will sometimes be able to exploit it. On the LSB side, the worst-case error analysis of previous works provides a value of g for which a solution is guaranteed to exist, but our objective function encourages the ILP to use fewer. The grey squares in Fig. 7 represent available bits that were removed by the linear program in one specific example.



(a) Faithful rounding

(b) Correct rounding

Fig. 7: Alignment for $\frac{2}{2^{-x}} - 1$ on 12 bits. The ILP had the possibility to use the grayed bits but chose not to.

B. Pruning the external loop

The previous analysis also provides a minimum possible size of the architecture (corresponding to the situation when all the removable bits are removed). This helps with the external loop: solutions whose minimum size is already bigger than the best solution found so far may be disregarded without even launching the ILP.

In addition, the external loop uses similar pruning as the literature, for instance a minimum γ_t for each table can be precomputed from the function and the other parameters.

C. Heuristics that jeopardize optimality

1) Using a estimate of the LUT size: The objective function (Obj.LUTs) is an improvement over (Obj.Bits) in terms of predicting the actual cost (after synthesis) but it only provides an estimate of the final LUT size: we are still at the mercy of the back-end tools.

2) Deciding symmetry a priori: Symmetry should always be used when optimising for (Obj.Bits), since it will always reduce the total size [3].

When optimising for (Obj.LUTs), symmetry may improve the solution, or not. Trying every possible combination of the sym_t is quite costly. Under the heuristic assumption that the ILP will find the same size_t in both alternatives of c_t in (Obj.LUTs), and also assuming that $\beta - 1 \leq \text{size}_t$, it is possible to decide that symmetry will be used for table T_t when $\gamma_t + \beta_t \geq 7$.

3) Maximum number of guard bits: For faithful rounding, the formula $g = 1 + \lfloor \log_2(m-1) \rfloor$ from the literature [6] provides a safe value of g, in the sense that a solution with this value is guaranteed to exist (and the ILP sometimes finds solutions with fewer guard bits). However, it could happen in principle that an overall better solution exists with a larger g, so this is a heuristic choice. For correct rounding, an empirical study for $n = p \in \{8, 10, 12, 14\}$ suggests the formula $g = 1 + (m-1) \times \lceil \log_2(n) - 2 \rceil$, so this formula is currently used in the code.

4) Setting an ILP timeout: When no solution exists, the ILP solver is usually very quick to decide so. It takes time when solutions exist and it is looking for the optimal one. A standard practice is to invoke the solver with a time-out period (we use 5 minutes) and keep the best solution found so far, which is not guaranteed to be optimal.

IV. Results

The ILP construction and the outer loop were implemented in the Julia language with JuMP, interfaced to the Gurobi solver. The obtained tables are then imported in FloPoCo to generate the architecture, in particular the compression tree. For the small bit width addressed here, the resulting VHDL could be tested exhaustively for correct or faithful rounding using the (reliable and time-tested) FloPoCo testbench framework. The source code can be found at https://github.com/OreganeD/flopoco-mpt-ilp.

Results are provided for the two functions $\frac{2}{2-x} - 1$ and $\sin(\frac{\pi}{4}x)$, respectively representative of the reciprocal and sine functions. The following methods are evaluated:

- ilpF is the proposed method targeting faithful rounding.
- mpt is the previous state of the art for faithful rounding: it is the current multipartite table implementation of FloPoCo, including several recent improvements such as Hsiao compression for the TIV [7].
- ilpC is the proposed method targeting correct rounding.
- table is the previous state of the art for correct rounding: it is a plain table of correctly rounded values, compressed using a generalization of Hsiao compression [11].

A first observation is that the ILP-based methods are much slower than the previous multipartite and table compressions methods. They take a few seconds for small sizes (8 and 10 bits), a few minutes for 12 bits, and a few hours for 16 bits, compared to less than a second for previous methods in all cases. ILP-based methods obviously scale poorly to input/output sizes larger than 16 bits. They are also slower for faithful rounding than for correct rounding.

The corresponding decompositions are given in Tables II and III. The decomposition columns describe the tables in the architecture in reverse order $(T_m, T_{m-1}, \ldots, T_0)$, and an underlined term means that symmetry was used. These tables also provide synthesis results obtained with Vivado v2022.1 for the Xilinx Kintex7 target. All the architectures here are purely combinatorial (without any pipelining).

An observation from these tables is that we are at the mercy of unpredictable optimizations by synthesis tools (Vivado here). It sometimes happens that the synthesis results (in LUTs) are worse for the architectures optimized for LUTs than for the architectures optimized for bit count. We have a telling example in the two ilpC:14 lines of Table III for $\sin(\frac{\pi}{4}x)$: the ILP has found the exact same decomposition, amounting to the exact same number of bits (15488) but the tables do not have exactly the same content, and this leads to a 3.5% difference in LUT cost after synthesis.

Figure 8 plots (for the two functions studied) the area improvement brought by the proposed method over the state of the art in the faithful case. Figure 9 similarly plots the area improvement of the proposed method versus the state of the art in the correct rounding case. The method is more beneficial for larger input/output sizes, with an area reduced by a factor 5 for 14-bit and 16-bit sine. In these figures, red is used for Obj.Bits, blue is used for Obj.LUTs, and dashed lines are used when we report a metric (LUTs or Bits) after optimising for the other.

Finally, Figure 10 plots the area overhead of correct rounding versus faithful rounding using the previous state of the art techniques (in green) and using the proposed method (optimising for Obj.Bits and counting bits, or optimising for Obj.LUTs and counting LUTs). Using linear programming has significantly reduced the relative overhead of correct rounding over faithful, but correct rounding remains an expensive luxury.

To sum up, ILP brings a consistent 20 to 30% gain in area for faithful rounding compared to the state of the art, and much more for correct rounding, with up to a factor 5 for the sine function compared to compressed tables [11]. Latency is also generally improved, as shown in Tables II and III.

It is difficult to compare this work to the multiplier-based solutions from [10] (they use piecewise polynomials of degree 1 or 2) since this publication does not show synthesis results for FPGAs. It is clear is that with much fewer variables and parameters, their approach scales better to large precisions, which is also the relevance domain of multiplier-based approximations.

V. Conclusion

This paper uses integer linear programming to improve the state of the art in multipartite architectures. This technique significantly reduces the cost of implementations and enables the construction of correctly rounded multipartite architectures. The main issue with this approach is that it does not scale well to large input/output sizes. A first direction for future work is therefore to improve the ILP model for better scaling. However, the proposed improvements to the objective function for targetting FPGAs could be backported to classical multipartite methods. This is another direction for future work on the subject.

Acknowledgements

This work was partly funded by the French Agence Nationale de la Recherche (ANR) through the Imprenum project.

		$\frac{2}{2-x}-1$					$\sin(\frac{\pi}{4}x)$				
specification		*algorithm results*		est.	*synthesis*		*algorithm results*		est.	*synthesis*	
name	Obj.	decomposition	bits	LUT	LUT	ns	decomposition	bits	LUT	LUT	ns
mpt:8	Bits	$5\cdot 2^3+8\cdot 2^5+\underline{6\cdot 2^6}$	680		25	6.8	$4\cdot 2^2+8\cdot 2^4+\underline{5\cdot 2^3}+\underline{3\cdot 2^2}$	196		28	6.7
ilpF:8	Bits	$1\cdot 2^2+10\cdot 2^4+\underline{7\cdot 2^5}+\underline{6\cdot 2^3}$	436		27	6.6	$2\cdot 2^2+9\cdot 2^4+\underline{6\cdot 2^2}+\underline{5\cdot 2^1}$	186		21	6.6
ilpF:8	LUTs	$3\cdot 2^3+7\cdot 2^6+4\cdot 2^4$	536	22	19	6.9	$3\cdot 2^3+6\cdot 2^6+3\cdot 2^2$	420	19	15	6.2
mpt:10	Bits	$6\cdot 2^4+10\cdot 2^6+\underline{8\cdot 2^6}+\underline{6\cdot 2^7}$	2016		59	7.7	$5\cdot 2^3+8\cdot 2^5+\underline{5\cdot 2^5}+\underline{3\cdot 2^4}$	504		32	7.1
ilpF:10	Bits	$4\cdot 2^4+10\cdot 2^6+\underline{7\cdot 2^5}+\underline{6\cdot 2^5}$	1120		47	7.2	$3\cdot 2^3+9\cdot 2^5+\underline{8\cdot 2^4}+\underline{6\cdot 2^3}$	488		27	6.8
ilpF:10	LUTs	$5\cdot 2^5+8\cdot 2^7+6\cdot 2^6$	1568	37	37	6.9	$5\cdot 2^5+6\cdot 2^7+4\cdot 2^5$	1056	29	29	6.7
mpt:12	Bits	$8\cdot 2^6+9\cdot 2^8+\underline{7\cdot 2^7}+\underline{5\cdot 2^7}$	4352		94	8.1	$6\cdot 2^4+11\cdot 2^6+\underline{8\cdot 2^5}+\underline{6\cdot 2^5}+\underline{4\cdot 2^3}$	1280		73	7.7
ilpF:12	Bits	$4\cdot 2^4+12\cdot 2^6+\underline{9\cdot 2^7}+\underline{8\cdot 2^6}+\underline{5\cdot 2^5}$	2656		77	8.2	$4\cdot 2^4+11\cdot 2^6+\underline{9\cdot 2^5}+\underline{7\cdot 2^3}+\underline{5\cdot 2^1}$	1122		56	7.6
ilpF:12	LUTs	$6\cdot 2^6+9\cdot 2^8+\underline{7\cdot 2^6}+5\cdot 2^6$	3456	73	75	7.7	$3\cdot 2^3+11\cdot 2^6+\underline{8\cdot 2^6}+5\cdot 2^5$	1400	47	47	7.4
mpt:14	Bits	$9\cdot 2^6+11\cdot 2^9+\underline{8\cdot 2^9}+\underline{5\cdot 2^8}$	11584		227	8.9	$7\cdot 2^5+11\cdot 2^7+\underline{8\cdot 2^7}+\underline{6\cdot 2^6}+\underline{3\cdot 2^4}$	3088		98	8.3
ilpF:14	Bits	$5\cdot 2^5+13\cdot 2^7+\underline{11\cdot 2^8}+\underline{9\cdot 2^7}+\underline{7\cdot 2^7}$	6688		171	8.7	$6\cdot 2^5+12\cdot 2^7+\underline{10\cdot 2^6}+\underline{8\cdot 2^5}+\underline{6\cdot 2^4}$	2720		84	8.3
ilpF:14	LUTs	$6\cdot 2^6+12\cdot 2^8+\underline{10\cdot 2^8}+\underline{8\cdot 2^6}+6\cdot 2^6$	6912	141	162	8.7	$4\cdot 2^4+12\cdot 2^7+\underline{10\cdot 2^6}+7\cdot 2^6+6\cdot 2^6$	3072	78	91	7.9
mpt:16	Bits	$10\cdot 2^7+13\cdot 2^{10}+\underline{10\cdot 2^9}+\underline{8\cdot 2^9}+\underline{6\cdot 2^9}$	26880		425	10.3	$7\cdot 2^5+14\cdot 2^8+\underline{10\cdot 2^7}+\underline{8\cdot 2^7}+\underline{6\cdot 2^6}+\underline{4\cdot 2^6}$	6752		190	9.3
ilpF:16	Bits	$6\cdot 2^6+15\cdot 2^8+\underline{13\cdot 2^9}+\underline{10\cdot 2^8}+\underline{8\cdot 2^7}+\underline{7\cdot 2^6}$	14912		328	10.3	$6\cdot 2^6+14\cdot 2^8+\underline{11\cdot 2^7}+\underline{10\cdot 2^6}+\underline{8\cdot 2^4}+\underline{6\cdot 2^1}$	6156		175	9.3
ilpF:16	LUTs	$6\cdot 2^6+15\cdot 2^8+\underline{12\cdot 2^9}+\underline{11\cdot 2^8}+\underline{8\cdot 2^7}+\underline{7\cdot 2^7}$	15104	289	273	10.1	$6\cdot 2^6+13\cdot 2^8+\underline{11\cdot 2^7}+\underline{9\cdot 2^6}+7\cdot 2^6+5\cdot 2^4$	6224	140	173	9.3

TABLE II: All synthesis results for faithful rounding

		$\frac{2}{2-x}-1$					$\sin(\frac{\pi}{4}x)$				
specification		*algorithm results*		est.	*synthesis*		*algorithm results*		est.	*synthesis*	
name	Obj.	decomposition	bits	LUT	LUT	ns	decomposition	bits	LUT	LUT	ns
table:8	Bits	$7\cdot 2^5+4\cdot 2^8$	1248		28	6.8	$7\cdot 2^5+4\cdot 2^8$	1248		28	6.8
ilpC:8	Bits	$4\cdot 2^4+8\cdot 2^6+\underline{6\cdot 2^6}$	960		25	6.6	$5\cdot 2^4+7\cdot 2^6+\underline{5\cdot 2^3}$	568		22	6.8
ilpC:8	LUTs	$3\cdot 2^3+9\cdot 2^6+\underline{6\cdot 2^6}$	984	31	25	6.7	$3\cdot 2^3+8\cdot 2^6+5\cdot 2^4$	616	25	19	6.3
table:10	Bits	$10\cdot 2^7+4\cdot 2^{10}$	5376		85	7.2	$10\cdot 2^7+4\cdot 2^{10}$	5376		85	7.2
ilpC:10	Bits	$5\cdot 2^5+10\cdot 2^7+\underline{8\cdot 2^7}$	2464		53	7.2	$4\cdot 2^4+11\cdot 2^6+\underline{9\cdot 2^6}+\underline{7\cdot 2^3}$	1400		43	7.0
ilpC:10	LUTs	$5\cdot 2^5+10\cdot 2^7+\underline{8\cdot 2^7}$	2464	59	53	7.2	$7\cdot 2^6+7\cdot 2^8+5\cdot 2^4$	2320	50	60	6.9
table:12	Bits	$12\cdot 2^9+4\cdot 2^{12}$	22528		334	9.0	$12\cdot 2^9+4\cdot 2^{12}$	22528		334	9.0
ilpC:12	Bits	$6\cdot 2^6+12\cdot 2^8+\underline{10\cdot 2^9}+\underline{8\cdot 2^6}$	9088		162	8.2	$4\cdot 2^4+15\cdot 2^6+\underline{12\cdot 2^7}+\underline{10\cdot 2^7}+\underline{9\cdot 2^4}$	3984		107	8.3
ilpC:12	LUTs	$6\cdot 2^6+12\cdot 2^8+\underline{10\cdot 2^9}+\underline{8\cdot 2^6}$	9088	172	149	8.2	$6\cdot 2^6+12\cdot 2^8+\underline{10\cdot 2^6}+7\cdot 2^6$	4544	96	113	7.4
table:14	Bits	$14\cdot 2^{11}+4\cdot 2^{14}$	94208		1271	14.4	$14\cdot 2^{11}+4\cdot 2^{14}$	94208		1271	14.4
ilpC:14	Bits	$8\cdot 2^7+14\cdot 2^{10}+\underline{11\cdot 2^9}+\underline{8\cdot 2^8}$	23040		439	10.4	$6\cdot 2^6+14\cdot 2^9+\underline{11\cdot 2^9}+\underline{9\cdot 2^8}$	15488		248	8.7
ilpC:14	LUTs	$8\cdot 2^7+14\cdot 2^{10}+\underline{11\cdot 2^9}+\underline{9\cdot 2^8}$	23296	398	396	9.8	$6\cdot 2^6+14\cdot 2^9+\underline{11\cdot 2^9}+\underline{9\cdot 2^8}$	15488	276	257	8.9
table:16	Bits	$16\cdot 2^{13}+4\cdot 2^{16}$	393216		5366	23.5	$16\cdot 2^{13}+4\cdot 2^{16}$	393216		5366	23.5
ilpC:16	Bits	$11\cdot 2^{10}+11\cdot 2^{13}+\underline{8\cdot 2^{11}}$	117760		1792	14.3	$8\cdot 2^7+15\cdot 2^{10}+\underline{12\cdot 2^{11}}+\underline{10\cdot 2^{11}}+\underline{8\cdot 2^{10}}$	69632		974	11.4
ilpC:16	LUTs	$11\cdot 2^{10}+11\cdot 2^{13}+\underline{8\cdot 2^{11}}$	117760	1861.5	1792	14.3	$10\cdot 2^9+13\cdot 2^{11}+\underline{11\cdot 2^{11}}+\underline{8\cdot 2^{10}}$	62464	1010	856	11.5

TABLE III: All synthesis results for correct rounding

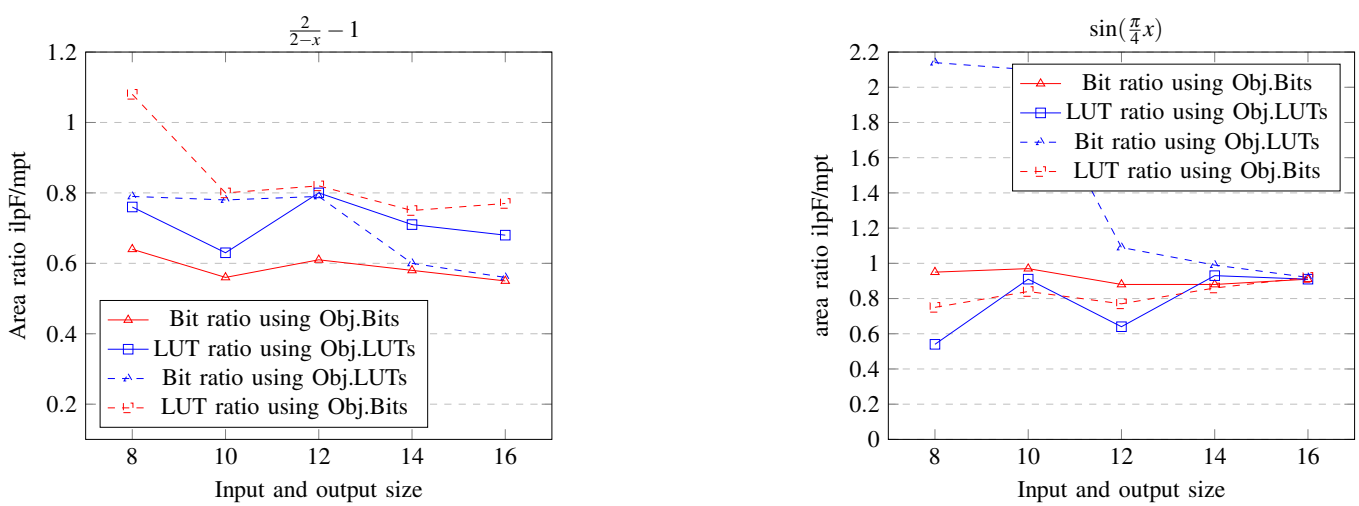


Fig. 8: Area improvement of the proposed method with respect to the state of the art in the faithful case (ratio ilpF/mpt)

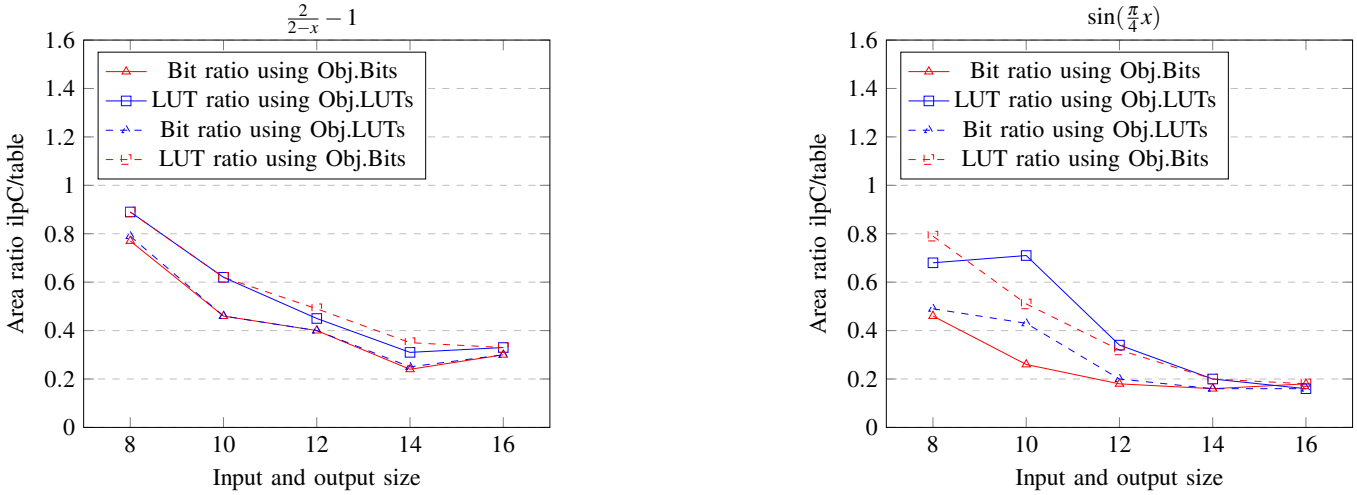


Fig. 9: Area improvement of the proposed method w.r.t. the state of the art in the correctly rounded case (ratio ilpC/table)

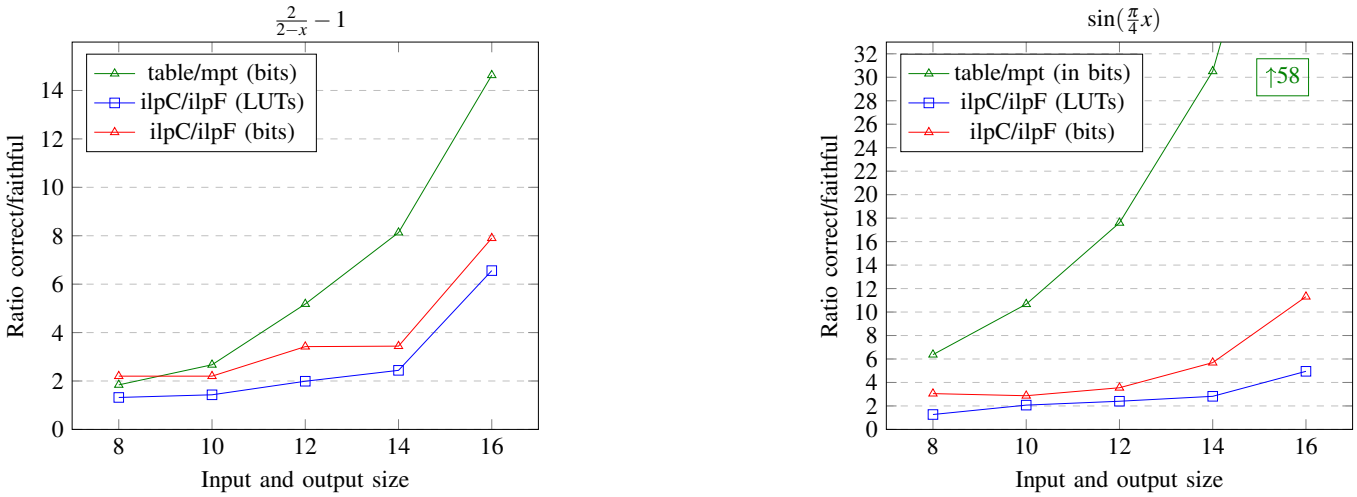


Fig. 10: Relative overhead of correct rounding (in green, the state of the art; in red and blue, the proposed method)

References

[1] D. A. Sunderland, R. A. Strauch, S. S. Wharfield, H. T. Peterson, and C. R. Role, "CMOS/SOS frequency synthesizer LSI circuit for spread spectrum communications," *IEEE Journal of Solid-State Circuits*, vol. 19, no. 4, pp. 497–506, 1984.

[2] D. Das Sarma and D. Matula, "Faithful bipartite ROM reciprocal tables," in *12th Symposium on Computer Arithmetic*, pp. 17–28, IEEE, 1995.

[3] M. Schulte and J. Stine, "Approximating elementary functions with symmetric bipartite tables," *IEEE Transactions on Computers*, vol. 48, no. 8, pp. 842–847, 1999.

[4] J. Stine and M. Schulte, "The symmetric table addition method for accurate function approximation," *Journal of VLSI Signal Processing*, vol. 21, no. 2, pp. 167–177, 1999.

[5] J.-M. Muller, "A few results on table-based methods," *Reliable Computing*, vol. 5, no. 3, pp. 279–288, 1999.

[6] F. de Dinechin and A. Tisserand, "Multipartite table methods," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 319–330, 2005.

[7] S.-F. Hsiao, P.-H. Wu, C.-S. Wen, and P. K. Meher, "Table size reduction methods for faithfully rounded lookup-table-based multiplierless function evaluation," *Transactions on Circuits and Systems II*, vol. 62, no. 5, pp. 466–470, 2015.

[8] S.-F. Hsiao, C.-S. Wen, Y.-H. Chen, and K.-C. Huang, "Hierarchical multipartite function evaluation," *Transactions on Computers*, vol. 66, no. 1, pp. 89–99, 2017.

[9] H. Hassler and N. Takagi, "Function evaluation by table look-up and addition," in *12th Symposium on Computer Arithmetic*, pp. 10–16, IEEE, 1995.

[10] D. De Caro, E. Napoli, D. Esposito, G. Castellano, N. Petra, and A. G. Strollo, "Minimizing coefficients wordlength for piecewise-polynomial hardware function evaluation with exact or faithful rounding," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 5, pp. 1187–1200, 2017.

[11] M. Christ, L. Forget, and F. de Dinechin, "Lossless differential table compression for hardware function evaluation," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, pp. 1642–1646, Mar. 2022.

[12] C. Iordache and D. W. Matula, "Analysis of reciprocal and square root reciprocal instructions in the AMD K6-2 implementation of 3DNow!," *Electronic Notes in Theoretical Computer Science*, vol. 24, 1999.

[13] M. Kumm and J. Kappauf, "Advanced compressor tree synthesis for FPGAs," *IEEE Transactions on Computers*, vol. 67, no. 8, pp. 1078–1091, 2018.

[14] Y. Yuan, L. Tu, K. Huang, X. Zhang, T. Zhang, D. Chen, and Z. Wang, "Area optimized synthesis of compressor trees on Xilinx FPGAs using generalized parallel counters," *IEEE Access*, vol. 7, pp. 134815–134827, 2019.