

Lossless Differential Table Compression for Hardware Function Evaluation

Maxime Christ, Luc Forget, Florent de Dinechin

▶ **To cite this version:**

Maxime Christ, Luc Forget, Florent de Dinechin. Lossless Differential Table Compression for Hardware Function Evaluation. IEEE Transactions on Circuits and Systems II: Express Briefs, 2021, 10.1109/TCSII.2021.3131405 . hal-03040364v2

HAL Id: hal-03040364
https://hal.science/hal-03040364v2

Submitted on 6 Jan 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.

Lossless Differential Table Compression for Hardware Function Evaluation

Maxime Christ, Luc Forget, Florent de Dinechin

***Abstract*—Hsiao *et al.* recently introduced, in the context of multipartite table methods, a lossless compression technique that replaces a table of numerical values with two smaller tables and one addition. The present work shows that this technique has many more applications than originally published, and that in many of these applications the addition is for free in practice. It also improves this technique and the resulting architecture by exposing a wider implementation space, and an exhaustive but fast algorithm exploring this space. These contributions are implemented in the open-source FloPoCo core generator and evaluated on FPGA and ASIC, reducing area up to a factor 2.**

***Index Terms*—Table of numerical values, hardware function evaluation, compression, computer arithmetic, ASIC, FPGA.**

I. Introduction

Tables of precomputed values are pervasive in the design of application-specific hardware, especially in the field of elementary function evaluation [1], [2]. For low precisions (typically up to 12 bits), a look-up table may store the value of a function for all the possible input values. For larger precisions, many evaluation methods may be used [1], [2]. These methods often rely on tables of precomputed values [3], [4], [5], [6], [7], [8]. Such table-based methods expose a trade-off between storage and computation. This enables FPGA designers to finely tune their architecture to the target device, and ASIC designers to match the silicon budget or performance requirements of an application.

Hsiao *et al.* introduced [6] then improved [7] a technique for compressing one specific table appearing in multipartite table methods [3]. This *lossless differential table compression* (LDTC) replaces one table with two smaller tables and an addition (Fig. 1). The present article extends this work in several ways.

A first contribution is, in Section II, an improvement to the compression method itself: the space of compression opportunities is wider than previous works suggest. The optimal can be found by a simple and fast exhaustive exploration.

A second contribution is to show in Section III that this technique is not limited to multipartite table methods: it is applicable as soon as the tabulated function presents small local variations, which is a very common case. Although LDTC was developed for low-precision function evaluation (up to 24 bits), it actually improves most function evaluation methods, including those that scale to 64-bit precision and beyond. It could even be used in some software contexts. Besides, since this compression is errorless, it is very easy to plug into existing table-based methods, as illustrated with examples in the open-source core generator FloPoCo.

A last contribution in Section IV is the observation that in many of these applications, LDTC is lossless in terms of functionality, but also in terms of performance. Indeed, the addition in LDTC adds two numbers with only a few bits of overlap (Fig. 1, Fig. 4). When the table value is itself added to a bit array [9] to be computed thanks to a compressor tree [10], [11], then the area overhead will be very little, and there will usually be no timing overhead.

Section V gathers experimental results that support all the previous claims.

TABLE I
Notations used in this article

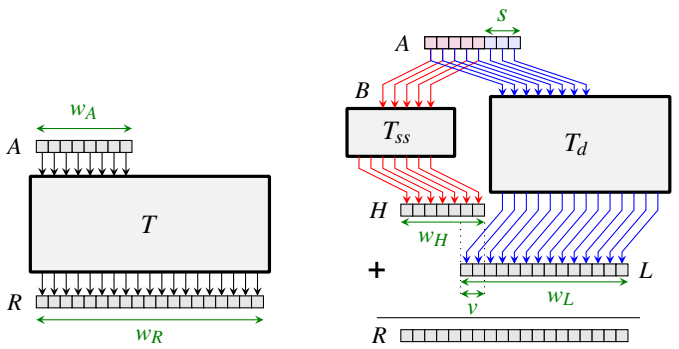
original table	$T : A \mapsto R$
input and output sizes	w_A, w_R
subsampling table	$T_{ss} : B \mapsto H$
input and output sizes	$w_B = w_A - s$, w_H
difference table	$T_d : A \mapsto L$
input and output sizes	w_A, w_L
number of overlap bits	$v = w_H + w_L - w_R$

II. Lossless differential table compression

Fig. 1(a) shows an uncompressed table T with $w_A = 8$ address bits and $w_R = 19$ output bits. Informally, a table T has some potential for the compression studied here if the values stored at neighbouring addresses present small variations with respect to the full output range of the table[1]. For instance, the TIV (Table of Initial Values) of the original article [6] samples a continuous and differentiable function at regularly spaced points. What is important, however, is not the possible mathematical properties (here continuity) of the underlying function, but the "small local variations" property of the discrete table. For illustration, Fig. 3 plots the content of two tables that are the result of a numerical optimization process [12]. There is no closed-form real function of which such a table is a sampling, the content of the C_3 table is not even monotonic, and still these tables are perfectly suitable for the compression studied here.

Maxime Christ is with Université Grenoble Alpes and CITI-Lab, Univ. Lyon, INSA-Lyon, Inria.
Luc Forget and Florent de Dinechin are with CITI-Lab, Univ. Lyon, INSA-Lyon, Inria.
{Maxime.Christ, Luc.Forget, Florent.de-Dinechin}@insa-lyon.fr.
This work was supported by the MIAI and the ANR Imprenum project.

[1]A more formal but less intuitive definition will come in Section II-C: a table T can be compressed if Algorithm 1, presented there, succeeds.



(a) Without compression (b) With compression

Fig. 1. Table compression example (here for the TIV of a multipartite architecture for $\sin(\frac{\pi}{4}x)$ on $[0,1)$ with 16-bit inputs and outputs)

A. Previous work

The core idea [6] of LDTC is the following. The original table T is sub-sampled by a factor 2^s, which gives a sub-sampling table T_{ss}. Obviously, T_{ss} is smaller than T since it has fewer entries (2^{w_A-s} instead of 2^{w_A}). Each value of the original table is then reconstructed by adding, to one entry of T_{ss}, the difference between this entry and the original value of T. This difference is stored in a second table T_d of 2^{w_A} entries. T_d has as many entries as the original table T, but its output range w_L is smaller than that of T thanks to the "small local variations" property of T: indeed T_d stores local variations. Hence T_d has fewer output bits than the original table, and is therefore also smaller. There is a compression as soon as the sum of the sizes of the two smaller tables is smaller than the original size of T [6]. Reconstructing the value of T requires an addition, whose architectural cost will be discussed in Section IV.

In all the following, we call a *slice* of T the subset of 2^s consecutive values to be reconstructed from one value of T_{ss}. If built as exposed previously, T_d systematically has 2^s entries equal to 0: one for each slice. These systematic zeroes suggest that a further optimization is possible. Indeed, an improvement in [7] is to add, to each entry of T_d, the value of the $w_R - w_L$ least significant bits (LSB) of the corresponding T_{ss} entry. Thus, these bits can be removed from T_{ss}, reducing its output size by $w_R - w_L$ bits. However, we now have larger values in T_d: in some cases this adds one bit (overflow bit) to the output size of T_d.

B. A wider implementation space

The possible overflow bit in T_d is expensive, since it is added to 2^{w_A} entries. In the present work, instead of removing the maximum number of LSB output bits from T_{ss} as in [7], we consider leaving some of these bits when it allows to avoid the overflow bit in T_d. If k extra output bits in T_{ss} allow for a T_d without overflow, the extra cost is $k \times 2^s$ bits and the benefit is 2^{w_A} bits, so there is a potential net gain in storage.

Conversely, once we acknowledge that T_d may overflow and that its output size w_L must be enlarged, it is worth attempting to reduce w_H at the LSB to benefit from the new freedom that a wider w_L provides.

To capture all these cases, for a given table T with its input size w_A and output size w_R, a compression parameter vector is defined as the triplet (s, w_H, w_L) shown in Fig. 1. A vector is valid if it is possible to achieve LDTC with these parameters. A vector also has an implementation cost, estimated thanks to a cost function $c(w_A, s, w_H, w_L)$, further discussed in Section II-D.

The implementation space thus defined is a strict superset of the one explored in [7]. In particular, as Section V will show, the optimal solution often shows $v = 2$ bits of overlap between H and L (see Figures 1 and 4), and is therefore out of the space explored by previous approaches [6], [7].

C. Improved LDTC optimization algorithm

A generic LDTC optimization is then provided by Algorithm 1. It simply enumerates this parameter space, and selects among the valid vectors the one with the smallest cost. This space is fairly small since s, w_H, and w_L are numbers of bits.

Algorithm 1: Generic LDTC optimization

```
function optimizeLDTC(T, w_A, w_R)
    bestVector ← (0, w_R, 0) ;                    // no compression
    bestCost ← c(bestVector) ;
    forall (s, w_H, w_L) do
        cost ← c(w_A, s, w_H, w_L);
        if cost < bestCost then
            if isValid(T, w_A, w_R, s, w_H, w_L) then
                bestCost ← cost;
                bestVector ← (s, w_H, w_L);
            end if
        end if
    end forall
    return bestVector
```

Actually, Algorithm 1 first filters by cost, then by validity, because cost (see Section II-D) is faster to evaluate than validity. Algorithm 2 determines if a parameter vector is valid.

Algorithm 2: Is a parameter vector valid ?

```
function isValid(T, w_A, w_R, s, w_H, w_L)
    for B ∈ (0, 1, ..., 2^(w_A−s) − 1) ;          // loop on slices
     do
        S ← {T[j]}_{j∈{B·2^s ... (B+1)·2^s−1}} ;   // slice
        M ← max(S) ;                               // max on slice
        m ← min(S) ;                               // min on slice
        mask ← 2^(w_R−w_H) − 1 ;
        H ← m − (m & mask) ;                       // w_H upper bits of m
        M_low ← M − H ;                            // max diff value on this slice
        if M_low ≥ 2^w_L then
            return false ;                         // one slice won't fit: exit with false
        end if
    end for
    return true
```

Note that Algorithm 2 is faster than attempting to fill the tables: it only needs the max and min of T on each slice, which can be computed only once for each value of s, and

memoized. Therefore one invocation of Algorithm 2 requires time proportional to 2^{w_A-s}, not to 2^{w_A}.

Altogether, the exploration of this parameter space on current computers is almost instantaneous for any size for which tabulation is practical.

D. Cost functions

Table II gives possible ways of evaluating the hardware cost $c^{\text{table}}(m,n)$ of a table with m input bits and n output bits. Most previous works [3], [6], [7] use $c^{\text{table}}_{\text{bit}}(m,n)$, which counts the total number of stored bits. On FPGAs, $c^{\text{table}}_{\text{LUT}}(m,n)$ estimates the number of FPGA architectural LUTs with ℓ inputs. This model is both pessimistic (it ignores the optimizations performed by synthesis tools) and optimistic (it doesn't count LUTs used as address decoding multiplexers for large m), but it is accurate for small tables. The third function, $c^{\text{table}}_{\text{SC}}(m,n)$, defined empirically [13], estimates the cost of a table implemented in ASIC as standard cells.

The cost function $c(w_A,s,w_H,w_L)$ used in Algorithm 1 is the sum of the cost of T_{ss}, which is $c^{\text{table}}(w_A-s,w_H)$, the cost of T_d, which is $c^{\text{table}}(w_A,w_L)$, and the cost of the addition $c^{\text{add}}(w_H,w_L)$ converted to the relevant unit from Table II. This addition cost will be further discussed in Section IV.

TABLE II
POSSIBLE TABLE COST FUNCTIONS

number of bits	$c^{\text{table}}_{\text{bit}}(m,n) = 2^m \times n$
number of FPGA LUTs	$c^{\text{table}}_{\text{LUT}}(m,n) = 2^{\min(m-\ell,0)} \times n$
ASIC standard cells [13]	$c^{\text{table}}_{\text{SC}}(m,n) = 2^{0.65\min(m,n)} \times 2^{0.19\lvert m-n\rvert}$

III. A REVIEW OF APPLICATIONS

This section gives a non-exhaustive list of applications of the LDTC technique to function evaluation [1], [2] beyond the original multipartite approximation. Applications probably also exist beyond function evaluation.

First, LDTC works, and even works extremely well, for plain function tables with $w_A = w_R$. Such tables are routinely used for very low precisions (up to 12 bits). Table III shows that the gain may in such cases exceed 50%.

For larger precisions, approximation techniques must be used. Many generic function evaluation methods (including the multipartite methods) are variations or refinements of piecewise polynomial approximation. Fig. 2 shows a typical uniform piecewise approximation architecture [14], [15], [8]. The input domain of the function is decomposed in 2^{w_A} segments of identical size by a simple splitting of the input word X in its w_A leading bits (which become the segment address A) and its remaining least significant bits (which become the index Y within a segment). On each segment, a good polynomial approximation is precomputed and its coefficients C_i are stored in a table indexed by A. It is possible to use more complex architectures where the segments of the input domain may have have different sizes [5], [16], but there is always a coefficient table.

For LDTC, instead of the wide table of Fig. 2, we consider as many independent tables as there are coefficients. It turns

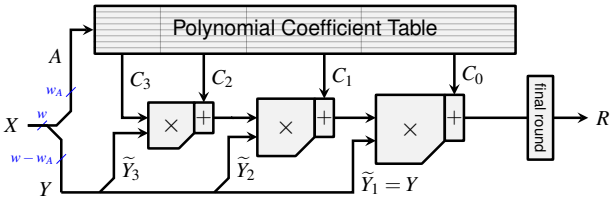


Fig. 2. A fixed-point polynomial evaluator, using uniform segmentation and a Horner scheme with truncated multipliers – see [14] for more details.

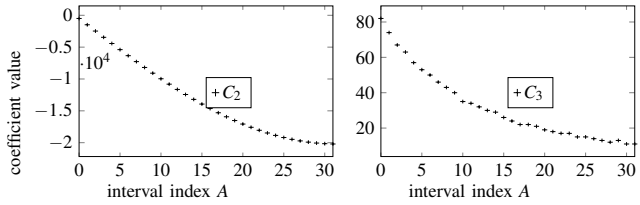


Fig. 3. Plot of all the degree-2 and degree-3 coefficients (the values of C_2 and C_3 in Fig. 2) of a 24-bit, degree 3 piecewise ($w_A = 5$) polynomial approximation to $\log(1+x)$ when using machine-efficient polynomials [12].

out that each coefficient table has the property of small local variations. This is an indirect consequence of a requirement of all these approximation methods, namely that the function must be differentiable up to a certain order. For instance, the degree-one coefficient C_1 is closely related to the derivative of the function, taken somewhere in each segment. As long as the derivative itself is continuous, $C_1(A)$ will present small local variations. For illustration, Fig. 3 shows $C_2(A)$ and $C_3(A)$ for a degree-3 approximation to $\log(1+x)$, and Table III shows the compression achieved in each of the coefficients of a degree-2 approximation to the sine function (all obtained using FloPoCo's `FixFunctionByPiecewisePoly` generator).

Another large class of applications consists in function-specific range reduction algorithms that rely on large tables of precomputed values. It was pioneered in software by Tang [17] then used in hardware to implement e.g. exponential [18, Fig. 2], [19, Fig. 10], logarithm [19, Fig. 6] or trigonometric functions [20, Fig. 2]. These tables all present small local variations and are suitable for LDTC, as illustrated by Table III on the e^A table of [18] (obtained using FloPoCo's `FPExp`).

Such range reduction techniques may be used iteratively, as in most CORDIC variants [1], in which case the table is addressed by the iteration index. It is unclear if there is compression potential in this case, however there is in high-radix iterative algorithms [1], [21, Fig. 1].

Finally, table-based methods have been also used to implement multiplication by constants [22], [23], [24] which are themselves used in elementary function implementations [18, Fig. 2], [19, Fig. 6], [20, Fig. 2]. Again the tables there are perfectly suited to LDTC. However, it is less obvious here that this potential can be exploited, as the tables are already finely tailored to the LUT-based logic of the FPGAs [23], [24].

IV. WITH COMPRESSION TREES, LDTC IS FOR FREE

If an adder is used to compute the sum, it should be obvious from Fig. 1(b) that the adder size is only w_H bits (the $w_R - w_H$ lower bits of the sum are those of L).

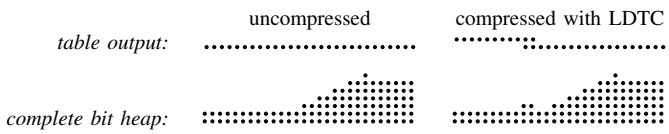


Fig. 4. The dot-diagram overhead of lossless table compression, here for the 24-bit multipartite implementation of $\sin(\frac{\pi}{4}x)$

However, in many of the applications reviewed in Section III, the table result is added to a value that is itself computed by a compression tree. This is the case in the original multipartite method (where it is added to multi-operand addition) [3, Fig. 7]. It is also the case for all the coefficients except C_3 in Fig. 2: in the Horner evaluation scheme [1] used there, each C_i is added to a product, and the best way to implement this product is also a compression tree [25], [11] (if a parallel evaluation scheme is used [8] only C_0 is added to a product). The table outputs are also summed in table-based constant multiplication techniques [23], [24], and in many other cases (e.g. [19, Fig. 6] where the tabulated $E \times \log(2)$ is added to another term).

In all these cases, thanks to merged arithmetic [9], the addition adds only $v = w_H + w_L - w_R$ bits to the bit array. This is illustrated in Fig. 4. The area cost of the addition therefore becomes proportional to the overlap v, in other words negligible. Furthermore, as long as the addition of these v bits does not entail one more compression stage [11], the delay overhead will be zero. It is the case in Fig. 4.

V. Evaluation

LDTC has been integrated in FloPoCo (git master branch), where it will benefit to all the table-based operators. Table III reports the compression rate in terms of bit count for a representative range of application (see section III). Compression ratio up to 0.36 are possible, to be be compared to the best ratio of 0.74 in the previous state of the art [7, Table VII].

A general observation on Table III is that the best compression ratios are observed for functions with the smaller difference between w_R and w_A. The intuition here is that the larger table is always T_d, and that T_d has very few output bits when $w_R \approx w_A$. Note that LDTC even works when $w_R < w_A$. For plain tables, the compression ratio improves with the size of the initial table (see the pt lines in Table III).

Surprisingly, the optimal overlap is often $v = 2$ bits. However, this is not a strict rule, as the ptl:12 line shows.

Table IV compares the $c_{\text{bit}}^{\text{table}}$ of the state of the art hierarchical multipartite methods including the best previous compression (hmp) [7, Tables 2 and 3] to that achieved by the (older, non hierarchical) multipartite implementation of FloPoCo, enhanced with LDTC (mpt+LDTC). The function used here is $sin(\frac{\pi}{4}x)$, the only one on which we could reproduce the results of [7]. There is one more addition in mpt+LDTC than in hmp, however the total number of bits input to the compressor tree is smaller for mpt+LDTC (on 16 bits, 20+12+10+8+6=56 bits for hmp, versus 7+14+10+8+6+4=49 bits for mpt+LDTC). Both observations also hold in the 24-bit case, not detailed for lack of space. This experiment suggests that integrating our improved LDTC to the hierarchical multipartite method will improve the state of the art of multipartite methods.

Results for FPGA and ASIC synthesis are reported in Tables V and VI respectively. All the LDTC were optimized using the $c_{\text{bit}}^{\text{table}}$ cost function, the other cost functions of Table II failing to provide any significant improvement so far.

The pt architectures include an adder which costs area and adds to the delay. Conversely, the mpt evaluators use a compressor tree, in which case Section IV claimed that the compression is for free, both in terms of area (see Figure 4) and delay. The ratios reported in Tables V and VI support this

TABLE III
Compression results in terms of bit storage

pt, ptl	plain tabulation of $\sin(\frac{\pi}{4}x)$ (pt) or $\log(1+x)$ (ptl) on $[0,1)$
mpt	multipartite approximation [6] to $\sin(\frac{\pi}{4}x)$ on $[0,1)$
mpti	Table of Initial Values (TIV) of mpt
ea	e^A table of [18], for single (sp) or double (dp) precision
C_i	coefficients of a degree-2 uniform piecewise approximation [14] to $sin(\frac{\pi}{4}x)$ on $[0,1)$ for 32-bit accuracy ($w_A = 9$)

	plain table	LDTC (ratio)	T_{ss}	T_d	v
pt:8	2,048 $8 \cdot 2^8$	992 (0.48)	224 $7 \cdot 2^5$	768 $3 \cdot 2^8$	2
pt:9	4,608 $9 \cdot 2^9$	2,048 (0.44)	512 $8 \cdot 2^6$	1,536 $3 \cdot 2^9$	2
pt:10	10,240 $10 \cdot 2^{10}$	4,224 (0.41)	1,152 $9 \cdot 2^7$	3,072 $3 \cdot 2^{10}$	2
pt:12	49,152 $12 \cdot 2^{12}$	17,920 (0.36)	5,632 $11 \cdot 2^9$	12,288 $3 \cdot 2^{12}$	2
ptl:12	49,152 $12 \cdot 2^{12}$	18,432 (0.38)	6,144 $12 \cdot 2^9$	12,288 $3 \cdot 2^{12}$	3
mpti:12	960 $15 \cdot 2^6$	800 (0.83)	96 $6 \cdot 2^4$	704 $11 \cdot 2^6$	2
mpti:14	2,048 $16 \cdot 2^7$	1,632 (0.8)	224 $7 \cdot 2^5$	1,408 $11 \cdot 2^7$	2
mpti:16	4,864 $19 \cdot 2^8$	3,808 (0.78)	224 $7 \cdot 2^5$	3,584 $14 \cdot 2^8$	2
mpti:20	24,576 $24 \cdot 2^{10}$	18,560 (0.76)	1,152 $9 \cdot 2^7$	17,408 $17 \cdot 2^{10}$	2
mpti:24	114,688 $28 \cdot 2^{12}$	83,456 (0.73)	5,632 $11 \cdot 2^9$	77,824 $19 \cdot 2^{12}$	2
ea:sp:10	28,672 $28 \cdot 2^{10}$	22,656 (0.79)	1,152 $9 \cdot 2^7$	21,504 $21 \cdot 2^{10}$	2
ea:dp:10	58,368 $57 \cdot 2^{10}$	52,352 (0.90)	1,152 $9 \cdot 2^7$	51,200 $50 \cdot 2^{10}$	2
ea:dp:12	233,472 $57 \cdot 2^{12}$	202,240 (0.87)	5,632 $11 \cdot 2^9$	196,608 $48 \cdot 2^{12}$	2
ea:dp:14	933,888 $57 \cdot 2^{14}$	780,288 (0.84)	26,624 $13 \cdot 2^{11}$	753,664 $46 \cdot 2^{14}$	2
C_0	17,920 $35 \cdot 2^9$	15,360 (0.86)	512 $8 \cdot 2^6$	14,848 $29 \cdot 2^9$	2
C_1	12,800 $25 \cdot 2^9$	9,792 (0.76)	576 $9 \cdot 2^6$	9,216 $18 \cdot 2^9$	2
C_2	6,656 $13 \cdot 2^9$	4,160 (0.62)	576 $9 \cdot 2^6$	3,584 $7 \cdot 2^9$	2
all C_i	$73 \cdot 2^9$	29,312 (0.78)	$26 \cdot 2^6$	$54 \cdot 2^9$	

TABLE IV
Comparison with the state of the art in multipartite methods

	16 bits	24 bits
hmp [7]	$6,272 = 20 \cdot 2^7 + 12 \cdot 2^7 + 10 \cdot 2^7 + 8 \cdot 2^6 + 6 \cdot 2^6$	166,528
mpt+LDTC	$6,752 = 7 \cdot 2^5 + 14 \cdot 2^8 + 10 \cdot 2^7 + 8 \cdot 2^7 + 6 \cdot 2^6 + 4 \cdot 2^6$	158,208

TABLE V
FPGA IMPLEMENTATION RESULTS FOR $\sin(\frac{\pi}{4}x)$ (ADDITION INCLUDED)

case	without LDTC		with LDTC (ratio)	
pt:8	27 LUT	5.5 ns	26 LUT (0.96)	6.4 ns (1.17)
pt:9	61 LUT	6.3 ns	46 LUT (0.75)	6.7 ns (1.08)
pt:10	134 LUT	6.8 ns	83 LUT (0.62)	7.5 ns (1.1)
pt:12	536 LUT	9.8 ns	287 LUT (0.54)	8.7 ns (0.89)
mpt:12	76 LUT	7.6 ns	73 LUT (0.96)	7.7 ns (1.02)
mpt:14	102 LUT	8.1 ns	98 LUT (0.96)	8.1 ns (1.0)
mpt:16	184 LUT	8.9 ns	190 LUT (1.03)	9.3 ns (1.04)
mpt:20	676 LUT	12.8 ns	637 LUT (0.94)	12.5 ns (0.98)
mpt:24	2489 LUT	18.1 ns	2322 LUT (0.93)	16.1 ns (0.89)

Results after implementation on Kintex7 using Vivado 2020.2. The compressor tree used for mpt is FloPoCo's default.

TABLE VI
ASIC IMPLEMENTATION RESULTS FOR $\sin(\frac{\pi}{4}x)$ (ADDITION INCLUDED)

case	LDTC results (ratio WRT uncompressed table)		
pt:8	130 μm^2 (0.82)	0.3 ns (1.44)	0.12 mW (0.91)
pt:9	184 μm^2 (0.56)	0.3 ns (1.12)	0.16 mW (0.62)
pt:10	298 μm^2 (0.54)	0.4 ns (0.82)	0.26 mW (0.58)
pt:12	1,190 μm^2 (0.63)	0.6 ns (0.94)	1.06 mW (0.66)
mpt:12	269 μm^2 (0.98)	0.6 ns (1.00)	0.34 mW (0.96)
mpt:14	424 μm^2 (0.98)	0.7 ns (1.00)	0.50 mW (0.96)
mpt:16	793 μm^2 (0.92)	0.8 ns (1.00)	0.88 mW (0.88)
mpt:20	2,888 μm^2 (0.84)	1.2 ns (1.12)	2.76 mW (0.67)
mpt:24	11,801 μm^2 (0.87)	1.8 ns (1.01)	11.13 mW (0.72)

Synthesis results obtained through Synopsys design compiler using a 28nm FDSOI standard cell library from STMicroelectronics. Timing includes an estimation of interconnect delays. The compressor tree used for mpt is FloPoCo's default. Uncompressed synthesis results are not shown due to lack of space, only the ratio is given.

claim[2]. The power consumption is essentially proportional to the area, and is therefore improved by LDTC.

In both Tables V and VI, the area compression ratios are not as good as those of Table III: this shows that the logic optimization of synthesis tools discover some of the compression opportunity exploited by LDTC [13].

We do not report results targetting FPGAs with block RAM, since actual savings will depend on the block RAM capabilities of the target, which are very discrete (e.g. M20k blocks on Altera/Intel devices can be configured as $2^9 \times 40$, $2^{10} \times 20$, or $2^{11} \times 10$ bits, while the Xilinx/AMD 36kbit blocks can do from $2^9 \times 72$ to $2^{15} \times 1$). The larger the table, the higher the chances that LDTC is useful in this case.

VI. CONCLUSION

This work adds one optimization technique to the bag of tricks of arithmetic designers: lossless differential table compression can reduce up to a factor two the storage requirement of most tables used in function evaluation, at the cost of one small integer addition that can be hidden for free in an existing compressor tree. The optimal compression can be found by exhaustive enumeration. This technique is available in the open-source FloPoCo core generator.

Acknowledgement: Many thanks to Frédéric Pétrot for his comments and his help with ASIC synthesis.

[2]The mpt timing variations in Table V are routing artifacts – even the improvement for mpt:24 is actually due to variations in input buffer net delay.

REFERENCES

[1] J.-M. Muller, *Elementary functions, algorithms and implementation, 3rd Edition*. Birkhaüser Boston, 2016.

[2] A. Omondi, *Computer-Hardware Evaluation of Mathematical Functions*. Imperial College Press, 2016.

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

[4] J. Detrey and F. de Dinechin, "Table-based polynomials for fast hardware function evaluation," in *Application-specific Systems, Architectures and Processors*. IEEE, 2005, pp. 328–333.

[5] D.-U. Lee, P. Cheung, W. Luk, and J. Villasenor, "Hierarchical segmentation schemes for function evaluation," *IEEE Transactions on VLSI Systems*, vol. 17, no. 1, 2009.

[6] 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.

[7] 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.

[8] 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.

[9] E. E. Swartzlander, "Merged arithmetic," *IEEE Transactions on Computers*, vol. C-29, no. 10, pp. 946 –950, 1980.

[10] H. Parendeh-Afshar, A. Neogy, P. Brisk, and P. Ienne, "Compressor tree synthesis on commercial high-performance FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 4, no. 4, 2011.

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

[12] N. Brisebarre and S. Chevillard, "Efficient polynomial L^∞- approximations," in *18th Symposium on Computer Arithmetic*. IEEE, 2007, pp. 169–176.

[13] O. Gustafsson and K. Johansson, "An empirical study on standard cell synthesis of elementary function lookup tables," *Asilomar Conference on Signals, Systems and Computers*, pp. 1810–1813, 2008.

[14] F. de Dinechin, M. Joldes, and B. Pasca, "Automatic generation of polynomial-based hardware architectures for function evaluation," in *Application-specific Systems, Architectures and Processors*. IEEE, 2010.

[15] S.-F. Hsiao, H.-J. Ko, and C.-S. Wen, "Two-level hardware function evaluation based on correction of normalized piecewise difference functions," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 59, no. 5, pp. 292–296, 2012.

[16] D. Chen and S.-B. Ko, "A dynamic non-uniform segmentation method for first-order polynomial function evaluation," *Microprocessors and Microsystems*, vol. 36, pp. 324–332, 2012.

[17] P. T. P. Tang, "Table-driven implementation of the exponential function in IEEE floating-point arithmetic," *ACM Transactions on Mathematical Software*, vol. 15, no. 2, pp. 144–157, 1989.

[18] F. de Dinechin and B. Pasca, "Floating-point exponential functions for DSP-enabled FPGAs," in *Field Programmable Technologies*, Dec. 2010, pp. 110–117.

[19] M. Langhammer and B. Pasca, "Single precision logarithm and exponential architectures for hard floating-point enabled FPGAs," *IEEE Transactions on Computers*, vol. 66, no. 12, pp. 2031–2043, 2017.

[20] F. de Dinechin, M. Istoan, and G. Sergent, "Fixed-point trigonometric functions on FPGAs," *SIGARCH Computer Architecture News*, vol. 41, no. 5, pp. 83–88, 2013.

[21] J.-A. Piñeiro, M. Ercegovac, and J. Bruguera, "High-radix logarithm with selection by rounding: Algorithm and implementation," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 40, no. 1, pp. 109–123, 2005.

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

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

[24] F. de Dinechin, S.-I. Filip, L. Forget, and M. Kumm, "Table-based versus shift-and-add constant multipliers for FPGAs," in *26th IEEE Symposium of Computer Arithmetic (ARITH-26)*, Jun. 2019.

[25] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2004.