Towards Hardware IIR Filters Computing Just Right: Direct Form I Case Study

Anastasia Volkova, Matei Istoan, Florent de Dinechin, Thibault Hilaire

Towards Hardware IIR Filters Computing Just Right: Direct Form I Case Study

Anastasia Volkova†, Matei Istoan*, Florent de Dinechin*, Thibault Hilaire‡
*Université de Lyon, INRIA, INSA-Lyon, CITI-INRIA, F-69621, Villeurbanne, France
† INRIA, Université de Lyon – Laboratoire LIP (CNRS, Inria, ENS de Lyon, UCBL), Lyon, France
‡INRIA, Université Paris-Saclay, F-91120 Palaiseau, France
Sorbonne Universités,UMPC Univ. Paris 06, UMR 7606, LIP6, F-75005 Paris, France

***Abstract*—Linear Time Invariant (LTI) filters are often specified and simulated using high-precision software, before being implemented in low-precision fixed-point hardware. A problem is that the hardware does not behave exactly as the simulation due to quantization and rounding issues. This article advocates the construction of LTI architectures that behave as if the computation was performed with infinite accuracy, then rounded only once to the low-precision output format. From this minimalist specification, it is possible to deduce the optimal values of many architectural parameters, including all the internal data formats. This requires a detailed error analysis that captures not only the rounding errors but also their infinite accumulation in recursive filters. This error analysis then guides the design of hardware satisfying the accuracy specification at the minimal hardware cost. We detail this generic methodology for the case of low-precision LTI filters in the Direct Form I implemented in FPGA logic. This approach is demonstrated by a fully automated and open-source architecture generator tool, and validated on a range of Infinite Impulse Response filters.**

I. Introduction

This article addresses the automatic implementation of Linear Time Invariant (LTI) digital filters. Such filters are ubiquitous in signal processing and control, and are typically defined in the frequency domain as a transfer function $\mathcal{H}(z)$:

$$\mathcal{H}(z) = \frac{\sum_{i=0}^{n_b} b_i z^{-i}}{1 + \sum_{i=1}^{n_a} a_i z^{-i}}, \quad \forall z \in \mathbb{C}. \tag{1}$$

where $n_a \geq n_b$ and n_a is the order of the filter.

In the time domain, a filter specified with $\mathcal{H}(z)$ may be evaluated with various algorithms [1]. In this work we consider the Direct Form I (DFI) [1] algorithm that relates the output signal $y(k)$ and the input signal $u(k)$ in the following way:

$$y(k) = \sum_{i=0}^{n_b} b_i u(k-i) - \sum_{i=1}^{n_a} a_i y(k-i). \tag{2}$$

The DFI algorithm is the simplest algorithm for the evaluation of Infinite Impulse Response (IIR) filters, since it directly uses the coefficients of the transfer function. Small-order DFIs are often used as basic building blocks for second-order sections algorithms [1]. Due to its high numerical sensitivity, DFI is rarely used for high order filters. A contribution of this article is a methodology that protects designers from such sensitivity issues. This methodology is not restricted to DFI and could be applied to other filter algorithms (cascade and/or parallel decomposition, state-space, ρ-operator forms [2], etc.). This article focuses on DFI because of its simplicity.

This work is partly supported by the MetaLibm project (ANR-13-INSE-0007) of the French *Agence Nationale de la Recherche*.

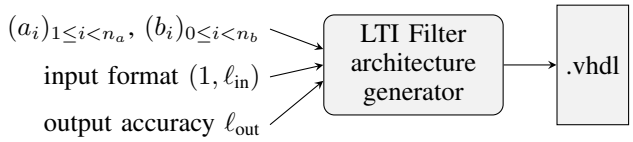

Fig. 1. Interface of the proposed tool.

Equation (1) or (2), along with a definition of each coefficient a_i and b_i, constitute the *mathematical specification* of the filter. This article deals with the *implementation* of such a specification as fixed-point hardware operating on low- to moderate-precision data (typically 8 to 24 bits).

To specify such an implementation, a designer needs to define several parameters on top of the mathematical specification. Obviously, he needs to define the finite-precision input and output formats. He also needs to make several architecture choices, some of which will impact the accuracy of the computation.

For instance, each real-valued coefficient must be rounded to some internal machine format. A naive choice is to round the coefficients to the input/output format, but then, for sensitive filters, the result can become very inaccurate. Some design tools for filter synthesis let the designer chose an extended internal precision. The risk is then to obtain an architecture that internally computes more accurately than it can output, thus wasting area, time and power.

The main contribution of this article is to show that such design decisions can be automated into a push-button tool with the minimalistic interface from Figure 1.

The method for this is to generate DFI architectures which obey the following specification: they behave *as if the computation was performed with infinite accuracy with respect to* (2)*, then rounded only once to the low-precision output format*. This is achieved by a complete and rigorous analysis of the rounding errors and their propagation through the

feedback loop. This error analysis enables the computation of architectural parameters that ensure the specification at the minimal hardware cost.

With such a tool, the designer may focus on those design parameters which are relevant: the (real) coefficients, and the input/output formats.

All the above is demonstrated in this paper as an open source tool that builds DFI implementations optimized for a particular hardware target: FPGAs based on Look-Up Tables (LUTs). Built upon the FloPoCo project[1], this tool automatically generates VHDL for DFI filters from the specification of Fig. 1.

The automated and reliable error analysis builds upon recent work on bounding error propagation through filters [3], [4].

This new tool also incorporates several architectural novelties. The constant multipliers are built using an evolution of the KCM algorithm [5], [6] that manages multiplications by a real constant without needing to truncate it first [7]. The summation is efficiently performed thanks to the BitHeap framework recently introduced in FloPoCo [8], [9]. These technical choices lead to logic-only architectures suited even to low-end FPGAs, a choice motivated by work on implementing the ZigBee protocol standard [10]. However, the same philosophy could be used to build other architecture generators, for instance exploiting embedded multipliers and DSP blocks.

Section II provides some prerequisites on the target arithmetic and error models. A complete methodology on the rigorous error analysis is presented in the Section III. Section IV gives details on the architecture of the arithmetic units and their rounding error analysis. Finally, Section V demonstrates implementation results, and Section VI discusses the perspectives of this work.

II. Definitions and notations

A. Fixed-point formats

There are many standards for representing fixed-point data. The one we use in this work is inspired by the VHDL `sfixed` standard. For simplicity we only deal with signed fixed-point numbers, classically represented in two's complement. As illustrated by Figure 2, a fixed-point format is then fully specified by two integers (m, ℓ) that denote the positions of the most and least significant bits (MSB and LSB) respectively. Both m and ℓ can be negative if the format includes fractional bits. The weight of the bit at position i is always 2^i, except for the bit at position m, whose weight is -2^m (due to two's complement representation). The LSB position ℓ denotes the *precision* of the format. The MSB position denotes its *range*. The wordlength of the fixed-point number in the format (m, ℓ) is $m - \ell + 1$. Notate, that the range of numbers that can be represented with the format (m, ℓ) is $[-2^m, 2^m - 2^\ell]$. For instance, for a signed fixed-point format representing numbers in $(-1, 1)$ on 16 bits, we have one "sign bit" to the left of the point and 15 bits to the right, so $(m, \ell) = (0, -15)$. Then, we can represent numbers in $[-1, 1 - 2^{-15}]$ with a step 2^{-15}.

[1] http://flopoco.gforge.inria.fr/, version 4.1.3.

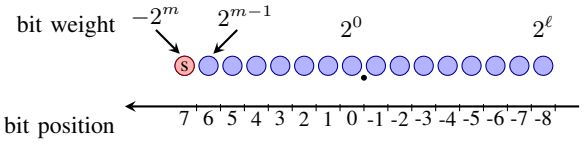

Fig. 2. The bits of a two's complement fixed-point format, here $(m, \ell) = (7, -8)$.

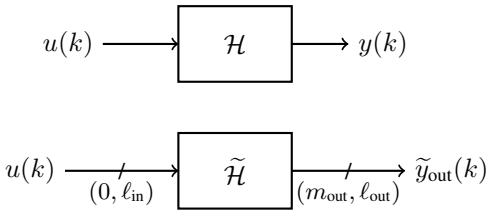

Fig. 3. The ideal filter (top) and its implementation (bottom)

B. Approximations and errors

Due to the finite precision implementation, the exact filter $\mathcal{H}$, with exact output y, cannot usually be synthesized. An actually implemented filter will produce a finite precision output $\widetilde{y}_{\text{out}}$ (see Fig. 3). The overall error, noted ε_{out}, of an architecture that outputs a fixed-point result $\widetilde{y}_{\text{out}}$ is defined as the difference between the computed value and its mathematical specification:

$$\varepsilon_{\text{out}}(k) \quad = \quad \widetilde{y}_{\text{out}}(k) - y(k). \tag{3}$$

More generally, throughout the article, we denote ε (with some subscript) an error, which is always defined as the difference between a more accurate term and a less accurate one.

We also try to use tilded letters (e.g. $\widetilde{y}_{\text{out}}$ above) for approximate or rounded terms. This is but a convention, and the choice is not always obvious. For instance, the $u(k)$ in (2) are fixed-point inputs, and most certainly the result of some approximate measurement or computation. However, from the point of view of the architecture, inputs are given, so they are considered exact.

In all the following, we also note $\overline{\varepsilon}$ a bound on $\varepsilon(k)$, *i.e.* the maximum value of $|\varepsilon(k)|$ over time.

C. Perfect rounding vs. last-bit accuracy

The rounding of a real such as our ideal output y to the nearest fixed-point number of precision ℓ is denoted $\circ_\ell(y)$. In the worst case, it entails an error $|\circ_\ell(y(k)) - y(k)| \leq 2^{\ell-1}, \quad \forall k$. For instance, rounding a real to the nearest integer ($\ell = 0$) may entail an error up to $0.5 = 2^{-1}$. This is a limitation of the format itself. Therefore, the best we can do, when implementing (2) with a precision-ℓ output, is a *perfectly rounded* computation with an error bound $\overline{\varepsilon}_{\text{out}} = 2^{\ell-1}$.

Unfortunately, reaching perfect rounding accuracy may require arbitrary intermediate precision. This is not acceptable in an architecture. We therefore impose a slightly relaxed constraint: $\overline{\varepsilon}_{\text{out}} < 2^\ell$. We call this *last-bit accuracy*, because the error must be smaller than the value of the last (LSB) bit

of the result. It is sometimes called *faithful rounding* in the literature.

Considering that the output format implies that $\overline{\varepsilon}_{\text{out}} \geq 2^{\ell-1}$, it is still a tight specification. For instance, if the exact y happens to be a representable precision-ℓ number, then a last-bit accurate architecture will return exactly this value.

The main reason for chosing last-bit accuracy over perfect rounding is that, as will be shown in the sequel, it can be reached with very limited hardware overhead. Therefore, in terms of cost and efficiency, an architecture that is last-bit-accurate to ℓ bits makes more sense than a perfectly rounded architecture to $\ell - 1$ bits, for the same accuracy bound 2^{ℓ}.

The main conclusion of this discussion is the following: specifying the output precision (ℓ_{out} on Fig. 1) is enough to also specify the accuracy of the implementation.

This is a huge improvement over classical approaches, such as the various Matlab toolboxes that generate hardware filters. In such approaches, one must provide ℓ_{out} and various other parameters that impact the accuracy, then measures the resulting accuracy using simulation-based techniques whose reliability is highly input-dependant, and iterate until a satisfactory implementation has been reached. Not only is the proposed interface simpler, it also enables architecture optimization under a strict accuracy constraint. An optimal architecture will be an architecture that is accurate enough, but no more.

D. Worst-case peak gain of an LTI filter

To determine the MSB position of the output (m_{out}) and to perform the roundoff analysis, we need to capture the amplification of a signal by an LTI filter. This amplification can be computed using the so-called *Worst-Case Peak-Gain* (WCPG) [11], [12] measure through the following theorem.

Theorem 1 (Worst-Case Peak Gain)**.** *Let $\mathcal{H}$ be a stable[2] single-input single-output LTI filter. If for all possible $k \geq 0$ an input signal $u(k)$ is bounded in magnitude by $\bar{u}$, then the output $y(k)$ is bounded:*

$$\forall k, \quad |y(k)| \leq \bar{y} = \langle\!\langle \mathcal{H} \rangle\!\rangle \, \bar{u} \tag{4}$$

where $\langle\!\langle \mathcal{H} \rangle\!\rangle$ is the Worst-Case Peak Gain [11], [12] of the system. It can be computed as the ℓ_1-norm of the system's impulse response $h(k)$:

$$\langle\!\langle \mathcal{H} \rangle\!\rangle = ||h||_1 = \sum_{k=0}^{\infty} |h(k)| \tag{5}$$

Proof: The proof of the theorem comes directly from the expression of the filter's output through the convolution as

$$y(k) = \sum_{l=0}^{k} h(l)u(k-l). \tag{6}$$

■

Remark 1. *The bound $\langle\!\langle \mathcal{H} \rangle\!\rangle \, \overline{u}$ on the output is quite conservative in practice, but for any filter it is possible to construct a finite input signal $\{u(k)\}_{0 \leq k \leq K}$ that yields an output that approaches the $\langle\!\langle \mathcal{H} \rangle\!\rangle \, \bar{u}$ up to any arbitrarily small distance.*

Indeed, in

$$|y(k)| = \left| \sum_{l=0}^{k} h(l)u(k-l) \right| \leq \bar{u} \sum_{k=0}^{\infty} |h(k)| \tag{7}$$

we obtain the equality if the input $u(k)$ is such that

$$u(k) = \bar{u} \cdot sign(h(K-k)), \tag{8}$$

where $sign(x)$ returns ± 1 or 0 depending on the sign of x.

In this work, we compute WCPGs with arbitrary precision using the reliable algorithm presented in [3] and its fast but rigorous implementation[3].

III. Error analysis of direct-form LTI filter implementations

This section shows how to obtain an implementation of the mathematical definition (2) in fixed-point with last-bit accuracy on the computed result with respect to this mathematical definition. The two filters are exhibited on Fig. 3.

We remind the reader that since the considered filters are linear, we assume without loss of generality that the MSB of the input is equal to 0. Based on the Theorem 1, the MSB of the output m_{out} is defined by:

$$m_{\text{out}} = \lceil \log_2 \langle\!\langle \mathcal{H} \rangle\!\rangle \rceil \ . \tag{9}$$

Technically, it may happen, rarely, that rounding errors propagate all the way to the MSB. Since these errors will be bounded by $2^{\ell_{\text{out}}-1}$, the formula to be used is actually

$$m_{\text{out}} = \left\lceil \log_2 \left(\langle\!\langle \mathcal{H} \rangle\!\rangle + 2^{\ell_{\text{out}}-1} \right) \right\rceil \tag{10}$$

In addition, using the algorithm from [4], the computed MSB position may be guaranteed to never be underestimated.

In fixed-point arithmetic, instead of computing output $y(k)$, we will compute an approximation $\widetilde{y}(k)$ of the involved Sum of Product by Constants (SOPC) using some internal format $(m_{\text{out}}, \ell_{\text{ext}})$

$$\widetilde{y}(k) \approx \sum_{i=0}^{n_b} b_i u(k-i) - \sum_{i=1}^{n_a} a_i \widetilde{y}(k-i) \tag{11}$$

and the final output $\widetilde{y}_{\text{out}}(k)$ will be some rounding of this intermediate value $\widetilde{y}(k)$.

Remark 2. *Here we set the MSB of the internal format to be the same as that of the result (m_{out}). Some overflows may occur in the internal computation, but since the computation is performed modulo $2^{m_{out}}$, the final result will be correct.*

This computational scheme is summed up by the abstract architecture of Figure 4.

Formally, we refine the definition of the overall evaluation error as

$$\varepsilon_{\text{out}}(k) \quad = \quad \widetilde{y}_{\text{out}}(k) - y(k) \tag{12}$$

[2] i.e. poles of $H(z)$ are strictly in the unit circle.

[3] https://scm.gforge.inria.fr/anonscm/git/metalibm/wcpg.git

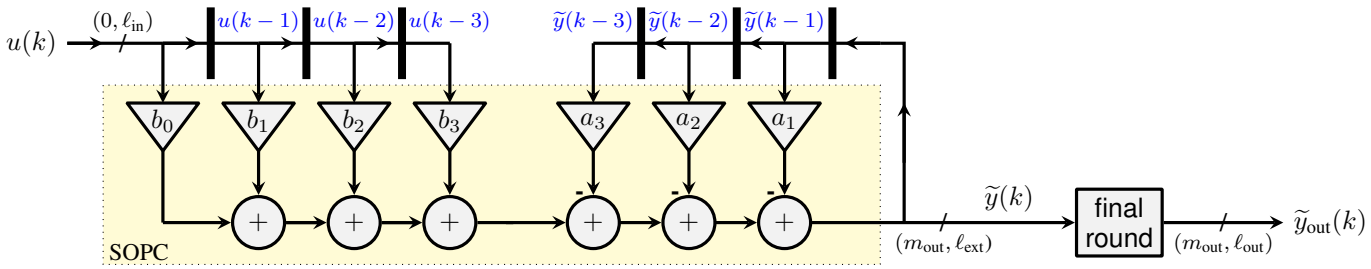

Fig. 4. Abstract architecture for the direct form realization of an LTI filter

Our goal is to detect all sources of errors and express them in terms of the choices of LSB positions. Then, under the constraints of the last-bit accuracy, i.e. $\varepsilon_{\text{out}}(k) < 2^{\ell_{\text{out}}}$, we will look for the optimal internal format.

Let us first decompose this error into its sources.

A. Final rounding of the internal format

The architecture needs to internally use a fixed-point format that offers extended precision with respect to the input/output format. This extended format $(m_{\text{out}}, \ell_{\text{ext}})$ offers additional LSB bits (sometimes called *guard bits*) in which rounding errors may accumulate without touching the output bits. The sequel will show more formally how to compute this extended format in an optimal way. Eventually we need to round the intermediate result in this extended format to the output format (in the "final round" box on Figure 4). This entails an additional error ε_{f}, formally defined as

$$\varepsilon_{\text{f}}(k) = \widetilde{y}_{\text{out}}(k) - \widetilde{y}(k) \quad . \tag{13}$$

This error may be bounded by $\overline{\varepsilon}_{\text{f}} = 2^{\ell_{\text{out}}-1}$, as round to nearest is easy to achieve here.

Remark that we feed back the intermediate result $\widetilde{y}(k)$ (on the extended format), not the output result $\widetilde{y}_{\text{out}}(k)$. This prevents an amplification of $\varepsilon_{\text{f}}(k)$ by the feedback loop that could compromise the goal of last-bit accuracy.

B. Rounding and quantization errors in the sum of products

As the coefficients a_i and b_i are real numbers, they must be rounded to some finite value (quantization) before the multiplication can take place. Then, the multiplication and the summation may themselves involve rounding errors. Managing all these rounding errors will be the subject of section IV, which will show how to build an architecture that achieves a given accuracy goal at the minimum cost. For now, we may summarize all these errors in a single term $\varepsilon_{\text{r}}(k)$ mathematically defined as

$$\varepsilon_{\text{r}}(k) \quad = \quad \widetilde{y}(k) - \left(\sum_{i=0}^{n_b} b_i u(k-i) - \sum_{i=1}^{n_a} a_i \widetilde{y}(k-i) \right) \tag{14}$$

This equation should be read as follows: $\varepsilon_{\text{r}}(k)$ measures how much a result $\widetilde{y}(k)$ computed by the SOPC architecture diverges from that computed by an ideal SOPC (that would use the infinitely accurate coefficients a_i and b_i, and be free of rounding errors), this ideal SOPC being applied on the same inputs $u(k-i)$ and $\widetilde{y}(k-i)$ as the architecture.

C. Error amplification in the feedback loop

The input signal $u(k)$ can be considered exact, in the sense that whatever error it may carry is not due to the filter under consideration. However, the feedback signal $\widetilde{y}(k)$ that is input to the computation (see Figure 4) differs from the ideal $y(k)$. Let us define $\varepsilon_{\text{t}}(k)$ as the error of $\widetilde{y}(k)$ with respect to $y(k)$:

$$\varepsilon_{\text{t}}(k) \ = \widetilde{y}(k) - y(k). \tag{15}$$

This error is potentially amplified by the architecture.

Using (15), let us rewrite $\widetilde{y}(k-i)$ in the right-hand side of (14):

$$\begin{aligned}
\varepsilon_{\text{r}}(k) \quad &= \quad \widetilde{y}(k) - \sum_{i=0}^{n_b} b_i u(k-i) + \sum_{i=1}^{n_a} a_i y(k-i) \\
&\qquad + \sum_{i=1}^{n_a} a_i \varepsilon_{\text{t}}(k-i) \\
&= \quad \widetilde{y}(k) - y(k) + \sum_{i=1}^{n_a} a_i \varepsilon_{\text{t}}(k-i) \quad \text{(using (2))} \\
&= \quad \varepsilon_{\text{t}}(k) + \sum_{i=1}^{n_a} a_i \varepsilon_{\text{t}}(k-i) \qquad \text{(using (15))}.
\end{aligned} \tag{16}$$

If we rewrite equation (16) as

$$\varepsilon_{\text{t}}(k) \quad = \quad \varepsilon_{\text{r}}(k) - \sum_{i=1}^{n_a} a_i \varepsilon_{\text{t}}(k-i) \tag{17}$$

we obtain the equation of an IIR filter inputting $\varepsilon_{\text{r}}(k)$ and outputting $\varepsilon_{\text{t}}(k)$, whose transfer function is

$$\mathcal{H}_{\varepsilon}(z) = \frac{1}{1 + \sum_{i=1}^{n_a} a_i z^{-i}} \ . \tag{18}$$

Figure 5 illustrates this relationship between the ideal output y, the implemented output $\widetilde{y}_{\text{out}}$ and the different error terms.

We can now apply the Worst-Case Peak-Gain theorem to $\mathcal{H}_{\varepsilon}$ with input ε_{r} in order to bound ε_{t} by

$$\overline{\varepsilon}_{\text{t}} = \langle\!\langle \mathcal{H}_{\varepsilon} \rangle\!\rangle \, \overline{\varepsilon}_{\text{r}} \ . \tag{19}$$

Therefore, we can also keep $\overline{\varepsilon}_{\text{t}}$ as low as needed by increasing the internal precision ℓ_{ext} to reduce $\overline{\varepsilon}_{\text{r}}$.

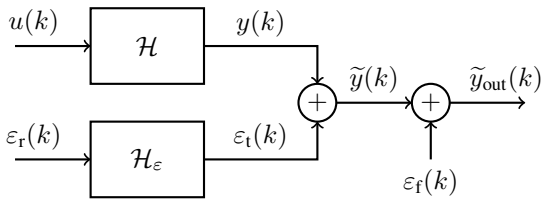

Fig. 5. A signal view of the error propagation with respect to the ideal filter

D. Putting it all together

Using above considerations, we can put all errors together and rewrite (12) as

$$\begin{aligned}\varepsilon_{\text{out}}(k) &= \widetilde{y}_{\text{out}}(k) - \widetilde{y}(k) + \widetilde{y}(k) - y(k) \\ &= \varepsilon_{\text{f}}(k) + \varepsilon_{\text{t}}(k). \end{aligned} \tag{20}$$

Hence,

$$\overline{\varepsilon}_{\text{out}} = \overline{\varepsilon}_{\text{f}} + \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle \, \overline{\varepsilon}_{\text{r}} \tag{21}$$

The objective of the last-bit accuracy of the architecture translates into the constraint $\overline{\varepsilon}_{\text{out}} < 2^{\ell_{\text{out}}}$. Taking into account the final rounding (which implies the error $\overline{\varepsilon}_{\text{f}} = 2^{\ell_{\text{out}}-1}$), we obtain the constraint on the error $\overline{\varepsilon}_{\text{r}}$ of the SOPCs that is required to satisfy the last-bit accuracy of the overall architecture:

$$\overline{\varepsilon}_{\text{r}} < \frac{2^{\ell_{\text{out}}-1}}{\langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle} \quad . \tag{22}$$

This constraint finally translates to the LSB ℓ_{ext} of the intermediate result as follows. As we stated in the Section III-B, we assume that we may build an SOPC last-bit accurate to any value of ℓ_{ext}: for this SOPC we will have $\overline{\varepsilon}_{\text{r}} < 2^{\ell_{\text{ext}}}$.

Using (22), we obtain that the optimal value of ℓ_{ext} that ensures this constraint is

$$\ell_{\text{ext}} = \ell_{\text{out}} - 1 - \lceil \log_2 \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle \rceil \quad . \tag{23}$$

The implementation of this error analysis actually uses a guaranteed overestimation of $\langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle$ [3]. This ensures that rounding errors in the the computation of $\langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle$ itself do not jeopardize the accuracy. Because of this overestimation, very rarely, we might compute on one bit more than it was necessary by (23). This rare one-bit overestimation of the datapath size has no impact in practice.

IV. Sum of products computing just right

A. Problem statement

In this section, we address the sub-problem of building a last-bit accurate Sum of Product by Constants (SOPC), *i.e.* an architecture computing

$$r = \sum_{i=1}^{N} c_i x_i \tag{24}$$

accurate to 2^{ℓ_r}, for a set of real constants c_i, and a set of fixed-point inputs x_i.

In previous work [13], all the x_i shared the same format, as is the case in an FIR filter. In the context of an IIR filter, this is no longer true: on Figure 4, we have a single SOPC where the c_i may be a_i or b_i, and the x_i may be either some delayed u_i, or some delayed y_i. The format of the y_i, as determined by previous section, is in general different from that of the u_i. Therefore, the present work uses a more generic interface to the SOPC generator, where the format of each input may be specified independently. This interface is shown on Figure 6. The input LSBs are provided as ℓ_i. For the input MSBs, instead of m_i, the interface uses the maximum absolute value $\overline{x_i}$ of each x_i, which provides a finer information that will be exploited in the sequel. In the context of an IIR filter, the output precision will be $\ell_r = \ell_{\text{ext}}$, this value being defined by the error analysis of previous sections.

Another difference with [13] is that the output MSB m_r is input to the generator. An overestimation of m_r could be computed out of the c_i and the input formats, as in [13]. However, the worst case peak gain of an IIR filter provides a finer value of m_r, and in this case we want to provide this value to the SOPC generator.

Here again, the weight ℓ_r of the least significant bit of the SOPC output also specifies the accuracy of this SOPC: the present section shows how to build an SOPC accurate to 2^{ℓ_r}. This is what was assumed in the previous section with $\ell_r = \ell_{\text{ext}}$.

B. Error analysis for a last-bit accurate SOPC

The fixed-point summation of the various terms $c_i x_i$ is depicted on Fig. 7. For this figure, we take as an example the 4-input SOPC of an IIR of order 2 with arbitrary coefficients: it is a smaller version of the one depicted on Figure 4, where x_0 and x_1 are respectively $u(k)$ and $u(k-1)$, while x_2 and x_3 are respectively $\widetilde{y}(k-1)$ and $\widetilde{y}(k-2)$. The output r will become $\widetilde{y}(k)$.

As shown on the figure, a real c_i may have an infinite number of bits. Therefore, even though the x_i are finite, each product $c_i x_i$ potentially also has an infinite number of bits.

The MSB of each product $c_i x_i$ is easily determined out of the value of c_i itself and $\overline{x_i}$: $|x_i| \leq \overline{x_i}$, therefore $|c_i x_i| \leq c_i \overline{x_i}$, so the MSB of $c_i x_i$ will be $\lceil \log_2(|c_i \overline{x_i}|) \rceil$. Here, using $\overline{x_i}$ instead of an MSB specification for x_i can save one bit. As previously, to anticipate possible overflows due to rounding, the implementation must add, before taking the $\log_2$, an upper bound of its rounding error. This bound will be detailed in the sequel.

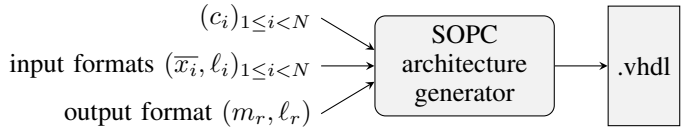

Fig. 6. Interface to a sum-of-product-by-constant generator

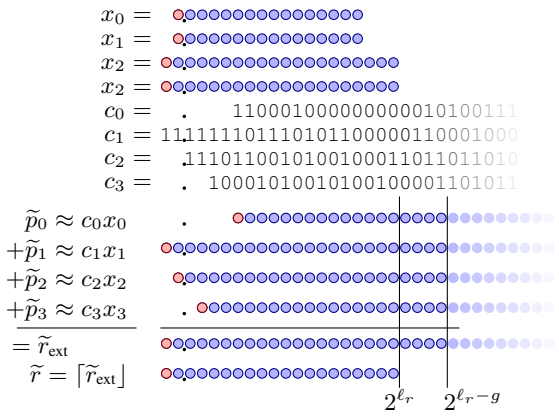

Fig. 7. Alignment of the $c_i x_i$ for fixed-point x_i and real c_i

Negative $c_i x_i$ must have have their sign extended to the MSB of the sum, so it could seem that Figure 7 only shows the cases when all the $c_i x_i$ are positive. Here we must explain another technicality. The sign extension `ss...ssxxxxxxxx` of a signed number `sxxxx`, where `s` is the sign bit, may be performed as follows [14]:

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

Here $\overline{\texttt{s}}$ is the boolean complement of `s`. The reader may check this equation in the two cases, $\texttt{s} = 0$ and $\texttt{s} = 1$. Now the variable part $\overline{\texttt{s}}\texttt{xxxxxxx}$ has the same MSB as in the positive case, and this is what Figure 7 shows.

This transformation is not for free: we need to add the constant `11...110000000`. Fortunately, in the context of a summation, we may add in advance all these constants together. Thus the overhead cost of two's complement in a summation is limited to the addition of one single constant. In the following, we will use another trick that allows to merge this addition for free in the computations of one of the $c_i x_i$.

Performing all the internal computations to the output precision ℓ_r would in general not allow the last-bit accuracy to precision ℓ_r, due to the accumulation of rounding errors. The solution is, as previously, to use a slightly extended precision $\ell_r - g$ for the internal computation: g is a number of "guard" bits. As this extended precision will require more hardware, we now discuss how to compute the extended precision that will minimize this hardware overhead.

We assume that we are able to build hardware constant multipliers that compute some approximation

$$\widetilde{p}_i = c_i x_i + \varepsilon_i(g) \tag{25}$$

of the mathematical product $c_i x_i$ such that the LSB of each $\widetilde{p}_i$ is $\ell_r - g$ (see Figure 7), and we assume that the rounding error ε_i of each of these multipliers is bounded by some $\overline{\varepsilon}_i(g)$:

$$|\varepsilon_i(g)| < \overline{\varepsilon}_i(g) \quad . \tag{26}$$

The value of $\overline{\varepsilon}_i(g)$ depends on the constant: multiplication by zero will be exact, as will be, under some conditions, multiplications by powers of two and by other constants that can be written in binary on few bits. In the general case where c_i is real, the multiplier will entail a rounding error which depends on the multiplier technique used (a detailed example will be shown in the sequel). However, whatever the technique, this error bound can be made as small as needed by increasing g (in other words, by computing more accurately).

The output value $\widetilde{r}$ is computed in an architecture as the sum of the $\widetilde{p}_i$. This summation, as soon as it is performed with adders of the proper size, will entail no error (Figure 7). Indeed, fixed-point addition of numbers of the same format may entail overflows (these have been taken care of), but no rounding error. This enables us to write

$$\widetilde{r}_{\text{ext}} = \sum_{i=0}^{N-1} \widetilde{p}_i, \tag{27}$$

therefore the total rounding error of the sum of product is defined as

$$\varepsilon_{\text{SOPC}} = \sum_{i=0}^{N-1} \widetilde{p}_i - \sum_{i=0}^{N-1} c_i x_i = \sum_{i=0}^{N-1} \varepsilon_i(g) \tag{28}$$

and thanks to (26) can be bounded as follows:

$$\overline{\varepsilon}_{\text{SOPC}} < \sum_{i=0}^{N-1} \overline{\varepsilon}_i(g) \quad . \tag{29}$$

As each $\overline{\varepsilon}_i(g)$ can be made arbitrarily small by increasing g, there exists some g such that

$$\sum_{i=0}^{N-1} \overline{\varepsilon}_i(g) < 2^{\ell_r - 1} \quad . \tag{30}$$

The intermediate result now has g more bits at its LSB than we need (Figure 7). It therefore needs itself to be rounded to the target format. This is easy, using the identity $\circ(x) = \lfloor x + \frac{1}{2} \rfloor$: rounding to precision $2^{-\ell_r}$ is obtained by first adding $2^{\ell_r - 1}$ (this is a single bit) then discarding bits lower than $2^{-\ell_r}$. However, in the worst case, this will entail an error $\varepsilon_{\text{final rounding}}$ of at most $2^{\ell_r - 1}$.

To sum up, the overall error of a last-bit accurate SOPC architecture is

$$\widetilde{r} - \sum_{i=0}^{N-1} c_i x_i = \varepsilon_{\text{final rounding}} + \varepsilon_{\text{SOPC}} \tag{31}$$

$$< 2^{\ell_r - 1} + 2^{\ell_r - 1} = 2^{\ell_r} \quad . \tag{32}$$

All the previous is quite independent of the target technology. However, the actual computation of the optimal g out of constraint (30) will depend on the multiplier technique chosen. This is the reason why we do not give a generic formula providing g.

However, for illustration and completeteness, the remainder of this section focusses on a particular technology: LUT-based SOPC architectures for FPGAs. It explores architectural means to reach last-bit accuracy at the smallest possible cost on this technology.

C. Perfectly rounded constant multipliers

On most FPGAs, the basic logic element is the look-up-table (LUT), a small memory addressed by α bits. For the current generation of FPGAs, $\alpha = 6$.

As we have a finite number of possible values for x_i, it is possible to build a perfectly rounded multiplier by simply tabulating all the possible products. The precomputation of table values must be performed with large enough accuracy (using multiple-precision software) to ensure the correct rounding of each entry. This even makes perfect sense for small input precisions on recent FPGAs: if x_i is a 6-bit number, each output bit of the perfectly rounded product $c_i x_i$ will cost exactly one 6-input LUTs. For 8-bit inputs, each bit costs only 4 LUTs. In general, for $(6+k)$-bit inputs, each output bit costs 2^k 6-LUTs: this approach scales poorly to larger inputs.

Perfect rounding to precision $\ell_r + g$ means a maximum error smaller than an half-LSB: $\bar{\varepsilon}_i = 2^{\ell_r - g - 1}$. Note that for real-valued c_i, this is more accurate than rounding the result of a multiplier inputting $\circ_{\ell_r}(c_i)$: the latter would accumulate two successive rounding errors.

D. Table-based constant multipliers for FPGAs

For larger precisions, we may use a variation of the KCM technique, due to Chapman [5] and further studied by Wirthlin [6]. The original KCM method addresses the multiplication by an integer constant. We here present a variation called FixRealKCM that performs the multiplication by a *real* constant.

This method consists in breaking down the binary decomposition of an input x_i into D chunks d_{ik} of α bits. With the input size being $m_i - \ell_i + 1$, we have

$$D = \lceil (m_i - \ell_i + 1)/\alpha \rceil \tag{33}$$

(for instance $D = 3$ on Figure 8). Mathematically, this is written

$$x_i \;=\; \sum_{k=1}^{D} 2^{-k\alpha} d_{ik} \quad \text{where} \quad d_{ik} \in \{0, ..., 2^\alpha - 1\} \quad . \tag{34}$$

Another point of view is that the input x_i is considered as a radix-2^α number, the d_{ik} being its digits. For instance with $\alpha = 4$ we obtain the classical hexadecimal writing of x_i.

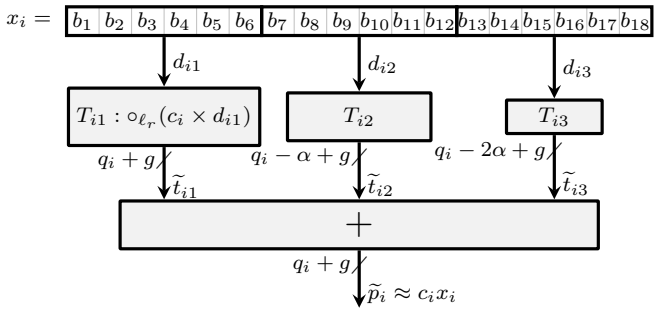

Fig. 8. The FixRealKCM method when x_i is split in 3 chunks

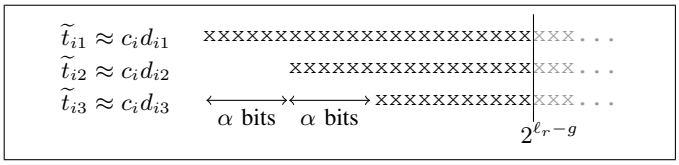

Fig. 9. Aligment of the terms in the KCM method

The product becomes

$$c_i x_i \;=\; \sum_{k=1}^{D} 2^{-k\alpha} c_i d_{ik} \quad . \tag{35}$$

Since each chunk d_{ik} consists of α bits, where α is the LUT input size, we may tabulate each product $c_i d_{ik}$ in a look-up table that will consume exactly one α-bit LUT per output bit. This is depicted on Figure 8. Of course, $c_i d_{ik}$ has an infinite number of bits in the general case: as previously, we will round it to precision $2^{-\ell_r - g}$. In all the following, we define $\widetilde{t}_{ik} = \circ_{\ell_r - g}(c_i d_{ik})$ this rounded value (see Figure 9).

Contrary to classical (integer) KCM, all the tables do not consume the same amount of resources. The factor $2^{-k\alpha}$ in (35) shifts the MSB of the table output $\widetilde{t}_{ik}$, as illustrated by Figure 9.

Here also, the fixed-point addition is errorless. The error of such a multiplier therefore is the sum of the errors of the D tables, each perfectly rounded:

$$\varepsilon_i < D \times 2^{\ell_r - g - 1} \quad . \tag{36}$$

This error is proportional to 2^{-g}, so can made as small as needed by increasing g.

E. Computing the sum

Instead of considering each KCM in isolation, it is better to consider the summation at the SOPC level. Indeed, our SOPC result is now obtained by computing a double sum:

$$\widetilde{y} = \circ_p \left(\sum_{i=0}^{N-1} \sum_{k=1}^{D} 2^{-k\alpha} \widetilde{t}_{ik} \right) \tag{37}$$

There, the errors of each $\widetilde{t}_{ik}$ add up into an overall SOPC error, out of which the value of g can be computed.

Before that, let us also observe that it is often possible to use a finer bound than (36). Indeed, some constant multipliers entail no error: it is for instance the case for multiplication by 0 and by 1. Such trivial cases will happen quite often if the proposed SOPC generator is used as a backend for a larger architecture generator, as is the case in the present article. Besides, such trivial cases deserve specific treatment since their implementation is much simpler than the generic case.

Therefore, the implementation first invokes, for each constant, a method that returns the maximum error that will be entailed by a multiplier by this constant. This error is expressed in units in the last place (ulp), whatever the value of g will be. The implementation sums these errors, then uses this sum to compute the value of g that will enable last-bit accuracy.

Once this g has been determined it may proceed with the actual construction of the multipliers.

Here is the list of cases currently managed by the implementation:

- if $c_i = 0$, then $\varepsilon_i = 0$.
- if $|c_i| = 1$ or more generally if $|c_i| = 2^k$, then $\varepsilon_i = 0$ if $k + l_i \geq \ell_r$ (shift of x_i such that all the bits will be kept), otherwise $\varepsilon_i = 1$ (shift to the right, losing some bits due to truncation). Here we may overestimate the error, because the test should be if $k + l_i \geq \ell_r - g$, but we don't know g yet.
- In the general case when we use the generic KCM architecture, $\varepsilon_i = D/2$ (we have D tables, each entailing one half-ulp of error).

One final technicality: we have so far assumed that the number of tables D is computed out of the input size, using (33). However, for small constants, it may happen that the contribution of the lower tables can be neglected. To understand this, consider Figure 9: each table output is shifted right if c_i is small. Therefore, the implementation will not generate a table if its MSB is smaller than $\ell_r - g - 1$. The error analysis remains valid in this case, although the source of the error is no longer the rounding of the table, but its being neglected altogether. If more than one table is fully neglected, this error analysis was slightly pessimistic (we could have a single half-ulp for all the neglected tables), but it remains safe.

F. Computing the sum

In FPGAs, each bit of an adder also consumes one LUT. Therefore, in a KCM architecture, the LUT cost of the summation is expected to be roughly proportional to that of the tables. However, using the associativity of exact fixed-point addition, this summation can be implemented very efficiently using compression techniques developed for multipliers [14] and more recently applied to sums of products [15], [16]. In this work, we may use the *bit heap* framework introduced in [8]. Each table throws its $\widetilde{t}_{ik}$ to a bit heap that is in charge of performing the final summation. The bit heap framework is naturally suited to adding terms with various MSBs, as is the case here. It also manages two's complement numbers efficiently – the interested reader is referred to [8] for details.

Figures 10 and 11 show examples of bit heaps obtained by the proposed method. On these figures (which are generated by the tool), we have binary weights on the horizontal axis, and the various terms to add on the vertical axis. Roughly speaking, the height of a bit heap is proportional to the number of non-zero coefficients. The width shows the needed internal precision computed by the tool to ensure both filter stability and last-bit accuracy: on the figures, the vertical lines mark the bit weights ℓ_{out} and ℓ_{ext}: they illustrate the $1 + \lceil \log_2 \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle \rceil$ extra bits needed to manage the error amplification on the feedback loop (see equation (23)), as well as the g extra bits needed to absorb the rounding errors in each KCM table.

The current FixRealKCM implementation properly manages trivial constants such as zero and powers of two. A multiplication by 0 will add nothing to the bit heap, while a multiplication by a power of two will just add to the bit heap

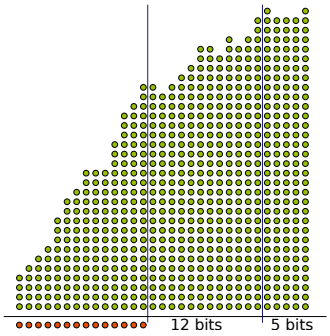

Fig. 10. The bit heap of a narrowband 12-bit filter (example 2 of section V with $\omega_p = 0.5$). In this case the tool has computed $-\log_2 \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle = 12$ and $g = 5$.

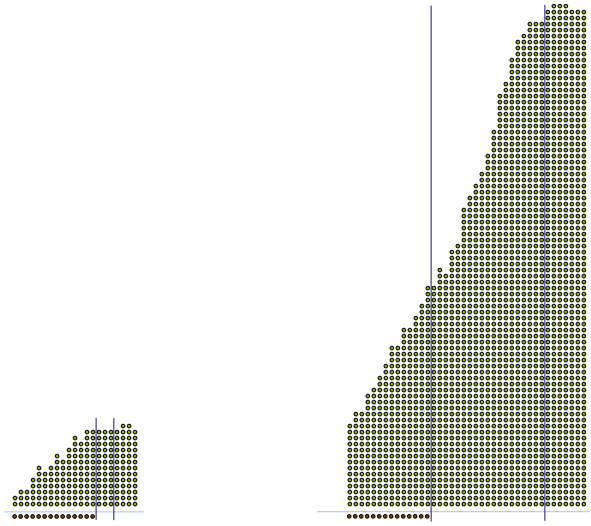

Fig. 11. The bit heaps of two 12-bit Butterworth filters (the extreme cases of example 1 of section V). On the left, order 4, $-\log_2 \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle = 3$ and $g = 4$. On the right, order 20,$-\log_2 \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle = 19$ and $g = 7$

the (properly shifted and possibly truncated) input. The next step will be to similarly optimize constants that fit on a few bits, such as those produced by quantization tools. This can significantly change the bit heap shapes.

The previous figures only illustrate the initial bit heap. FloPoCo also generates an architecture that computes the sum of all these bits. The current version still uses the greedy heuristic of [8]. The state of the art is to compute an optimal architecture using Integer Linear Programming techniques [8], and will be used soon.

G. Final rounding by truncation

There is one more term to add to the summation of (37): the rounding bit $2^{\ell_r - 1}$, necessary for the final rounding by truncation. Its value is added to one of the tables.

Finally, the typical architecture generated by our tool is depicted by Figure 12.

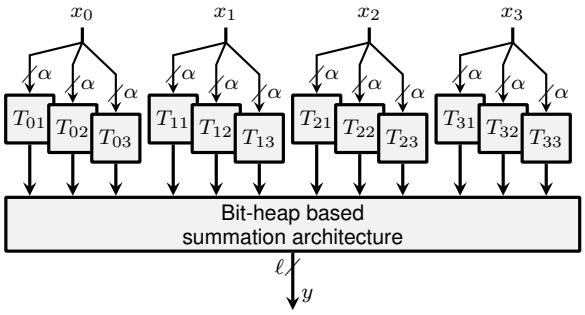

Fig. 12. KCM-based SOPC architecture for $N = 4$, each input being split into 3 chunks

V. Implementation and results

The method described in this paper is implemented as the `FixIIR` operator of FloPoCo. `FixIIR` offers the interface shown on Fig. 1, and inputs the coefficients a_i and b_i as arbitrary-precision numbers or mathematical expressions. `FixIIR`, like most FloPoCo operators, was designed with a testbench generator [17]. Operators reported here have been checked for last-bit accuracy by extensive simulation.

The results below were obtained after place and route for Kintex-7 (7k70tfbv484-3) using Vivado 2016.4. The reported timings don't include the IBUF/OBUF delay.

Various possible use-cases are possible for the proposed tool. For instance, one can easily explore a large quantity of filter implementation settings. We demonstrate our tool on two different families of bandpass digital filters. These experiments all target implementations with 12 fractional bits on the output.

Example 1: fixed passband, moving stopbands. We consider implementation of 23 bandpass filters that have the following specifications w.r.t. the normalized Nyquist frequencies:

- passband $[0.45, 0.55]$ with the maximum passband ripple 1dB
- stopbands $[0, 0.43 - 0.01k]$ and $[0.57 + 0.01k, 1]$ with the parameter $k = 0, 1, \ldots, 22$; minimum attenuation is 20dB for each band.

Each filter in this family is a Butterworth filter designed with Matlab. Figure 13 shows three different transfer functions from this family.

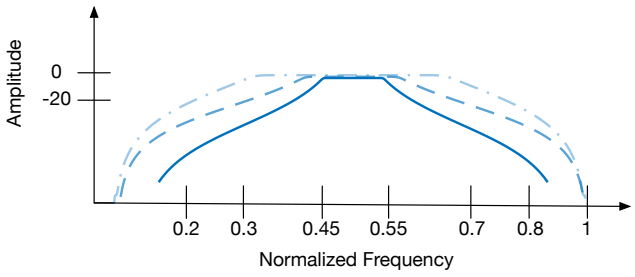

Fig. 13. Example 1, family of filters with fixed passband and increasing transition band. Here, three different transfer functions of this family.

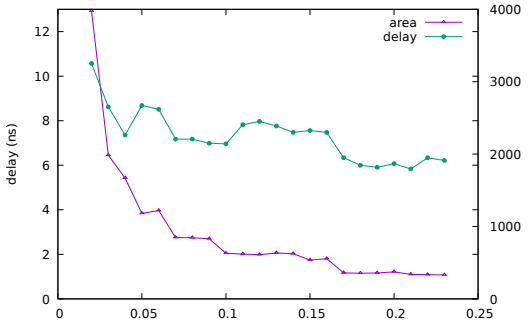

Fig. 14. Example 1, area and delay of implementations as a function of the width of transition band.

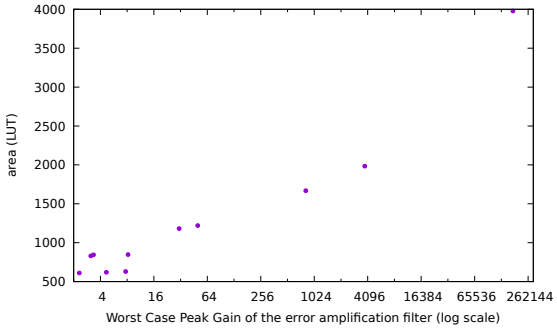

Fig. 15. Example 1, dependency of area on $\langle\langle \mathcal{H}_\varepsilon \rangle\rangle$.

Fig. 14 gives the area (in FPGA LUTs) and delay of obtained implementations as functions of the width of the transition band (which varies between 0.02 and 0.24).

On this example we can observe how the hardware cost of the filters decreases with the increase of the width of transition bands. The first filter in this family (i.e. $k = 0$) has an extremely narrow transition band, and the highest hardware cost. To ensure that the output of this filter is last-bit accurate to 12 fractional bits, the tool added $g = 7$ guard bits for the evaluation of the SOPC, then 19 more bits to absorb the amplification of errors in the feedback loop. Fig. 11 (right hand side) shows the bit heap that corresponds to this filter.

Classically, increasing the transition band simplifies the implementation: the filter's orders decrease and the rounding errors have less impact on the output, i.e. $\langle\langle \mathcal{H}_\varepsilon \rangle\rangle$ decreases. The proposed tool allows a designer to quantify this effortlessly and precisely.

A useful rule of thumb, illustrated by Fig. 15, is that the area is roughly linear in the number of bits of the internal format.

Example 2: sliding passband. In this setting we consider implementations of bandpass filters that have a very narrow passband and narrow transition band. Such filters are frequently used in Software Defined Radio [18], biomedical circuit design [19], digital television, etc. Classically, implementation of such bandpass filters is more complicated when the passband is near the the bounds of the frequency

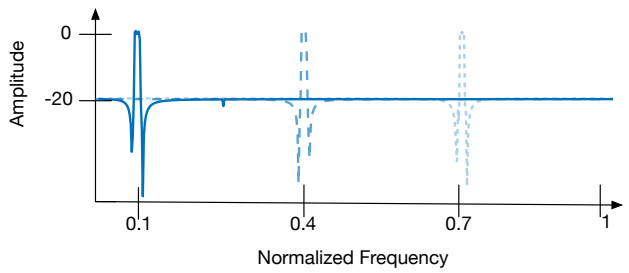

Fig. 16. Example 2, family of filters with fixed passband and transition bands with sliding parameter ω_p. Here three transfer functions of this family.

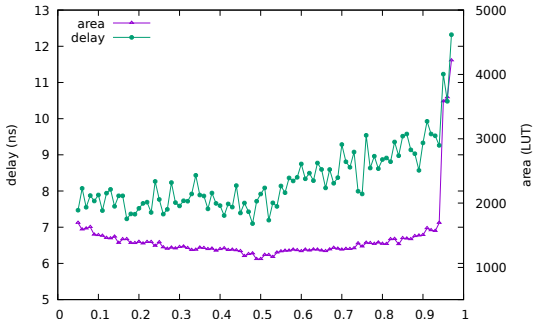

Fig. 17. Example 2, area and delay of implementations as a function of the parameter ω_p.

domain [1]. To illustrate how the proposed tool may quantify this effect, we fix the widths of the passband and the transitions bands to 0.01 and "slide" the passband through the normalized frequency interval $[0, 1]$. Formally, we consider the following family of filters:

- passband $[\omega_p, \omega_p + 0.01]$ with passband ripple 1dB
- stopbands in $[0, \omega_p - 0.01]$ and $[\omega_p + 0.02, 1]$ with minimum attenuation 20dB

where $\omega_p = 0.05 + 0.01k$, $k = 0, 1, \ldots, 94$. Each filter in this family is an elliptic filter designed with Python Scipy. All filters are of the same order 4, except for $k \in [92, 94]$ where the order increased to 6.

Fig. 17 gives the area and delay with respect to the position of $\omega_p \in [0.05, 0.97]$. It can be clearly observed that narrow bandpass filters have smaller complexity when the normalized passband frequency ω_p approaches 0.5. The bit heap corresponding to the implementation with $\omega_p = 0.5$ is given on Fig.10.

Interestingly, the last three filters ($k \in [92, 94]$) are considerably more sensitive to rounding errors (i.e. $\langle\langle \mathcal{H}_\varepsilon \rangle\rangle$ increases for them), thus require more area to ensure the output accurate to 12 fractional bits.

Such phenomena for the narrow-band bandpass filters is well known. However, in general case, overcoming difficulties in the implementation of the recursive filters is usually not straightforward and might require specific knowledge from the designer. The proposed approach automatically provides accurate implementations of even the most sensitive filters at the minimal cost, and the designer is informed of the cost of this accuracy.

VI. Conclusion and perspectives

This article advocates a very simple specification for low-precision architectures of digital filters: whatever the inputs, the difference between the results computed by the architecture and the result computed by an infinite precision machine should be less than the weight of the least significant bit of the output. This specification brings safety to designers who can trust that the architecture behaves as, e.g., a double-precision Matlab simulation. It also defines a universal rule of the game by which different architectures can be fairly compared.

This work demonstrates this approach through an end-to-end open-source tool that generates the VHDL code of a Direct Form I implementation from its mathematical coefficients. It explores architectural choices in the family of Direct Form I filters, and selects the architecture that, while guaranteeing the respect of the above specification, has minimal-size internal formats.

An important observation is that even for simple, low-order filters, the intermediate format needs to be significantly larger than the input/output format: in our experiments, we never need less than 4 extra bits on the internal format to achieve the specification. For higher-order or unstable filters, several tens of extra bits may be needed, even if the output format is only 8, 12 or 16 bits. This overhead depends on the worst-case peak gain of the error filter: it is mostly independant from the output format.

This work, however, does not claim to close the subject of digital filter architecture generation.

Future work first includes several technical improvements to the current implementation, such as the exploitation of symmetries in the coefficients, or a better bit heap compression algorithm.

Beyond that, this work is a solid foundation on which to build future research. Here are some directions.

In this work we have assumed real coefficients. We may now address the issue of *coefficient quantization* from a new point of view. If a designer manages to quantize coefficients on very few bits and still obtain a stable filter with acceptable transfer function, the proposed technique will work without change: quantized coefficients are also real coefficients. Therefore, the proposed approach enables a clear decoupling of the issue of coefficient quantization from the issue of intermediate rounding in the architecture.

However, quantized coefficients will also enable further optimizations. Indeed, the product of a quantized coefficient by an input (or by a subword of the input in the case of KCM multipliers) will have a finite number of bits. This potentially improves both error analysis and architecture generation: if these bits are all within the internal bit range determined by the tool, the product becomes exact, while the corresponding table output becomes smaller than in the case of a real coefficient, also leading to smaller bit heaps. In addition, with quantized coefficients, it also becomes relevant to compare with shift-and-add implementations of constant multipliers [20].

The next step is to explore filter structures that are more interesting than the DFI, for instance the decomposition into second-order sections [1]. In this structure a filter is decomposed into a cascade of biquad filters, each of which can be implemented with a DFI algorithm. An advantage of such a decomposition is smaller sensitivity towards rounding errors. Quantitatively, each section of the cascaded system may be analyzed independently using the worst-case peak gain and require different number of additional bits ℓ_{ext}. However, it is not completely straightforward how to choose the rounding strategy: on the one hand, rounding the output of each section to some $\tilde{y}_{\text{out}_i}$ increases the overall output error; on the other hand, direct propagation of $\tilde{y}(k)$ without rounding will increase the size of subsequent SOPCs. A trade-off might be achieved by a clever choice of section ordering [21].

It would be also interesting to consider other filter structures, such as Direct Form II (transposed or not), state-spaces, cascade and/or parallel decomposition, ρ-operator based structures [2], Lattice Wave Digital filters [22], etc. These algorithms are less sensitive to finite precision effects [22] (coefficient quantization and roundoff errors) and, even if some of them require more computations than the DFI, the total area (LUT) to achieve *just right* computing may be reduced. This work will allow one to quantify this, although it will not be completely straightforward: these structures have many intermediate formats, whose sizes also have to be determined and minimized. A unifying filter representation called SIF [23] could be the key to generalize the approach of the present article.

REFERENCES

[1] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, 3rd ed. NJ, USA: Prentice Hall Press, 2009.

[2] G. Li and Z. Zhao, "On the generalized DFIIt structure and its state-space realization in digital filter implementation," *IEEE Trans. on Circuits and Systems*, vol. 51, no. 4, pp. 769–778, April 2004.

[3] A. Volkova, T. Hilaire, and C. Lauter, "Reliable evaluation of the worst-case peak gain matrix in multiple precision," in *2015 IEEE 22nd Symposium on Computer Arithmetic*, June 2015, pp. 96–103.

[4] ——, "Determining fixed-point formats for a digital filter implementation using the worst-case peak gain measure," in *2015 49th Asilomar Conference on Signals, Systems and Computers*, Nov 2015, pp. 737–741.

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

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

[7] F. de Dinechin, H. Takeugming, and J.-M. Tanguy, "A 128-tap complex FIR filter processing 20 giga-samples/s in a single FPGA," in *44th Asilomar Conference on Signals, Systems & Computers*, 2010.

[8] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes, and B. Popa, "Arithmetic core generation using bit heaps," in *Field-Programmable Logic and Applications*, Sep. 2013.

[9] M. Kumm and P. Zipf, "Pipelined compressor tree optimization using integer linear programming," in *Field Programmable Logic and Applications (FPL)*, Sept 2014.

[10] IEEE Std 802.15.4-2006, *IEEE Standard for Information technology– Telecommunications and information exchange between systems– Local and metropolitan area networks– Specific requirements– Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)*, 2006.

[11] V. Balakrishnan and S. Boyd, "On computing the worst-case peak gain of linear systems," *Systems & Control Letters*, vol. 19, pp. 265–269, 1992.

[12] S. P. Boyd and J. Doyle, "Comparison of peak and RMS gains for discrete-time systems," *Syst. Control Lett.*, vol. 9, no. 1, pp. 1–6, June 1987.

[13] F. de Dinechin, M. Istoan, and A. Massouri, "Sum-of-product architectures computing just right," in *Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2014. [Online]. Available: http://hal.inria.fr/hal-00957609

[14] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2003.

[15] 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.

[16] R. Kumar, A. Mandal, and S. P. Khatri, "An efficient arithmetic sum-of-product (SOP) based multiplication approach for FIR filters and DFT," in *International Conference on Computer Design (ICCD)*. IEEE, Sep. 2012, pp. 195–200.

[17] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[18] L. Tsoeunyane, S. Winberg, and M. Inggs, "Software-defined radio FPGA cores: Building towards a domain-specific language," *Int. J. Reconfig. Comp.*, vol. 2017, pp. 3 925 961:1–3 925 961:28, 2017. [Online]. Available: https://doi.org/10.1155/2017/3925961

[19] K. Limnuson, H. Lu, H. J. Chiel, and P. Mohseni, "FPGA implementation of an IIR temporal filtering technique for real-time stimulus artifact rejection," in *2011 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, Nov 2011, pp. 49–52.

[20] K. Möller, M. Kumm, M. Kleinlein, and P. Zipf, "Pipelined reconfigurable multiplication with constants on FPGAs," in *Field Programmable Logic and Application (FPL)*. IEEE, 2014, pp. 1–6.

[21] M. Lankarany and H. Marvi, "Noise reduction in digital iir filters by finding optimum arrangement of second-order sections," in *2008 Canadian Conference on Electrical and Computer Engineering*, May 2008, pp. 689–692.

[22] A. Volkova and T. Hilaire, "Fixed-point implementation of lattice wave digital filter: comparison and error analysis," in *Proc. European Signal Processing Conference (EUSIPCO'15)*, September 2015.

[23] T. Hilaire, "Towards Tools and Methodology for the Fixed-Point Implementation of Linear Filters," in *Digital Signal Processing Workshop and IEEE Signal Processing Education Workshop (DSP/SPE), 2011 IEEE*, January 2011, pp. 488–493.