Hardware IIR Filters: Direct Form I Computing Just Right

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

Hardware IIR Filters: Direct Form I Computing Just Right

Florent de Dinechin*, Thibault Hilaire†, Matei Istoan*, Anastasia Volkova†
*Université de Lyon, INRIA,
INSA-Lyon, CITI-INRIA, F-69621, Villeurbanne, 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 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 the rounding errors, but also their infinite accumulation in infinite impulse response filters. This error analysis then guides the design of hardware satisfying the accuracy specification at the minimal hardware cost. This is illustrated on the case of low-precision LTI filters implemented in FPGA logic. This approach is fully automated in a generic, open-source architecture generator tool built upon the FloPoCo framework, and evaluated on a range of Finite and 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 as a transfer function in the frequency domain:

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

Equivalently, the output signal $y(k)$ and the input signal $u(k)$ may also be related by the following equation in the time domain:

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

Equation (1) or (2), along with a mathematical definition of each coefficient a_i and b_i, constitute the *mathematical specification* of the filter. In this specification, the coefficients are considered as real numbers. They may be given as explicit formulae, as for instance in textbook pulse-shaping filters (half-sine or root-raised cosine) used in wireless communication [1]. The coefficients may also be provided as high-precision floating-point numbers.

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

This article deals with the *implementation* of such a specification as fixed-point hardware operating on low-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 large filter orders (large values of n_a and n_b) or 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 wastes area, time and power by computing more accuracy than it can output.

The main contribution of this article is to show that such design decisions can be automated, based on the following simple claim:

An LTI architecture should be designed to compute results that are accurate to the last bit, but no more.

This claim is based on two common-sense observations. On the one hand, there is no point in designing an architecture that outputs bits which we know hold no useful information. On the other hand, there is no point in computing internally to an accuracy that we will not be able to express on the output.

Section III will show how this claim may be formalized

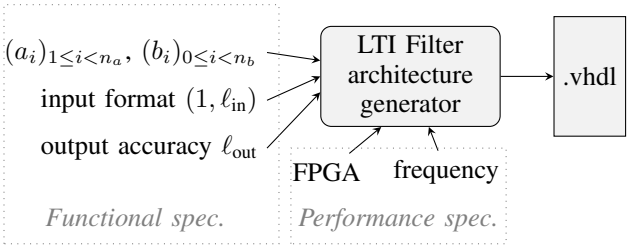

Fig. 1. Interface to the proposed tool. The coefficients a_i and b_i are considered as real numbers: they may be provided as high-precision numbers from e.g. Matlab, or even as mathematical formulae such as `sin(3*pi/8)`. The integers ℓ_{in} and ℓ_{out} respectively denote the bit position of the least significant bits of the input and of the result. In the proposed approach, ℓ_{out} specifies output precision, but also output accuracy.

and developed into a complete error analysis. This enables a very simple interface (Fig. 1) to an LTI implementation tool. The designer may focus on those design parameters which are relevant: the (real) coefficients, and the input/output formats. The construction of a minimal-cost architecture of proven last-bit accuracy can be fully automated out of this information.

As an illustration, an open-source tool demonstrates this implementation process for a particular hardware target: FPGAs based on Look-Up Tables (LUTs). Built upon the FloPoCo project[1], this tool automatically generates VHDL for LTI filters from the specification of Fig. 1. It also benefits from the FloPoCo back-end framework: the generated architectures are optimized for a user-specified FPGA family, and a user-specified frequency.

This demonstrator also incorporates several architectural novelties. The constant multipliers are built using an evolution of the KCM algorithm [2], [3] that manages multiplications by a real constant without needing to truncate it first [4]. The summation is efficiently performed thanks to the BitHeap framework recently introduced in FloPoCo [5]. 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 [1] (some of the examples illustrating this article address this standard). However, the same philosophy could be used to build other architecture generators, for instance exploiting embedded multipliers and DSP blocks.

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 number, 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 respectively denote the position of the most significant and least significant bit (MSB and LSB) of the data. Both m or ℓ can be negative if the format includes fractional bits. The value of bit at position i is always 2^i, except for the sign bit at position m, whose weight is -2^m (two's complement). The LSB position ℓ denotes the *precision* of the format. The MSB position denotes its *range*. Both MSB and LSB are included, therefore the size of a fixed-point number in (m, ℓ) is $m-\ell+1$. 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 $(m, \ell) = (0, -15)$. More precisely, the 2^{16} possible numbers in that format are in $[-1, 1 - 2^{-15}]$.

B. Approximations and errors

Due to the finite precision implementation, the exact filter $\mathcal{H}$, with exact output y, cannot usually be synthesized. An actual 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, in all 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.

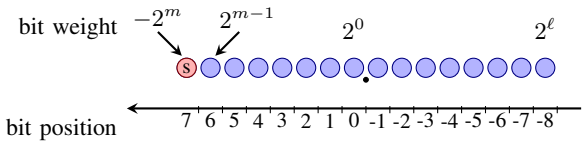

Fig. 2. The bits of a fixed-point format, here $(m, \ell) = (7, -8)$.

C. Perfect and faithful rounding

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 $|\circ_\ell(y(k)) - y(k)| < 2^{\ell-1}, \quad \forall k$. 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.

[1] http://flopoco.gforge.inria.fr/

In such approaches, one must provide ℓ_{out} and various other parameters that impact the accuracy, then measures the resulting accuracy, 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 measure is called the *Worst-Case Peak-Gain* (WCPG) [6], [7], and is defined as follows. If we consider a LTI filter $\mathcal{H}$ with input u (bounded by $\overline{u}$) and output y, then the WCPG of $\mathcal{H}$, denoted $\langle\!\langle \mathcal{H} \rangle\!\rangle$ is defined as the largest peak value of the output y over all possible input u with unitary peak value, i.e.

$$\langle\!\langle \mathcal{H} \rangle\!\rangle = \max_{||u||_\infty=1} ||y||_\infty \tag{4}$$

where $||u||_\infty$ is defined as $||u||_\infty = \max_k |u(k)|$.

Due to the linearity of the filter $\mathcal{H}$ and the property of the impulse response, the output $y(k)$ is a convolution between the impulse response and the input:

$$y(k) = \sum_{l=0}^{k} h(l)u(k-l) \tag{5}$$

So,

$$|y(k)| \leq \sum_{l=0}^{k} |h(l)|\overline{u} \tag{6}$$

with equality if $\forall 0 \leq l \leq k \quad u(k-l) = \text{sign}(h(l))\overline{u}$. Finally, the WCPG can be computed as the ℓ_1-norm of the impulse response h of $\mathcal{H}$, *i.e.*

$$\langle\!\langle \mathcal{H} \rangle\!\rangle = \sum_{k=0}^{\infty} |h(k)|. \tag{7}$$

Remark: the WCPG cannot be computed if the filter $\mathcal{H}$ is not Bounded Input Bounded Output (BIBO) stable [8], *i.e.* if the moduli of the poles of its transfer function are not all strictly smaller than 1 (otherwise, that makes its impulse response not absolutely summable).

The bound $\langle\!\langle \mathcal{H} \rangle\!\rangle\, \overline{u}$ on the output is quite conservative in practice, but it is always possible to exhibit an input $u(k)$ bounded by $\overline{u}$ such that the corresponding output is arbitrary close to its bound $\langle\!\langle \mathcal{H} \rangle\!\rangle\, \overline{u}$.

In this work, we compute WCPGs using the reliable algorithm exhibited in [9].

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 Figure 3.

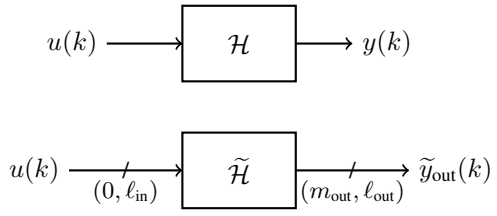

Fig. 3. The ideal filter (top) and its implementation (bottom)

Since the considered filters are linear, we can assume without loss of generality that the MSB of the input is equal to 0. The MSB of the output m_{out} is therefore defined by:

$$m_{\text{out}} = \lceil \log_2 \langle\!\langle \mathcal{H} \rangle\!\rangle \rceil \ . \tag{8}$$

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 $m_{\text{out}} = \left\lceil \log_2 \left(\langle\!\langle \mathcal{H} \rangle\!\rangle + 2^{\ell_{\text{out}}-1} \right) \right\rceil$. In addition, the implementation computes for (8) a slight but safe overestimation [10] of $\langle\!\langle \mathcal{H} \rangle\!\rangle$.

Then, instead of computing $y(k)$ with equation (2), we will compute an approximation $\widetilde{y}(k)$ of the involved Sum of Product by Constants (SOPC) using some internal format $(m_{\text{out}}, \ell_{\text{ext}})$

$$\widetilde{y}(k) \approx \sum_{i=0}^{n_b} b_i u(k-i) - \sum_{i=1}^{n_a} a_i \widetilde{y}(k-i) \tag{9}$$

and the final output $\widetilde{y}_{\text{out}}(k)$ will be some rounding of this intermediate value $\widetilde{y}(k)$. This computations scheme is summed up by the abstract architecture of Figure 4.

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

$$\varepsilon_{\text{out}}(k) \quad = \quad \widetilde{y}_{\text{out}}(k) - y(k) \tag{10}$$

Let us now 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

$$\varepsilon_{\text{f}}(k) = \widetilde{y}_{\text{out}}(k) - \widetilde{y}(k) \quad . \tag{11}$$

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 faithful rounding.

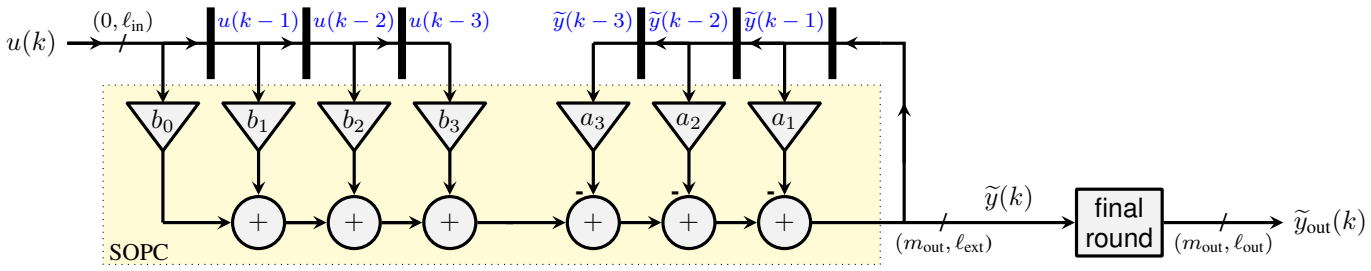

Fig. 4. Abstract architecture for the direct form realization of an LTI filter

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

$$\varepsilon_{\text{r}}(k) \quad = \quad \widetilde{y}(k) - \left(\sum_{i=0}^{n_b} b_i u(k-i) - \sum_{i=1}^{n_a} a_i \widetilde{y}(k-i) \right) \tag{12}$$

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 now define $\varepsilon_{\text{t}}(k)$ as the error of $\widetilde{y}(k)$ with respect to $y(k)$:

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

This error is potentially amplified by the architecture.

Using (13), let us rewrite $\widetilde{y}(k-i)$ in the right-hand side of (12):

$$\begin{aligned}
\varepsilon_{\text{r}}(k) \quad &= \quad \widetilde{y}(k) - \sum_{i=0}^{n_b} b_i u(k-i) + \sum_{i=1}^{n_a} a_i y(k-i) \\
&\qquad + \sum_{i=1}^{n_a} a_i \varepsilon_{\text{t}}(k-i) \\
&= \quad \widetilde{y}(k) - y(k) + \sum_{i=1}^{n_a} a_i \varepsilon_{\text{t}}(k-i) \quad \text{(using (2))} \\
&= \quad \varepsilon_{\text{t}}(k) + \sum_{i=1}^{n_a} a_i \varepsilon_{\text{t}}(k-i) \qquad \text{(using (13))}.
\end{aligned} \tag{14}$$

If we rewrite equation (14) as

$$\varepsilon_{\text{t}}(k) \quad = \quad \varepsilon_{\text{r}}(k) - \sum_{i=1}^{n_a} a_i \varepsilon_{\text{t}}(k-i) \tag{15}$$

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

$$\mathcal{H}_\varepsilon(z) = \frac{1}{1 + \sum_{i=1}^{n_a} a_i z^{-i}} \ . \tag{16}$$

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

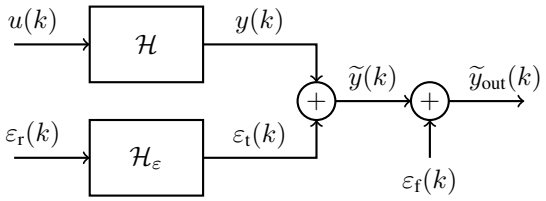

Fig. 5. A signal view of the error propagation with respect to the ideal filter

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

$$\overline{\varepsilon}_{\text{t}} \quad = \quad \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle \, \overline{\varepsilon}_{\text{r}} \ . \tag{17}$$

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

D. Putting it all together

We may now rewrite (10) as

$$\begin{aligned}
\varepsilon_{\text{out}}(k) \quad &= \quad \widetilde{y}_{\text{out}}(k) - \widetilde{y}(k) + \widetilde{y}(k) - y(k) \\
&= \quad \varepsilon_{\text{f}}(k) + \varepsilon_{\text{t}}(k)
\end{aligned} \tag{18}$$

hence

$$\begin{aligned}
\overline{\varepsilon}_{\text{out}} \quad &= \quad \overline{\varepsilon}_{\text{f}} + \overline{\varepsilon}_{\text{t}} \\
&= \quad \overline{\varepsilon}_{\text{f}} + \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle \, \overline{\varepsilon}_{\text{r}}
\end{aligned} \tag{19}$$

The objective of faithful rounding translates to the accuracy constraint $\overline{\varepsilon}_{\text{out}} < 2^{\ell_{\text{out}}}$. The final rounding implies an error bounded by $\overline{\varepsilon}_{\text{f}} = 2^{\ell_{\text{out}}-1}$ (round to nearest). To achieve faithful

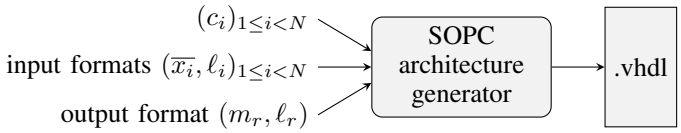

Fig. 6. Interface to a sum-of-product-by-constant generator

rounding, it therefore suffices that the error $\overline{\varepsilon}_{\mathrm{t}}$ of the filter before final rounding is bounded by $2^{\ell_{\mathrm{out}}-1}$. This leads to the following constraint on $\overline{\varepsilon}_{\mathrm{r}}$:

$$\overline{\varepsilon}_{\mathrm{r}} < \frac{2^{\ell_{\mathrm{out}}-1}}{\langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle} \quad . \tag{20}$$

This constraint finally translates to the LSB ℓ_{ext} of the intermediate result as follows. We assume that we may build an SOPC faithful to any value of ℓ_{ext}: for this SOPC we will have $\overline{\varepsilon}_{\mathrm{r}} < 2^{\ell_{\mathrm{ext}}}$.

Therefore, the constraint (20) holds if

$$2^{\ell_{\mathrm{ext}}} < \frac{2^{\ell_{\mathrm{out}}-1}}{\langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle} \quad . \tag{21}$$

and the optimal value of ℓ_{ext} that ensures this constraint is

$$\ell_{\mathrm{ext}} = \ell_{\mathrm{out}} - 1 - \lceil \log_2 \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle \rceil \quad . \tag{22}$$

The implementation of this error analysis actually uses a guaranteed overestimation of $\langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle$ [9]. 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, very rarely, the computed value of ℓ_{ext} may be one less than the mathematical value as defined per (22). This has no impact in practice.

Meanwhile, the MSB of the internal format is the same as that of the result (m_{out}). Some overflows may occur in the internal computation, but since the computation is performed modulo $2^{m_{\mathrm{out}}}$, the final result will be correct.

IV. Sum of products computing just right

A. Problem statement

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

$$r = \sum_{i=1}^{N} c_i x_i \tag{23}$$

for a set of real constants c_i, and a set of fixed-point inputs x_i.

In previous work [11], all the x_i shared the same format, as is the case in an FIR filter. In the context of an LTI 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.

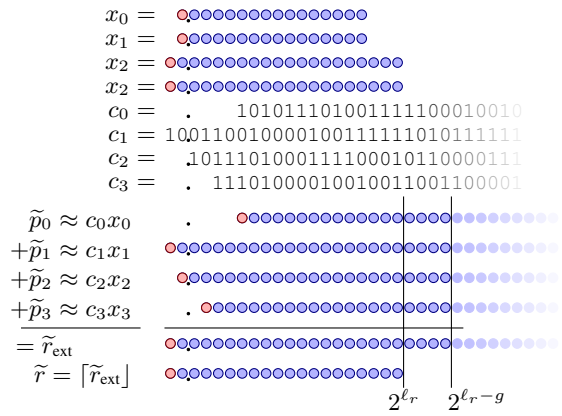

Fig. 7. Alignment of the $c_i x_i$ for fixed-point x_i and real c_i

Specifically, 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.

Another difference with [11] 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 [11]. However, the worst case peak gain of an IIR filter provides a much 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 $\ell_r = \ell_{\mathrm{ext}}$.

B. Error analysis for a faithfully rounded 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 $\overline{x_i}$: $|x_i| \le \overline{x_i}$, therefore $|c_i x_i| \le c_i \overline{x_i}$, so the MSB of $c_i x_i$ will be $\lceil \log_2(|c_i \overline{x_i}|) \rceil$. This is where 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.

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...ssxxxxxxx` of

a signed number sxxxx, where s is the sign bit, may be performed as follows [12]:

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

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 $\overline{\texttt{s}}$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 be accurate enough to achieve faithful rounding 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

$$\widetilde{p}_i = c_i x_i + \varepsilon_i \tag{24}$$

of the mathematical product $c_i x_i$. 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 $\overline{\varepsilon}_{\text{mult}}(g)$:

$$\varepsilon_i < \overline{\varepsilon}_{\text{mult}}(g) \quad . \tag{25}$$

The value of $\overline{\varepsilon}_{\text{mult}}(g)$ depends on the multiplier technique used (examples will be given in the sequel), but it can be made as small as needed by increasing g.

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. Indeed, fixed-point addition of numbers of the same format may entail overflows (these have been taken care of above), but no rounding error. This enables us to write

$$\widetilde{r}_{\text{ext}} = \sum_{i=0}^{N-1} \widetilde{p}_i, \tag{26}$$

therefore the total rounding error of the summation is defined as

$$\varepsilon_{\text{sum}} = \sum_{i=0}^{N-1} \widetilde{p}_i - \sum_{i=0}^{N-1} c_i x_i = \sum_{i=0}^{N-1} \varepsilon_i \quad . \tag{27}$$

As each ε_i is bounded by $\overline{\varepsilon}_{\text{mult}}(g)$, in the worst case the sum of the ε_i can come close to $N\overline{\varepsilon}_{\text{mult}}(g)$. However, this error term may be made as small as needed by increasing g: there exists some g such that $N\overline{\varepsilon}_{\text{mult}}(g) < 2^{\ell_r - 1}$.

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 faithful architecture SOPC is therefore

$$\widetilde{r} - \sum_{i=0}^{N-1} c_i x_i = \varepsilon_{\text{final rounding}} + \sum_{i=0}^{N-1} \varepsilon_i \tag{28}$$

$$< 2^{\ell_r - 1} + N\overline{\varepsilon}_{\text{mult}}(g) \tag{29}$$

and this error can be made smaller than $2^{-\ell_r}$ as soon as we are able to build multipliers such that

$$N\overline{\varepsilon}_{\text{mult}}(g) < 2^{\ell_r - 1} \quad . \tag{30}$$

All the previous was quite independent of the target technology: it could apply to ASIC synthesis as well as FPGA.

The remainder of this section, conversely, is more focused 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.

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, $\alpha = 6$.

C. Perfectly rounded constant multipliers

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 consume exactly one 6-input LUTs. For 8-bit inputs, each bit consumes only 4 LUTs. In general, for $(6 + k)$-bit inputs, each output bit consumes 2^k 6-LUTs: this approach scales poorly to larger inputs. However, perfect rounding to precision $\ell_r + g$ means a maximum error smaller than an half-LSB: $\overline{\varepsilon}_i = 2^{\ell_r - g - 1}$. Note that for real-valued c_i, this is more accurate than using a perfectly rounded multiplier that inputs $\circ_{\ell_r}(c_i)$: this 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 [2] and further studied by Wirthlin [3]. The original KCM method addresses the multiplication by an integer constant. We here present a variation 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

$$D = \lceil (m_i - \ell_i + 1)/\alpha \rceil \tag{31}$$

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

$$x_i = \sum_{k=1}^{D} 2^{-k\alpha} d_{ik} \quad \text{where} \quad d_{ik} \in \{0, ..., 2^\alpha - 1\} \quad . \quad (32)$$

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.

The product becomes

$$c_i x_i = \sum_{k=1}^{D} 2^{-k\alpha} c_i d_{ik} \quad . \quad (33)$$

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 bit 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 (33) 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:

$$\varepsilon_i < D \times 2^{\ell_r - g - 1} \quad . \quad (34)$$

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 faithful

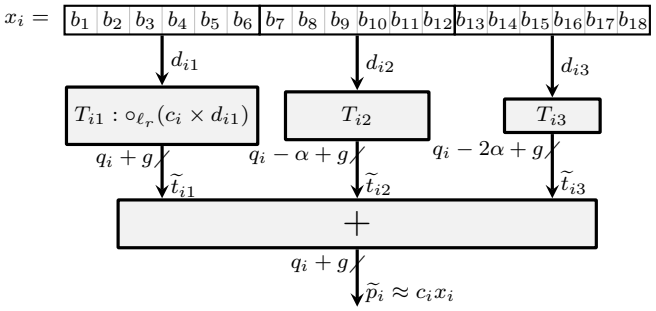

Fig. 8. The FixRealKCM method when x_i is split in 3 chunks

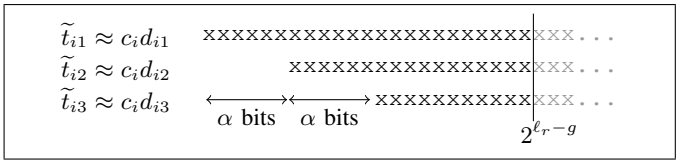

Fig. 9. Aligment of the terms in the KCM method

SOPC result is now obtained by computing a double sum:

$$\widetilde{y} = \circ_p \left(\sum_{i=0}^{N-1} \sum_{k=1}^{D} 2^{-k\alpha} \widetilde{t}_{ik} \right) \quad (35)$$

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 (34). 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 faithful rounding. Once this g has been determined it may proceed with the actual construction of the multipliers.

Here is the list of cases 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 (31). 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 fixed-point addition, this summation can be implemented very efficiently using compression techniques developed for multipliers [12] and more recently applied to sums of products [13], [14]. In this work, we may use the bit-heap framework introduced in [5]. Each table throws its $\widetilde{t}_{ik}$ to a bit-heap that is in charge

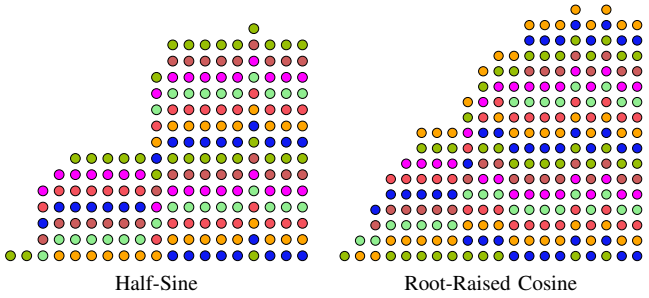

Fig. 10. Example bit heaps for simple FIR filters generated for Virtex-6

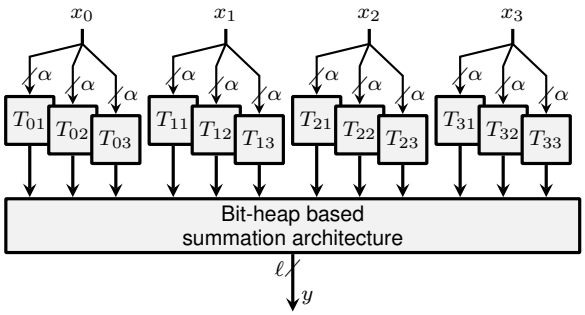

Fig. 11. KCM-based SOPC architecture for $N = 4$, each input being split into 3 chunks

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 [5] for details.

This is illustrated on Fig. 10. On these figures, we have binary weights on the horizontal axis, and the various terms to add on the vertical axis. These figure are generated by FloPoCo before bit heap compression.

The shape of the bit heap reflects the t_{ik}, which depend on the c_i. Some bit heaps are smaller due to special constant optimization.

G. Final rounding by truncation

There is one more term to add to the summation of (35): the rounding bit 2^{ℓ_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 11.

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 c_i as arbitrary-precision numbers.

To test `FixIIR`, a small Python script uses `numpy` and `scipy.signal` to generate the double-precision coefficients for Butterworth filters. Such wrappers can easily be written for other classical filter families.

f_c	guard bits	Speed	Area
0.6	7	175 MHz	170R + 1388L
0.7	8	168MHz	180R + 1621L
0.8	10	163MHz	190R + 1912L
0.9	15	169 MHz	215R + 2623L
0.95	19	159MHz	235R + 3141L

TABLE I. Needed guard bits and synthesis results for some 12-bit, 5th-order Butterworth filters.

Thanks to this script, several 5th-order Butterworth low-pass filters were generated for 12-bit signals, with increasingly values of the normalised cutoff frequency f_c. Table I shows how the number of guard bits computed by `FixIIR`, and the corresponding area and operating frequency.

`FixIIR`, like most FloPoCo operators, was designed with a testbench generator [15]. All these operators reported here have been checked for last-bit accuracy by extensive simulation.

These results were obtained for Virtex-6 (6vhx380tff1923-3) using ISE 14.7.

Conclusion

This paper claims that sum-of-product architectures should be last bit accurate, and demonstrates that this has two positive consequences: It gives a much clearer view on the trade-off between accuracy and performance, freeing the designer from several difficult choices. It actually leads to better solutions by enabling a "computing just right" philosophy. All this is demonstrated on an actual open-source tool that offers the highest-level interface.

Future work include several technical improvements to the current implementation, such as the exploitation of symmetries in the coefficients or optimization of the bit heap compression.

Beyond that, this work opens many perspectives.

As we have seen, fixed-point sum of products and sum of squares could be optimized for last-bit accuracy using the same approach.

We have only studied one small corner of the vast literature about filter architecture design. Many other successful approaches exist, in particular those based on multiple constant multiplication (MCM) using the transpose form (where the registers are on the output path) [16], [17], [18], [19], [20]. A technique called Distributed Arithmetic, which predates FPGA [21], can be considered a generalization of the KCM technique to the MCM problem. From the abstract of [22] that, among other things, compares these two approaches, "if the input word size is greater than approximately half the number of coefficients, the LUT based multiplication scheme needs less resources than the DA architecture and vice versa". Such a rule of thumb (which of course depends on the coefficients themselves) should be reassessed with architectures computing just right on each side. Most of this vast literature treats accuracy after the fact, as an issue orthogonal to architecture design.

A repository of FIR benchmarks exists, precisely for the purpose of comparing FIR implementations [23]. Unfortunately,

the coefficients there are already quantized, which prevents a meaningful comparison with our approach. Few of the publications they mention report accuracy results. However, cooperation with this group should be sought to improve on this.

We have only considered here the implementation of a filter once the c_i are given. Approximation algorithms, such as Parks-McClellan, that compute these coefficients, essentially work in the real domain. The question they answer is "what is the best filter with real coefficients that matches this specification". It is legitimate to wonder if asking the question: "what is the best filter with low-precision coefficients" could not lead to a better result.

Still in filter design, the approach presented here should be extended to infinite impulse response (IIR) filters. There, a simple worst-case analysis (as we did for SOPC) doesn't work due to the infinite accumulation of error terms. However, as soon as the filter is stable (i.e. its output doesn't diverge), it is possible to derive a bound on the accumulation of rounding errors [24]. This would be enough to design last-bit accurate IIR filters computing just right.

REFERENCES

[1] 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.

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

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

[4] 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.

[5] 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.

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

[7] 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.

[8] T. Kailath, *Linear Systems*. Prentice-Hall, 1980.

[9] A. Volkova, T. Hilaire, and C. Lauter, "Reliable evaluation of the Worst-Case Peak Gain matrix in multiple precision," in *IEEE Symposium on Computer Arithmetic*, 2015. [Online]. Available: http://hal.upmc.fr/hal-01083879

[10] A. Volkova, T. Hilaire, and C. Q. Lauter, "Determining fixed-point formats for a digital filter implementation using the worst-case peak-gain measure," in *Asilomar Conference on Signals, Systems and Computers*, November 2015.

[11] 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

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

[13] 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.

[14] 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.

[15] 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.

[16] M. Potkonjak, M. Srivastava, and A. Chandrakasan, "Efficient substitution of multiple constant multiplications by shifts and additions using iterative pairwise matching," in *ACM IEEE Design Automation Conference*, San Diego, CA USA, 1994, pp. 189–194.

[17] N. Boullis and A. Tisserand, "Some optimizations of hardware multiplication by constant matrices," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1271–1282, 2005.

[18] M. Mehendale, S. D.Sherlekar, and G. Venkatesh, "Synthesis of multiplier-less FIR filters with minimum number of additions," in *IEEE/ACM International Conference on Computer-Aided Design*, San Jose, CA USA, 1995, pp. 668–671.

[19] Y. Voronenko and M. Püschel, "Multiplierless multiple constant multiplication," *ACM Trans. Algorithms*, vol. 3, no. 2, 2007.

[20] L. Aksoy, E. Costa, P. Flores, and J. Monteiro, "Exact and approximate algorithms for the optimization of area and delay in multiple constant

multiplications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 6, pp. 1013–1026, 2008.

[21] S. White, "Applications of distributed arithmetic to digital signal processing: A tutorial review," *IEEE ASSP Magazine*, no. 3, pp. 4–19, Jul. 1989.

[22] M. Kumm, K. Möller, and P. Zipf, "Dynamically reconfigurable FIR filter architectures with fast reconfiguration," in *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, Jul. 2013.

[23] "FIR suite," http://www.firsuite.net/.

[24] B. Lopez, T. Hilaire, and L.-S. Didier, "Sum-of-products evaluation schemes with fixed-point arithmetic, and their application to IIR filter implementation," in *Design & Architectures for Signal & Image Processing*. IEEE, Oct. 2012.