SOAP: Structural Optimization of Arithmetic Expressions for High-Level Synthesis

Xitong Gao, Samuel Bayliss, George A. Constantinides
Department of Electrical and Electronic Engineering
Imperial College London
London SW7 2AZ, United Kingdom
{xi.gao08, s.bayliss08, g.constantinides}@imperial.ac.uk

***Abstract*—This paper introduces SOAP, a new tool to automatically optimize the structure of arithmetic expressions for FPGA implementation as part of a high level synthesis flow, taking into account axiomatic rules derived from real arithmetic, such as distributivity, associativity and others. We explicitly target an optimized area/accuracy trade-off, allowing arithmetic expressions to be automatically re-written for this purpose. For the first time, we bring rigorous approaches from software static analysis, specifically formal semantics and abstract interpretation, to bear on source-to-source transformation for high-level synthesis. New abstract semantics are developed to generate a computable subset of equivalent expressions from an original expression. Using formal semantics, we calculate two objectives, the accuracy of computation and an estimate of resource utilization in FPGA. The optimization of these objectives produces a Pareto frontier consisting of a set of expressions. This gives the synthesis tool the flexibility to choose an implementation satisfying constraints on both accuracy and resource usage. We thus go beyond existing literature by not only optimizing the precision requirements of an implementation, but changing the structure of the implementation itself. Using our tool to optimize the structure of a variety of real world and artificially generated examples in single precision, we improve either their accuracy or the resource utilization by up to 60%.**

I. Introduction

The IEEE 754 standard [1] for floating-point computation is ubiquitous in computing machines. In practice, it is often neglected that floating-point computations almost always have roundoff errors. In fact, associativity and distributivity properties which we consider to be fundamental laws of real numbers no longer hold under floating-point arithmetic. This opens the possibility of using these rules to generate an expression equivalent to the original expression in real arithmetic, which could have better quality than the original when evaluated in floating point computation.

By exploiting rules of equivalence in arithmetic, such as associativity $(a + b) + c \equiv a + (b + c)$ and distributivity $(a + b) \times c \equiv a \times c + b \times c$, it is possible to automatically generate different implementations of the same arithmetic expression. We optimize the structures of arithmetic expressions in terms of the following two quality metrics relevant to FPGA implementation: the resource usage when synthesized into circuits, and a bound on roundoff errors when evaluated. Our goal is the joint minimization of these two quality metrics. This optimization process provides a Pareto optimal set of implementations. For example, our tool discovered that with single precision floating-point representation, if $a \in [0.1, 0.2]$, then the expression $(a+1)^2$ uses fewest resources when implemented in the form $(a+1) \times (a+1)$ but most accurate when expanded into $(((a \times a) + a) + a) + 1$. However it turns out that a third alternative, $((1 + a) + a) + (a \times a)$, is never desirable because it is neither more accurate nor uses fewer resources than the other two possible structures. Our aim is to automatically detect and utilize such information to optimize the structure of expressions.

A naïve implementation of equivalent expression finding would be to explore all possible equivalent expressions to find optimal choices, However this would result in combinatorial explosion [2]. For instance, in worst case, the parsing of a simple summation of n variables could result in $(2n-1)!! = 1 \times 3 \times 5 \times \cdots \times (2n-1)$ distinct expressions [2], [3]. This is further complicated by distributivity as $(a+b)^k$ could expand into an expression with a summation of 2^k terms each with $k-1$ multiplications. Therefore, usually it would be infeasible to generate a complete set of equivalent expressions using the rules of equivalence, since an expression with a moderate number of terms will have a very large number of equivalent expressions. The methodology explained in this paper makes use of formal semantics as well as abstract interpretation [4] to significantly reduce the space and time requirements and produce a subset of the Pareto frontier.

In order to further increase the options available in the Pareto frontier, we introduce freedom in choosing mantissa widths for the evaluation of the expressions. Generally as the precision of the evaluation increases, the utilization of resources increases for the same expression. This gives flexibility in the trade-off between resource usage and precision. Our approach and its associated tool, SOAP, allow high-level synthesis flows to automatically determine whether it is a better choice to rewrite an expression, or change its precision in order to meet optimization goals.

The three contributions of this paper are:

1) Efficient methods for discovering equivalent structures of arithmetic expressions.
2) A semantics-based program analysis that allows joint reasoning about the resource usage and safe ranges of values and errors in floating-point computation of arithmetic expressions.
3) A tool which produces RTL implementations on the

area-accuracy trade-off curve derived from structural optimization.

This paper is structured as follows. Section II discusses related existing work in high-level synthesis and the optimization of arithmetic expressions. We explain the basic concepts of semantics with abstract interpretation used in this paper in Section III. Using this, Section IV explains the concrete and abstract semantics for finding equivalent structure in arithmetic expressions, as well as the analysis of their resource usage estimates and bounds of errors. Section V gives an overview of the implementation details in our tool. Then we discuss the results of optimized example expressions in Section VI and end with concluding remarks in Section VII.

II. Related Work

High-level synthesis (HLS) is the process of compiling a high-level representation of an application (usually in C, C++ or MATLAB) into register-transfer-level (RTL) implementation for FPGA [5], [6]. HLS tools enable us to work in a high-level language, as opposed to facing labor-intensive tasks such as optimizing timing, designing control logic in the RTL implementation. This allows application designers to instead focus on the algorithmic and functional aspects of their implementation [5]. Another advantage of using HLS over traditional RTL tools is that a C description is smaller than a traditional RTL description by a factor of 10 [5], [7], which means HLS tools are in general more productive and less error-prone to work with. HLS tools benefit us in their ability to automatically search the design space with a reasonable design cost [7], explore a large number of trade-offs between performance, cost and power [8], which is generally much more difficult to achieve in RTL tools. HLS has received a resurgence of interest recently, particularly in the FPGA community. Xilinx now incorporates a sophisticated HLS flow into its Vivado design suite [9] and the open-source HLS tool, LegUp [10], is gaining significant traction in the research community.

However, in both commercial and academic HLS tools, there is very little support for static analysis of numerical algorithms. LLVM-based HLS tools such as Vivado HLS and LegUp usually have some traditional static analysis-based optimization passes such as constant propagation, alias analysis, bitwidth reduction or even expression tree balancing to reduce latency for numerical algorithms. There are also academic tools that perform precision-performance trade-off by optimizing word-lengths of data paths [11]. However there are currently no HLS tools that perform the trade-off optimization between accuracy and resource usage by varying the *structure* of arithmetic expressions.

Even in the software community, there are only a few existing techniques for optimizing expressions by transformation, none of which consider accuracy/run-time trade-offs. Darulova *et al.* [12] employ a metaheuristic technique. They use genetic programming to evolve the structure of arithmetic expressions into more accurate forms. However there are several disadvantages with metaheuristics, such as convergence can only be proved empirically and scalability is difficult to control because there is no definitive method to decide how long the algorithm must run until it reaches a satisfactory goal. Hosangadi *et al.* [13] propose an algorithm for the factorization of polynomials to reduce addition and multiplication counts, but this method is only suitable for factorization and it is not possible to choose different optimization levels. Peymandoust *et al.* [14] present an approach that only deals with the factorization of polynomials in HLS using Gröbner bases. The shortcomings of this are its dependence on a set of library expressions [13] and the high computational complexity of Gröbner bases. The method proposed by Martel [15] is based on operational semantics with abstract interpretation, but even their depth limited strategy is, in practice, at least exponentially complex. Finally Ioualalen *et al.* [2] introduce the abstract interpretation of equivalent expressions, and creates a polynomially sized structure to represent an exponential number of equivalent expressions related by rules of equivalence. However it restricts itself to only a handful of these rules to avoid combinatorial explosion of the structure and there are no options for tuning its optimization level.

Since none of these above captures the optimization of both accuracy and performance by restructuring arithmetic expressions, we base ourselves on the software work of Martel [15], but extend this work in the following ways. Firstly, we develop new hardware-appropriate semantics to analyze not only accuracy but also resource usage, seamlessly taking into account common subexpression elimination. Secondly, because we consider both resource usage and accuracy, we develop a novel multi-objective optimization approach to scalably construct the Pareto frontier in a hierarchical manner, allowing fast design exploration. Thirdly, equivalence finding is guided by prior knowledge on the bounds of the expression variables, as well as local Pareto frontiers of subexpressions while it is optimizing expression trees in a bottom-up approach, which allows us to reduce the complexity of finding equivalent expressions without sacrificing our ability to optimize expressions.

We begin with an introduction to formal semantics in the following section, later in Section IV, we explain our approach by extending the semantics to reason about errors, resource usage and equivalent expressions.

III. Abstract Interpretation

This section introduces the basic concepts of formal semantics and abstract interpretation used in this paper. We illustrate these concepts by putting the familiar idea of interval arithmetic [16] in the framework of abstract interpretation. This is then further extended later in the paper to define a scalable analysis capturing ranges, errors and resource utilization. As an illustration, consider the following expression and its DFG:

$$(a + b) \times (a + b) \tag{1}$$

We may wish to ask: if initially a and b are real numbers in the range of $[0.2, 0.3]$ and $[2, 3]$ respectively, what would be the outcome of evaluating this expression with real arithmetic? A straightforward approach is simulation. Evaluating

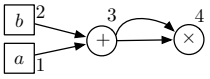


Fig. 1. The DFG for the sample program.

the expression for a large quantity of inputs will produce a set of possible outputs of the expression. However the simulation approach is unsafe, since there are infinite number of real-valued inputs possible and it is infeasible to simulate for all. A better method might be to represent the possible values of a and b using ranges. To compute the ranges of its output values, we could operate on ranges rather than values (note that the superscript $\sharp$ denotes ranges). Assume that $a^\sharp_{init} = [0.2, 0.3]$, $b^\sharp_{init} = [2, 3]$, which are the input ranges of a and b, and $A(l)$ where $l \in \{1, 2, 3, 4\}$ are the intervals of the outputs of the boxes labelled with l in the DFG. We extract the data flow from the DFG to produce the following set of equations:

$$\begin{aligned} A(1) &= a^\sharp_{init} & A(2) &= b^\sharp_{init} \\ A(3) &= A(1) + A(2) & A(4) &= A(3) \times A(3) \end{aligned} \tag{2}$$

For the equations above to make sense, addition and multiplication need to be defined on intervals. We may define the following interval operations:

$$\begin{aligned} &[a,b] + [c,d] = [a+c, b+d] \quad [a,b] - [c,d] = [a-d, b-c] \\ &[a,b] \times [c,d] = [\min(s), \max(s)] \\ &\qquad \text{where } s = \{a \times c, a \times d, b \times c, b \times d\} \end{aligned} \tag{3}$$

The solution to the set of (2) for $A(4)$ is $[4.84, 10.89]$, which represents a safe bound on the output at the end of program execution. Note that in actual execution of the program, the semantics represent the values of intermediate variables, which are real values. In our case, a set of real values forms the set of all possible values produced by our code. However computing this set precisely is not, in general, a possible task. Instead, we use abstract interpretation based on intervals, which gives the abstract semantics of this program. Here, we have achieved a classical interval analysis by *defining* the meaning of addition and multiplication on abstract mathematical structures (in this case intervals) which capture a safe approximation of the original semantics of the program. Later in Section IV, we further generalize the idea by defining the meaning of these operations on more complex abstract structures which allow us to scalably reason about the range, error, and area of FPGA implementations.

IV. Novel Semantics

A. Accuracy Analysis

We first introduce the concepts of the floating-point representation [1]. Any values v representable in floating-point with standard exponent offset can be expressed with the format given by the following equation:

$$v = s \times 2^{e+2^{k-1}-1} \times 1.m_1 m_2 m_3 \ldots m_p \tag{4}$$

In (4), the bit s is the sign bit, the k-bit unsigned integer e is known as the exponent bits, and the p-bits $m_1 m_2 m_3 \ldots m_p$ are the mantissa bits, here we use $1.m_1 m_2 m_3 \ldots m_p$ to indicate a fixed-point number represented in unsigned binary format.

Because of the finite characteristic of IEEE 754 floating-point format, it is not always possible to represent exact values with it. Computations in floating-point arithmetic often induces roundoff errors. Therefore, following Martel [15], we bound with ranges the values of floating-point calculations, as well as their roundoff errors. Our accuracy analysis determines the bounds of all possible outputs and their associated range of roundoff errors for expressions. For example, assume that $a \in [0.2, 0.3]$, $b \in [2.3, 2.4]$, it is possible to derive that in single precision floating-point computation with rounding to the nearest, $(a+b)^2 \in [6.24999857, 7.29000187]$ and the error caused by this computation is bounded by $[-1.60634534 \times 10^{-6}, 1.60634534 \times 10^{-6}]$.

We employ abstract error semantics for the calculation of errors described in [2], [15]. First we define the domain $\mathbb{E}^\sharp = \mathbf{Interval}_\mathbb{F} \times \mathbf{Interval}$, where $\mathbf{Interval}$ and $\mathbf{Interval}_\mathbb{F}$ respectively represent the set of real intervals, and the set of floating-point intervals (intervals exactly representable in floating-point arithmetic). The value $(x^\sharp, \mu^\sharp) \in \mathbb{E}^\sharp$ represents a safe bound on floating-point values and the accumulated error represented as a range of real values. Then addition and multiplication can be defined for the semantics as in (5):

$$\begin{aligned} &(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = (\uparrow^\sharp_\circ (x_1^\sharp + x_2^\sharp), \mu_1^\sharp + \mu_2^\sharp + \downarrow^\sharp_\circ (x_1^\sharp + x_2^\sharp)) \\ &(x_1^\sharp, \mu_1^\sharp) - (x_2^\sharp, \mu_2^\sharp) = (\uparrow^\sharp_\circ (x_1^\sharp - x_2^\sharp), \mu_1^\sharp - \mu_2^\sharp + \downarrow^\sharp_\circ (x_1^\sharp - x_2^\sharp)) \\ &(x_1^\sharp, \mu_1^\sharp) \times (x_2^\sharp, \mu_2^\sharp) = (\uparrow^\sharp_\circ (x_1^\sharp \times x_2^\sharp), \\ &\qquad x_1^\sharp \times \mu_2^\sharp + x_2^\sharp \times \mu_1^\sharp + \mu_1^\sharp \times \mu_2^\sharp + \downarrow^\sharp_\circ (x_1^\sharp \times x_2^\sharp)) \\ &\qquad\qquad \text{for } (x_1^\sharp, \mu_1^\sharp) \in \mathbb{E}^\sharp, (x_2^\sharp, \mu_2^\sharp) \in \mathbb{E}^\sharp \end{aligned} \tag{5}$$

The addition, subtraction and multiplication of intervals follow the standard rules of interval arithmetic defined earlier in (3). In (5), the function $\downarrow^\sharp_\circ : \mathbf{Interval} \to \mathbf{Interval}$ determines the range of roundoff error due to the floating-point computation under one of the rounding modes $\circ \in \{-\infty, \infty, 0, \neg 0, \sim\}$ which are round towards negative infinity, towards infinity, towards zero, away from zero and towards nearest floating-point value respectively. For instance, $\downarrow^\sharp_\sim ([a,b]) = [-z, z]$, where $z = \max(ulp(a), ulp(b))/2$, which denotes the maximum rounding error that can occur for values within the range $[a,b]$, and the unit of the last place (ulp) function $ulp(x)$ [17] characterizes the distance between two adjacent floating-point values f_1 and f_2 satisfying $f_1 \leq x \leq f_2$ [18]. In our analysis, the function ulp is defined as:

$$ulp(x) = 2^{e(x)+2^{k-1}-1} \times 2^{-p} \tag{6}$$

where $e(x)$ is the exponent of x, k and p are the parameters of the floating-point format as defined in (4). The function $\uparrow^\sharp_\circ : \mathbf{Interval} \to \mathbf{Interval}_\mathbb{F}$ computes the floating-point bound from a real bound, by rounding the infimum a and supremum b of the input interval $[a,b]$.

Expressions can be evaluated for their accuracy by the method as follows. Initially the expression is parsed into a data flow graph (DFG). By way of illustration, the sample

expression $(a+b)^2$ has the tree structure in Fig. 1. Then the exact ranges of values of a and b are converted into the abstract semantics using a cast operation as in (7):

$$cast(x^\sharp) = (\uparrow^\sharp_\circ(x^\sharp), \downarrow^\sharp_\circ(x^\sharp)) \tag{7}$$

For example, for the variable $a \in [0.2, 0.3]$ under single precision with rounding to nearest, $cast([0.2, 0.3]) = ([0.200000003, 0.300000012], [-1/67108864, 1/67108864])$.

After this, the propagation of bounds in the data flow graph is carried out as described in Section III, where the difference is the abstract error semantics defined in (5) is used in lieu of the interval semantics. At the root of the tree (*i.e.* the exit of the DFG) we find the value of the accuracy analysis result for the expression.

In this paper, the function $\mathrm{Error} : \mathbf{Expr} \to \mathbb{E}^\sharp$ is used to represent the analysis of evaluation accuracy, where $\mathbf{Expr}$ denotes the set of all expressions.

B. Resource Usage Analysis

Here we define similar formal semantics which calculate an approximation to the FPGA resource usage of an expression, taking into account common subexpression elimination. This is important as, for example, rewriting $a\times b+a\times c$ as $a\times(b+c)$ in the larger expression $(a \times b + a \times c) + (a \times b)^2$ causes the common subexpression $a \times b$ to be no longer present in both terms. Our analysis must capture this.

The analysis proceeds by labelling subexpressions. Intuitively, the set of labels $\mathbf{Label}$, is used to assign unique labels to unique expressions, so it is possible to easily identify and reuse them. For convenience, let the function $\mathit{fresh} : \mathbf{Expr} \to \mathbf{Label}$ assign a distinct label to each expression or variable, where $\mathbf{Expr}$ is the set of all expressions. Before we introduce the labeling semantics, we define the environment $\lambda : \mathbf{Label} \to \mathbf{Expr} \cup \{\bot\}$, which is a function that maps labels to expressions, and $\mathbf{Env}$ denotes the set of such environments. A label l in the domain of $\lambda \in \mathbf{Env}$ that maps to $\bot$ indicates that l does not map to an expression. An element $(l, \lambda) \in \mathbf{Label} \times \mathbf{Env}$ stands for the labeling scheme of an expression. Initially, we map all labels to $\bot$, then in the mapping λ, each leaf of an expression is assigned a unique label, and the unique label l is used to identify the leaf. That is for the leaf variable or constant x:

$$(l, \lambda) = (\mathit{fresh}(x), [\mathit{fresh}(x) \mapsto x]) \tag{8}$$

This equation uses $[\mathit{fresh}(x) \mapsto x]$ to indicate an environment that maps the label $\mathit{fresh}(x)$ to the expression x and all other labels map to $\bot$, in other words, if $l = \mathit{fresh}(x)$ and $l' \neq l$, then $\lambda(l) = x$ and $\lambda(l') = \bot$. For example, for the DFG in Fig. 1, we have for the variables a and b:

$$\begin{aligned}(l_a, \lambda_a) &= (\mathit{fresh}(a), [\mathit{fresh}(a) \mapsto a]) = (l_1, [l_1 \mapsto a]) \\ (l_b, \lambda_b) &= (l_2, [l_2 \mapsto b])\end{aligned} \tag{9}$$

Then the environments are propagated in the flow direction of the DFG, using the following formulation of the labeling semantics:

$$\begin{aligned}(l_x, \lambda_x) \circ (l_y, \lambda_y) &= (l, (\lambda_x \odot \lambda_y)[l \mapsto l_x \circ l_y]) \\ &\text{where } l = \mathit{fresh}(l_x \circ l_y), \circ \in \{+, -, \times\}\end{aligned} \tag{10}$$

Specifically, $\lambda = \lambda_x \odot \lambda_y$ signifies that λ_y is used to update the mapping in λ_x, if the mapping does not exist in λ_x, and result in a new environment λ; and $\lambda[l \mapsto x]$ is a shorthand for $\lambda \odot [l \mapsto x]$. As an example, with the expression in Fig. 1, using (9), recall to mind that $l_1 = l_a$, $l_2 = l_b$, we derive for the subexpression $a + b$:

$$\begin{aligned}(l_{a+b}, \lambda_{a+b}) &= (l_a, \lambda_a) + (l_b, \lambda_b) \\ &= (l_3, (\lambda_a \odot \lambda_b)[l_3 \mapsto l_a + l_b]) \\ &\quad \text{where } l_3 = \mathit{fresh}(l_a + l_b) \\ &= (l_3, [l_1 \mapsto a] \odot [l_2 \mapsto b] \odot [l_3 \mapsto l_1 + l_2]) \\ &= (l_3, [l_1 \mapsto a, l_2 \mapsto b, l_3 \mapsto l_1 + l_2])\end{aligned} \tag{11}$$

Finally, for the full expression $(a + b) \times (a + b)$:

$$\begin{aligned}(l, \lambda) &= (l_{a+b}, \lambda_{a+b}) \times (l_{a+b}, \lambda_{a+b}) \\ &= (l_4, [l_1 \mapsto a, l_2 \mapsto b, l_3 \mapsto l_1 + l_2, l_4 \mapsto l_3 \times l_3])\end{aligned} \tag{12}$$

From the above derivation, it is clear that the semantics capture the reuse of subexpressions. The estimation of area is performed by counting, for an expression, the numbers of additions, subtractions and multiplications in the final labeling environment, then calculating the number of LUTs used to synthesize the expression. If the number of operators is $n_\circ$ where $\circ \in \{+, -, \times\}$, then the number of LUTs in total for the expressions is estimated as $\sum_{\circ\in\{+,-,\times\}} A_\circ n_\circ$, where the value $A_\circ$ denotes the number of LUTs per $\circ$ operator, which is dependent on the type of the operator and the floating-point format used to generate the operator.

In the following sections, we use the function $\mathrm{Area} : \mathbf{Expr} \to \mathbb{N}$ to denote our resource usage analysis.

C. Equivalent Expressions Analysis

In earlier sections, we introduce semantics that define additions and multiplications on intervals, then gradually transition to error semantics that compute bounds of values and errors, as well as labelling environments that allows common subexpression elimination, by defining arithmetic operations on these structures. In this section, we now take the leap from not only analyzing an expression for its quality, to defining arithmetic operations on sets of equivalent expressions, and use these rules to discover equivalent expressions. Before this, it is necessary to formally define equivalent expressions and functions to discover them.

1) Discovering Equivalent Expressions: From an expression, a set of equivalent expressions can be discovered by the following inference system of equivalence relations $\rhd \subset \mathbf{Expr} \times \mathbf{Expr}$. Let's define $e_1, e_2, e_3 \in \mathbf{Expr}$, $v_1, v_2, v_3 \in \mathbb{R}$,

and $\circ \in \{+, \times\}$. First, the arithmetic rules are:

$$\text{Associativity}(\circ) : (e_1 \circ e_2) \circ e_3 \rhd e_1 \circ (e_2 \circ e_3)$$
$$\text{Commutativity}(\circ) : e_1 \circ e_2 \rhd e_2 \circ e_1$$
$$\text{Distributivity} : e_1 \times (e_2 + e_3) \rhd e_1 \times e_2 + e_1 \times e_3$$
$$\text{Distributivity}' : e_1 \times e_2 + e_1 \times e_3 \rhd e_1 \times (e_2 + e_3)$$

Secondly, the reduction rules are:

$$\text{Identity}(\times) : e_1 \times 1 \rhd e_1 \qquad \text{ZeroProp} : e_1 \times 0 \rhd 0$$
$$\text{Identity}(+) : e_1 + 0 \rhd e_1 \qquad \text{ConstProp}(\circ) : \frac{v_3 = v_1 \circ v_2}{v_1 \circ v_2 \rhd v_3}$$

The ConstProp rule states that if an expression is a summation/multiplication of two values, then it can be simply evaluated to produce the result. Finally, the following one allows structural induction on expression trees, *i.e.* it is possible to derive that $a + (b + c) \rhd a + (c + b)$ from $b + c \rhd c + b$:

$$\text{Tree}(\circ) : \frac{e_1 \rhd e_2}{e_3 \circ e_1 \rhd e_3 \circ e_2} \tag{13}$$

It is possible to transform the structure of expressions using these above-mentioned inference rules. We define the function $\blacktriangleright : \mathcal{P}(\mathbf{Expr}) \to \mathcal{P}(\mathbf{Expr})$, where $\mathcal{P}(\mathbf{Expr})$ denotes the power set of $\mathbf{Expr}$, which generates a (possibly larger) set of equivalent expressions from an initial set of equivalent expressions by one step of (13), *i.e.* $\blacktriangleright(\epsilon) = \{e' \in \mathbf{Expr} \mid e \rhd e' \wedge e \in \epsilon\}$, where ϵ is a set of equivalent expressions. From this, we may note that the set of equivalent expressions generated by taking the union of any number of steps of (13) of ϵ is the transitive closure $\blacktriangleright^\star(\epsilon)$, as given by the following formula, where $\blacktriangleright^i(\epsilon) = \blacktriangleright(\blacktriangleright^{i-1}(\epsilon))$ and $\blacktriangleright^0(\epsilon) = \epsilon$:

$$\blacktriangleright^\star(\epsilon) = \bigcup_{i=0}^{\infty} \blacktriangleright^i(\epsilon) \tag{14}$$

We say that e_1 is equivalent to e_2 if and only if $e_1 \in \blacktriangleright^\star(\{e\})$ and $e_2 \in \blacktriangleright^\star(\{e\})$ for some expression e. In general because of combinatorial explosion, it is not possible to derive the full set of equivalent expressions by rules of equivalence, which motivates us to develop scalable methods that executes fast enough even with large expressions.

2) Scalable Methods: The complexity of equivalent expression finding is reduced by fixing the structure of subexpressions at a certain depth k in the original expression. The definition of depth is given as follows: first the root of the parse tree of an expression is assigned depth $d = 1$; then we recursively define the depth of a node as one more than the depth of its greatest-depth parent. If the depth of the node is greater than k, then we fix the structure of its child nodes by disallowing any equivalence transformation beyond this node. We let $\blacktriangleright_k$ denote this "depth limited" equivalence finding function, where k is the depth limit used, and $\blacktriangleright_k^\star$ denotes its transitive closure. This approach is similar to Martel's depth limited equivalent expression transform [15], however Martel's method eventually allows transformation of subexpressions beyond the depth limit, because rules of equivalence would transform these to have a smaller depth. This contributes to a time complexity at least exponential in terms of the expression size. In contrast, our technique has a time complexity that does not depend on the size of the input expression, but grows with respect to the depth limit k. Note that the full equivalence closure using the inference system we defined earlier in (14) is at least $O(2n - 1!!)$ where n is the number of terms in an expression, as we discussed earlier.

3) Pareto Frontier: Because we optimize expressions in two quality metrics, *i.e.* the accuracy of computation and the estimate of FPGA resource utilization. We desire the largest subset of all equivalent expressions E discovered such that in this subset, no expression dominates any other expression, in terms of having both better area and better accuracy. This subset is known as the Pareto frontier. Fig. 2 shows a naïve algorithm for calculating the Pareto frontier for a set of equivalent expressions ϵ.

```
function FRONTIER(ϵ)
    e_1, e_2, ..., e_n ← sort_by_accuracy(ϵ)
    e ← e_1; frontier ← {e}
    for i ∈ {1, 2, ..., n} do
        if Area(e_i) < Area(e) then
            frontier ← frontier ∪ {e_i}; e ← e_i
        end if
    end for
    return frontier
end function
```

Fig. 2. The Pareto frontier from a set of equivalent expressions.

The function `sort_by_accuracy` sorts a set of expressions by their analyzed bounds of errors in increasing order, where the magnitudes of error bounds are used to perform the ordering [15]:

$$\text{AbsError}(e) = \max(\text{abs}(\mu^\sharp_{\min}), \text{abs}(\mu^\sharp_{\max})) \quad \text{where } (x^\sharp, [\mu^\sharp_{\min}, \mu^\sharp_{\max}]) = \text{Error}(e) \tag{15}$$

4) Equivalent Expressions Semantics: Similar to the analysis of accuracy and resource usage, a set of equivalent expressions can be computed with semantics. That is, we define structures, *i.e.* sets of equivalent expressions, that can be manipulated with arithmetic operators. In our equivalent expressions semantics, an element of $\mathcal{P}(\mathbf{Expr})$ is used to assign a set of expressions to each node in an expression parse tree. To begin with, at each leaf of the tree, the variable or constant is assigned a set containing itself, as for x, the set ϵ_x of equivalent expressions is $\epsilon_x = \{x\}$. After this, we propagate the equivalence expressions in the parse tree's direction of flow, using (16), where $E_\circ(\epsilon_x, \epsilon_y) = \{e_x \circ e_y \mid e_x \in \epsilon_x \wedge e_y \in \epsilon_y\}$, and $\circ \in \{+, -, \times\}$:

$$\epsilon_x \circ \epsilon_y = \text{FRONTIER}\left(\blacktriangleright_k^\star\left(E_\circ(\epsilon_x, \epsilon_y)\right)\right) \tag{16}$$

The equation implies that in the propagation procedure, it recursively constructs a set of equivalent subexpressions for the parent node from two child expressions, and uses the depth limited equivalence function $\blacktriangleright_k^\star$ to work out a larger set of equivalent expressions. To reduce computation effort,

we select only those expressions on the Pareto frontier for the propagation in the DFG. Although in worst case the complexity of this process is exponential, the selection by Pareto optimality accelerates the algorithm significantly. For example, for the subexpression $a+b$ of our sample expression:

$$\begin{aligned}\epsilon_a + \epsilon_b &= \textsc{Frontier}\left(\blacktriangleright_k^\star\left(E_\circ(\epsilon_a, \epsilon_b)\right)\right)\\ &= \textsc{Frontier}\left(\blacktriangleright_k^\star\left(E_\circ(\{a\}, \{b\})\right)\right)\\ &= \textsc{Frontier}\left(\{a+b, b+a\}\right)\end{aligned} \tag{17}$$

Alternatively, we could view the semantics in terms of DFGs representing the algorithm for finding equivalent expressions. The parsing of an expression directly determines the structure of its DFG. For instance, the expression $(a + b) \times (a + b)$ generates the DFG illustrated in Fig. 3. The circles labeled 3 and 7 in this diagram are shorthands for the operation E_+ and $E_\times$ respectively, where E_+ and $E_\times$ is defined in (16).

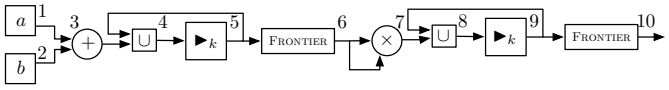


Fig. 3. The DFG for finding equivalent expressions of $(a + b) \times (a + b)$.

For our example in Fig. 3, similar to the construction of data flow equations in Section III, we can produce a set of equations from the data flow of the DFG, which now produces equivalent expressions:

$$\begin{aligned}
&A(1) = A(1) \cup \{a\} && A(2) = A(2) \cup \{b\}\\
&A(3) = E_+(A(1), A(2)) && A(4) = A(3) \cup A(5)\\
&A(5) = \blacktriangleright_k(A(4)) && A(6) = \textsc{Frontier}(A(5))\\
&A(7) = E_\times(A(6), A(6)) && A(8) = A(7) \cup A(9)\\
&A(9) = \blacktriangleright_k(A(8)) && A(10) = \textsc{Frontier}(A(9))
\end{aligned} \tag{18}$$

Because of loops in the DFG, it is no longer trivial to find the solution. In general, the analysis equations are solved iteratively. We can regard the set of equations as a single transfer function F as in (19), where the function F takes as input the variables $A(1), \ldots, A(10)$ appearing in the right-hand sides of (18) and outputs the values $A(1), \ldots, A(10)$ appearing in the left-hand sides. Our aim is then to find an input $\vec{x}$ to F such that $F(\vec{x}) = \vec{x}$, *i.e.* a fixpoint of F.

$$F((A(1), \ldots, A(10))) = (A(1) \cup \{a\}, \ldots, \textsc{Frontier}(A(9))) \tag{19}$$

Initially we assign $A(i) = \varnothing$ for $i \in \{1, 2, \ldots, 10\}$, and we denote $\vec{\varnothing} = (\varnothing, \cdots, \varnothing)$. Then we compute iteratively $F(\vec{\varnothing})$, $F^2(\vec{\varnothing}) = F(F(\vec{\varnothing}))$, and so forth, until the fixpoint solution $F^n(\vec{\varnothing})$ is reached for some iteration n, *i.e.* $F^n(\vec{\varnothing}) = F(F^n(\vec{\varnothing})) = F^{n+1}(\vec{\varnothing})$. The fixpoint solution $F^n(\vec{\varnothing})$ gives a set of equivalent expressions derived using our method, which is found at $A(10)$. In essence, the depth limit acts as a sliding window. The semantics allow hierarchical transformation of subexpressions using a depth-limited search and the propagation of a set of subexpressions that are locally Pareto optimal to the parent expressions in a bottom-up hierarchy.

The problem with the semantics above is that the time complexity of $\blacktriangleright_k^\star$ scales poorly, since the worst case number of subexpressions needed to explore increases exponentially with k. Therefore an alternative method is to optimize it by changing the structure of the DFG slightly, as shown in Fig. 4. The difference is that at each iteration, the Pareto frontier filters the results to decrease the number of expressions to process for the next iteration.

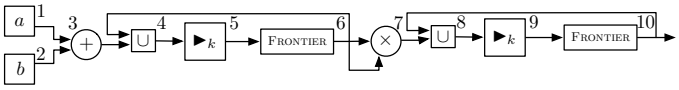


Fig. 4. The alternative DFG for $(a + b) \times (a + b)$.

In the rest of this paper, we use `frontier_trace` to indicate our equivalent expression finding semantics, and `greedy_trace` to represent the alternative method.

V. Implementation

The majority of our tool, SOAP, is implemented in Python. For computing errors in real arithmetic, we use exact arithmetic based on rational numbers within the GNU Multiple Precision Arithmetic Library (GMP) [19]. We also use the multi-precision floating-point library MPFR [20] for access to floating-point rounding modes and arbitrary precision floating-point computation.

Because of the workload of equivalent expression finding, the underlying algorithm is optimized in many ways. First, for each iteration, the relation finding function $\blacktriangleright_k$ is only applied to newly discovered expressions in the previous iteration. Secondly, results of functions such as $\blacktriangleright_k$, Area and Error are cached, since there is a large chance that these results from subexpressions are reused several times, subexpressions are also maximally shared to eliminate duplication in memory. Thirdly, the computation of $\blacktriangleright_k$ is fully multithreaded.

The resource statistics of operators are provided using FloPoCo [21] and Xilinx Synthesis Technology (XST) [22]. Initially, For each combination of an operator, an exponent width between 5 and 16, and a mantissa width ranging from 10 to 113, 2496 distinct implementations are generated using FloPoCo. All of them are optimized to use DSP blocks. They are then synthesized using XST, targeting a Virtex-6 FPGA device (XC6VLX760). Because LUTs are generally more contrained resources than DSP blocks in floating-point computations, we provide synthesis statistics in LUTs only.

VI. Results

Because Martel's approach defers selecting optimal options until the end of equivalent expression discovery, we developed a method that could produce exactly the same set of equivalent expressions from the traces computed by Martel, and has the same time complexity, but we adopted it to generate a Pareto frontier from the discovered expressions, instead of only error bounds. This allows us to compare `martel_trace`, *i.e.* our implementation of Martel's method, against our methods `frontier_trace` and `greedy_trace` discussed in Section IV-C. Fig. 5(a) optimizes the expression $(a + b)^2$ using the three methods above, all using depth limit 3, and the input ranges are $a \in [5, 10]$ and $b \in [0, 0.001]$ [15]. The IEEE754

single precision floating-point format with rounding to nearest was used for the evaluation of accuracy and area estimation. The scatter points represent different implementations of the original expression that have been explored and analyzed, and the (overlapping) lines denote the Pareto frontiers. In this example, our methods produce the same Pareto frontier that Martel's method could discover, while having up to 50% shorter run time. Because we consider an accuracy/area trade-off, we find that we can not only have the most accurate implementation discovered by Martel, but also an option that is only 0.0005% less accurate, but uses 7% fewer LUTs.

We go beyond the optimization of a small expression, by generating results in Fig 5(b) to show that the same technique is applicable to simultaneous optimization of multiple large expressions. The expressions e_x and e_y, with input ranges $a \in [1,2], b \in [10,20], c \in [10,200]$ are used as our example:

$$\begin{aligned} e_x &= (a+a+b) \times (a+b+b) \times (b+b+c) \times \\ &\quad (b+c+c) \times (c+c+a) \times (c+a+a) \\ e_y &= (1+b+c) \times (a+1+c) \times (a+b+1) \end{aligned} \tag{20}$$

We generated and optimized RTL implementations of e_x and e_y simultaneously using `frontier_trace` and `greedy_trace` with the depth limits indicated by the numbers in the legend of Fig. 5(b). Note that because the expressions evaluate to large values, the errors are also relatively large. We set the depth limit to 2 and found that `greedy_trace` executes much faster than `frontier_trace`, while discovering a subset of the Pareto frontier of `frontier_trace`. Also our methods are significantly faster and more scalable than `martel_trace`, because of its poor scalability discussed earlier, our computer ran out of memory before we could produce any results. If we normalize the time allowed for each method and compare the performance, we found that `greedy_trace` with a depth limit 3 takes takes slightly less time than `frontier_trace` with a depth limit 2, but produces a generally better Pareto frontier. The alternative implementations of the original expression provided by the Pareto frontier of `greedy_trace` can either reduce the LUTs used by approximately 10% when accuracy is not crucial, or can be about 10% more accurate if resource is not our concern. It also enables the ability to choose different trade-off options, such as an implementation that is 7% more accurate and uses 7% fewer LUTs than the original expression.

Furthermore, Fig. 5(c) varies the mantissa width of the floating-point format, and presents the Pareto frontier of both e_x and e_y together under optimization. Floating-point formats with mantissa widths ranging from 10 to 112 bits were used to optimize and evaluate the expressions for both accuracy and area usage. It turns out that some implementations originally on the Pareto frontier of Fig. 5(b) are no longer desirable, as by varying the mantissa width, new implementations are both more accurate and less resource demanding.

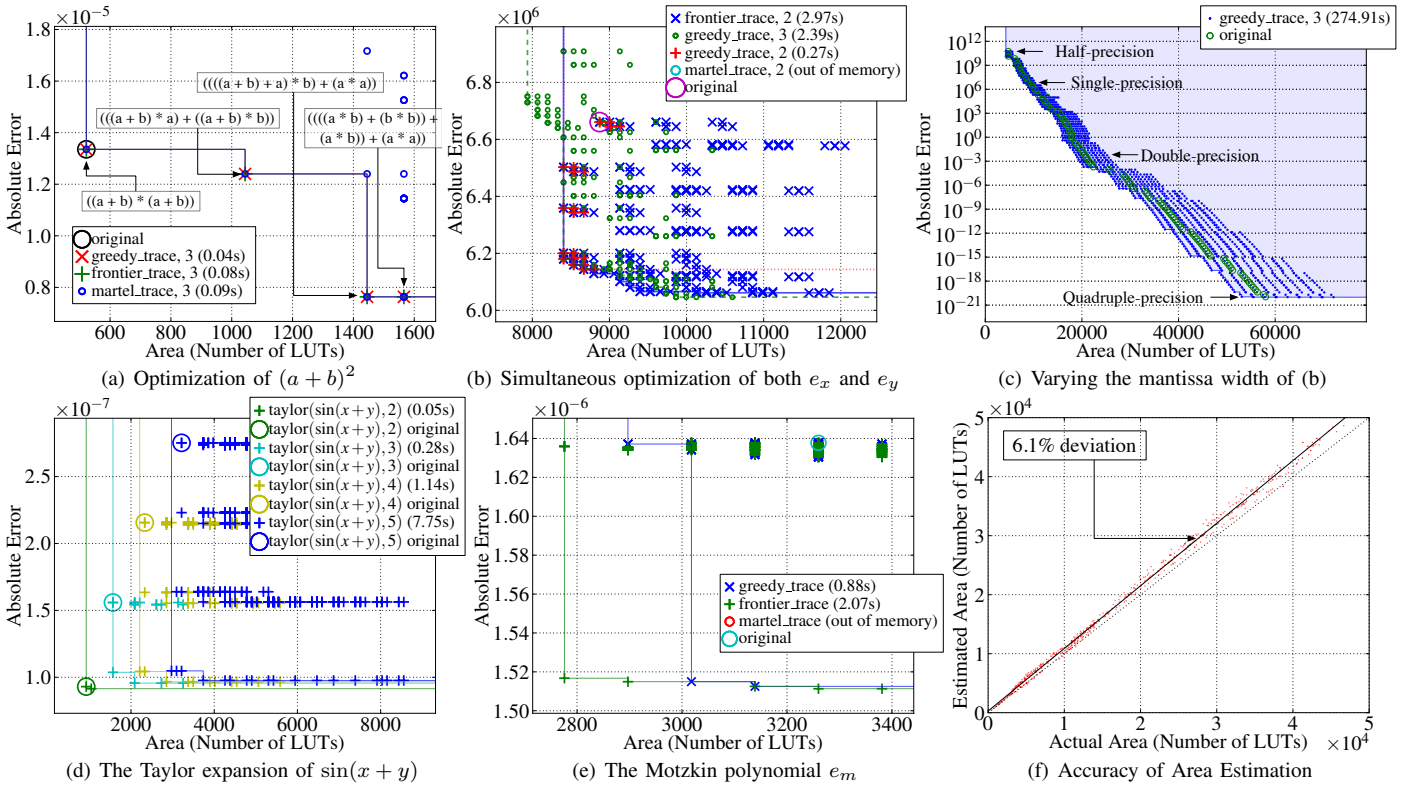
Besides the large example expressions above, Fig. 5(d) and Fig. 5(e) are produced by optimizing expressions with real applications under single precision. Fig. 5(d) shows the optimization of the Taylor expansion of $\sin(x+y)$, where $x \in [-0.1, 0.1]$ and $y \in [0,1]$, using `greedy_trace` with a depth limit 3. The function $\mathrm{taylor}(f,d)$ indicates the Taylor expansion of function $f(x,y)$ at $x=y=0$ with a maximum degree of d. For order 5 we reduced error by more than 60%. Fig. 5(e) illustrates the results obtained using the depth limit 3 with the Motzkin polynomial [23] $x^6+y^4z^2+y^2z^4-3x^2y^2z^2$, which is known to be difficult to evaluate accurately, especially using inputs $x \in [-0.99, 1]$, $y \in [1, 1.01]$, $z \in [-0.01, 0.01]$.

Finally, Fig. 5(f) demonstrates the accuracy of the area estimation used in our analysis. It compares the actual LUTs necessary with the estimated number of LUTs using our semantics, by synthesizing more than 6000 equivalent expressions derived from $a+b+c$, $(a+1)\times(b+1)\times(c+1)$, e_x, and e_y using varying mantissa widths. The dotted line indicates exact area estimation, a scatter points that is close to the line means the area estimation for that particular implementation is accurate. The solid black line represents the linear regression line of all scatter points. On average, our area estimation is a 6.1% over-approximation of the actual number of LUTs, and the worst case over-approximation is 7.7%.

VII. Conclusion

We provide a formal approach to the optimization of arithmetic expressions for both accuracy and resource usage in high-level synthesis. Our method and the associated tool, SOAP, encompass three kind of semantics that describe the accumulated roundoff errors, count operators in expressions considering common subexpression elimination, and derive equivalent expressions. For a set of input expressions, the proposed approach works out the respective sets of equivalent expressions in a hierarchical bottom-up fashion, with a windowing depth limit and Pareto selection to help reduce the complexity of equivalent expression discovery. Using our tool, we improve either the accuracy of our sample expressions or the resource utilization by up to 60%, over the originals under single precision. Our tool enables a high-level synthesis tool to optimize the structure as well as the precision of arithmetic expressions, then to automatically choose an implementation that satisfies accuracy and resource usage constraints.

We believe that it is possible to extend our tool for the multi-objective optimization of arithmetic expressions in the following ways. First, because of the semantics nature of our approach, it provides the necessary foundation which permits us to extend the method for general numerical program transformation in high-level synthesis. Secondly, it would be useful to further allow transformations of expressions while allowing different mantissa widths in the subexpressions, this further increases the number options in the Pareto frontier, as well as leads to more optimized expressions. Thirdly, as an alternative for interval analysis, we could employ more sophisticated abstract domains that capture the correlations between variables, and produce tighter bounds for results. Finally, there could be a lot of interest in the HLS community on how our



(a) Optimization of $(a+b)^2$

(b) Simultaneous optimization of both e_x and e_y

(c) Varying the mantissa width of (b)

(d) The Taylor expansion of $\sin(x+y)$

(e) The Motzkin polynomial e_m

(f) Accuracy of Area Estimation

Fig. 5. Optimization Results.

tool can be incorporated with Martin Langhammer's work on fused floating-point datapath synthesis [24].

In the future, after we extend our method and SOAP with general numerical program transform, we will make the tool available to the HLS community.

References

[1] ANSI/IEEE, "IEEE Standard for Floating-Point Arithmetic," Microprocessor Standards Committee of the IEEE Computer Society, Tech. Rep., Aug. 2008.

[2] A. Ioualalen and M. Martel, "A new abstract domain for the representation of mathematically equivalent expressions," in *Proceedings of the 19th International Conference on Static Analysis, SAS '12*. Springer-Verlag, 2012, pp. 75–93.

[3] C. Mouilleron, "Efficient computation with structured matrices and arithmetic expressions," Ph.D. dissertation, Ecole Normale Supérieure de Lyon-ENS LYON, 2011.

[4] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '77*. ACM, 1977, pp. 238–252.

[5] P. Coussy and A. Morawiec, *High-Level Synthesis: from Algorithm to Digital Circuit*, 1st ed. Springer-Verlag, 2008.

[6] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-level synthesis*. Kluwer Boston, 1992.

[7] Berkeley Design Technology, Inc, "High-level synthesis tools for Xilinx FPGAs," Berkeley Design Technology, Inc, Tech. Rep., 2010.

[8] M. C. McFarland, A. C. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proc. IEEE*, vol. 78, no. 2, pp. 301–318, 1990.

[9] Xilinx, "Vivado design suite user guide—high-level synthesis," 2012.

[10] U. of Toronto, "LegUp documentation—release 3.0," Jan. 2013. [Online]. Available: http://legup.eecg.utoronto.ca/

[11] G. Constantinides, A. Kinsman, and N. Nicolici, "Numerical data representations for FPGA-based scientific computing," *IEEE Des. Test. Comput.*, vol. 28, no. 4, pp. 8–17, 2011.

[12] E. Darulova, V. Kuncak, R. Majumdar, and I. Saha, "On the Generation of Precise Fixed-Point Expressions," École Polytechnique Fédérale De Lausanne, Tech. Rep., 2013.

[13] A. Hosangadi, F. Fallah, and R. Kastner, "Factoring and eliminating common subexpressions in polynomial expressions," in *Proceedings of the 2004 IEEE/ACM International Conference on Computer-aided Design, ICCAD '04*. IEEE Computer Society, 2004, pp. 169–174.

[14] A. Peymandoust and G. De Micheli, "Using symbolic algebra in algorithmic level DSP synthesis," in *Design Automation Conference, 2001. Proceedings*, 2001, pp. 277–282.

[15] M. Martel, "Semantics-based transformation of arithmetic expressions," in *Static Analysis, Lecture Notes in Computer Science*, vol. 4634. Springer-Verlag, 2007, pp. 298–314.

[16] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, 2009.

[17] J.-M. Muller, "On the definition of ulp(x)," in *Research report, Laboratoire de l'Informatique du Parallélisme, RR2005-09*, February 2005.

[18] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

[19] T. Granlund *et al.*, "GMP, the GNU multiple precision arithmetic library," 1991. [Online]. Available: http://gmplib.org/

[20] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 2, p. 13, 2007.

[21] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Des. Test. Comput.*, vol. 28, no. 4, pp. 18–27, 2011.

[22] Xilinx, Inc., *XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices*, Mar 2013. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_5/xst_v6s6.pdf

[23] J. Demmel, "Accurate and efficient expression evaluation and linear algebra," in *Proceedings of the 2011 International Workshop on Symbolic-Numeric Computation, SNC '11*. ACM, 2011, p. 2.

[24] M. Langhammer and T. VanCourt, "FPGA floating point datapath compiler," in *17th IEEE Symposium on Field Programmable Custom Computing Machines, FCCM '09*. IEEE, 2009, pp. 259–262.