Towards Line-aware Realizations of Expressions for HDL-based Synthesis of Reversible Circuits

Zaid Al-Wardi[1], Robert Wille[1,2], and Rolf Drechsler[1,2]

[1]Institute of Computer Science, University of Bremen
[2]Cyber-Physical Systems, DFKI GmbH, D-28359 Bremen, Germany

Abstract. *Hardware Description Languages* (HDLs) allow for the efficient synthesis of large and complex circuits. Consequently, researchers also investigated their potential in the domain of reversible logic. Here, existing HDL-based synthesis approaches suffer from the significant drawback of employing additional circuit lines in order to buffer intermediate results. In this work, we investigate the possibility of reducing this overhead. For this purpose, an alternative synthesis scheme is proposed and evaluated which aims at a more efficient realization of expressions. The general idea is to re-compute (i.e to undo) sub-expressions as soon as the respective intermediate results are not needed anymore. The observations and discussions result in initial guidelines on how to realize expressions more efficiently as well as a better understanding of the potential of HDL-based synthesis.

Keywords: Reversible circuits, synthesis, hardware description languages, optimization

1 Introduction

Motivated by applications e.g. in quantum computation [1], low-power design [2], or encoder and decoder design [3], research in the design of *reversible circuits* received significant interest. In the past decade, some substantial progress has been achieved in the development of corresponding (automated) design methods. This led to a variety of design solutions for a wide range of design tasks such as synthesis (see e.g. [4–8]), optimization (see e.g. [9,10]), verification (see e.g. [11, 12]), debugging (see e.g. [13]), and even automatic test pattern generation (see e.g. [14,15]).

Each design scenario results in a different circuit with equivalent functionality but with different cost parameters. In general, reversible circuit designers tend to synthesize circuits with a minimum number of lines. This is mainly motivated by the possible applications of reversible circuits in the domain of quantum computing, where circuit lines (realized by so-called qubits) are a very limited resource [1]. However, circuits with a minimal number of lines can, thus far, only been guaranteed by approaches that rely on Boolean/truth-table like synthesis approaches (e.g. [4–6]). These methods expand exponentially depending on the number of circuit inputs, which lead to the fact that these methods are practically applicable for simple design problems with a limited number of input signals only.

Investigating design flows that scale better and have the ability to handle complex systems with hundreds of input signals led to the hierarchical design approaches based on *Hardware Description Languages* (HDLs). *SyReC* is a reversible HDL, which has been introduced to facilitate the description of reversible circuits by means of simple high level codes [16]. A corresponding synthesis scheme showed the ability to describe and synthesize complex functionality such as a reversible CPU [17].

However, a major drawback of this approach is that it requires a significant number of additional circuit lines. These additional lines are used to buffer intermediate results needed in order to realize entire HDL-statements. Although first approaches aiming at the reduction of additional lines in HDL-based circuits have been introduced [18], they mainly focused on the realization of entire statements. However, further potential exists when also the realization of expressions (used in statements) are considered. This is motivated in more detail later in Section 3.

In this work, we investigate the possibility of improving the realization of expressions within the HDL-based synthesis of reversible circuits. The general idea is to re-compute (i.e. to undo) intermediate results of expressions as soon as they are not needed anymore. While this basically continues the idea of the "reversible undo" to the circuit realization of expressions, it also leads to new questions on how to realize the respective expressions in detail. Hence, we discuss some of the respective cases and provide suggestions on how to handle them best. Experimental case studies confirm the findings. This eventually provides new insights as well as ideas on how to improve HDL-based synthesis in general and leads to a better understanding of the remaining potential.

The remainder of this paper is structured as follows. The next section briefly reviews the background on reversible circuits, the HDL considered here, as well as the corresponding HDL-based synthesis scheme. Section 3 provides a motivation of this work and illustrates the general idea which, eventually, leads to an improved HDL-based synthesis scheme proposed in Section 4. Observations and discussions on the applicability of the proposed approach are given in Section 5. This is finally confirmed by an experimental case study summarized in Section 6 before the paper is concluded in Section 7.

2 Background

This section briefly reviews the basics on reversible circuits, a reversible HDL, as well as the corresponding HDL-based synthesis. It provides the necessary background to keep the paper self-contained.

2.1 Reversible Circuits

Reversible circuits realize functions $f : \mathbb{B}^n \to \mathbb{B}^n$ with a unique input/output mapping, i.e. bijections. A reversible circuit $G = g_1 \dots g_d$ is composed as a cascade of reversible gates g_i [1]. The inverse of G (representing the function f^{-1} and denoted by G^{-1}) can be obtained by cascading $g_d^{-1} g_{d-1}^{-1} \cdots g_1^{-1}$, where g_i^{-1} is the inverse gate of g_i. Since the self-inverse Toffoli and Fredkin gates are considered in this paper (see below), $g_i = g_i^{-1}$ holds and, thus, G^{-1} can simply be obtained by reversing the order of the gates of G.

For a set of Boolean signals $X = \{x_1, \dots, x_n\}$, a *reversible gate* has the form $g(C, T)$, where $C = \{x_{i_1}, \dots, x_{i_k}\} \subset X$ is the set of *control lines* and $T = \{x_{j_1}, \dots, x_{j_l}\} \subseteq X$ with $C \cap T = \emptyset$ is the non-empty set of *target lines.* The gate operation is applied to the target lines if, and only if, all control lines meet the required control conditions. Control lines and unconnected lines always pass through the gate unaltered.

In the literature, several types of reversible gates have been introduced. Usually, circuits realized by *Toffoli gates* and *Fredkin gates* are considered. A Toffoli gate has a single target line x_j and uniquely maps the input $(x_1, x_2, \dots, x_j, \dots, x_n)$ to the output $(x_1, x_2, \dots, x_{i_1} x_{i_2} \cdots x_{i_k} \oplus x_j, \dots, x_n)$. That is, a Toffoli gate inverts the target line if, and only if, all control lines are assigned the logic value 1. A Fredkin gate has two target lines x_{j_1} and x_{j_2} and interchanges their values if, and only if, the conjunction of all control lines evaluates to 1.

By definition, reversible circuits can only realize reversible functions. In order to realize non-reversible functions, *additional circuit lines* with constant inputs and garbage outputs (i.e. don't care outputs) are applied (see e.g. [19, 20]). Furthermore, additional circuit lines are also used frequently in hierarchical synthesis approaches (e.g. [7, 16]).

Example 1. Fig. 1 shows a reversible circuit realization of a 1-bit adder. Black circles represent control lines while $\oplus$ and $\times$ represent the target lines of a Toffoli and Fredkin gate, respectively. Since the adder is a non-reversible function, one additional circuit line is used to realize this function as a reversible circuit. The gates g_1, g_2, g_4, and g_5 are Toffoli gates, while the gate g_3 is a Fredkin gate.

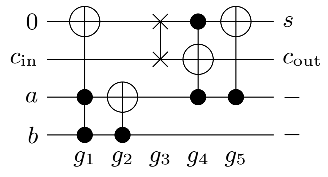

Fig. 1. Reversible circuit realizing a full adder

```
1 module example(in a(16), in b(16), in c(16), out f(16))
2   wire x(16)
3   x ^= (a & b)
4   x += (((a * b) + (a / b)) - ((a + c) / b))
5   f ^= (((x + b) ^ c) * ( a - b ))
```

Fig. 2. Simple SyReC program example

2.2 Reversible HDL

A major motivation of research in the domain of reversible circuit synthesis is the strive for a better scalability in order to enable the efficient design of complex functionality. Consequently, HDLs became a focus of ongoing research. A first version of an HDL for reversible circuits named *SyReC* has been introduced in [16]. SyReC is based on the reversible software language *Janus* [21], which has been enriched by further concepts (e.g. declaring circuit signals of different bit-widths), new operations (e.g. bit-access and shifts), and some restrictions (e.g. the prohibition of dynamic loops). In the following, we briefly review the main concepts of this HDL by means of Fig. 2 which depicts a simple SyReC specification[1].

This simple example shows that an HDL-circuit is described as a module. A module declaration starts by naming the module and, then, declaring the port signals for this module as in Line 1. This signal list associates each signal name with a type (i.e. in/out) and a bit-width (16 in the example above). Internal wire signals are defined within the scope of the module (Line 2) and are intermediately used to simplify the internal description of a module. These signals are transparent outside of the module. All signals represent non-negative integers or, in case of bit-width of 1, a Boolean.

A variety of statements and expressions are available to specify the functionality of the circuit without losing reversibility. Because of this, direct signal

[1] For a more detailed treatment, we refer to [16] as well as to the detailed documentation provided at the RevLib benchmark webpage [22].

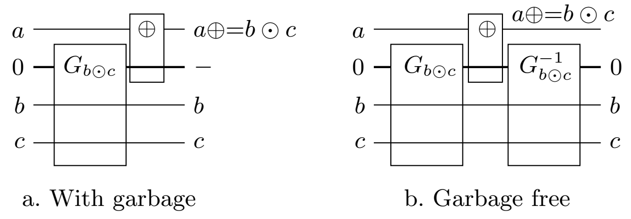

Fig. 3. Synthesis scheme

assignments of the form ($x = a$) are not allowed (as this would lead to a loss of the original value of x and, hence, will make the computation non-reversible). Consequently, signal assignments are restricted to so-called *reversible assignment operations*, i.e. the operations increase (+=), decrease (-=), and bit-wise XOR (ˆ =). These operations preserve the reversibility (i.e. it is possible to compute these operations in both directions) and they are generally denoted by $\oplus =$.

In contrast to the reversible operations, *binary operations* (denoted by $\odot$) which are not necessarily reversible (e.g. arithmetic, bit-wise, logical, or relational operations) and to be used only in right-hand expressions which preserve the values of the respective inputs. In doing so, all computations remain reversible since the input values can be applied to reverse any operation. For example, to specify the AND-operation in Line 3, a new free signal x in combination with a reversible assignment operation is applied. That results in the statement $x \wedge = (a\&b)$. All binary operations are written in the form: ($Operand_{left} \odot Operand_{right}$). An *Operand* can be a simple *signal*, an *integer*, or even another *expression* that has the same form. Nesting binary operations in such hierarchy, gives SyReC the ability to generate complex functions out of this basic set of binary-operators.

2.3 HDL-based Synthesis

In order to automatically synthesize the resulting designs, a hierarchical synthesis method is applied [16]. That is, existing realizations of the individual operations (i.e. building blocks) are combined so that the desired circuit is built. Fig. 3a illustrates the resulting scheme for the (generic) statement a $\oplus$= (b $\odot$ c). The $\oplus$-block ($\odot$-block) denotes thereby a building block for a reversible assignment operation (expression). Solid lines that cross the box represent the signals on the right-hand side of the statement, i.e. the signals whose values are preserved.

More precisely, the following two steps are performed:

1. Compose a sub-circuit $G_{\odot}$ realizing all the right-hand side expressions in the statement. For this purpose, use the respective building blocks. The result of the expression is buffered by means of additional circuit lines with constant input values.
2. Compose a sub-circuit $G_{\oplus}$ realizing the overall statement using the existing building blocks of the statement itself together with the buffered results of the expressions.

Obviously, this procedure leads to a significant number of additional circuit lines (and corresponding garbage outputs), since new circuit lines with constant values have to be introduced for each statement. Hence, an alternative has been evaluated in [18] where partial results (buffered in additional circuit lines) are inversely re-computed as soon as they are not needed anymore. This process (also called *reversible undo*) yields the original (constant) values on the already existing circuit lines which can be re-used e.g. in order to realize the following statements. More precisely, the synthesis scheme reviewed above is extended by a third step (see also Fig. 3b):

3) Add the inverse circuit from Step 1, i.e. $G_{\odot}^{-1}$ to the circuit in order to set the circuit lines buffering the result of the right-hand side expressions back to the constant 0.

In other words, the first sub-circuit $G_{b\odot c}$ ensures that the right-hand side expression is realized, sub-circuit $G_{a\oplus=b\odot c}$ ensures that the entire statement is realized, and sub-circuit $G_{b\odot c}^{-1}$ sets the circuit lines buffering the result of $b \odot c$ back to the constant 0.

3 Motivation and General Idea

Following the synthesis scheme reviewed in Section 2.3, the number of additionally required lines for the entire circuit depends on the statement with the "largest" expression. This is illustrated by means of the following example.

Example 2. Consider a sequence of three statements to be synthesized. Additionally, assume that 1, 6, and 4 circuit lines are needed to realize the respective expressions. Then, in total $\max\{1, 6, 4\} = 6$ additional circuit lines are needed to realize the respective circuit. Fig. 4 illustrates how these circuit lines are applied.

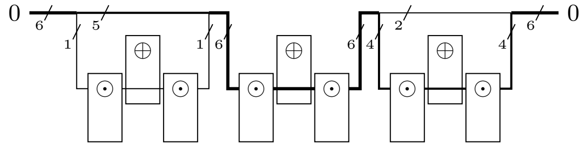

Fig. 4. Effect of the expression size

However, in many cases, even large expressions can be realized with a significantly smaller number of lines. To this end, consider the realization of arbitrary expressions. Expressions can be formulated as a variety of combinations of binary operations $\odot$ over circuit signals and (nested) sub-expressions. Each expression can thereby be represented as a binary-tree, where each node represents a binary operator which receives two inputs (operands) and buffers an output. The root node represents the entire expression, while the leafs represent the circuit signals. Obviously, it is impractical to provide a building block for each and every of such combinations. Hence, only building blocks (denoted by G_O) for each binary operation O are provided. Then, an expression E is realized by cascading the respective building blocks for each binary operation $\odot$ of E. For this purpose, additional circuit lines are required in order to buffer the respective intermediate results. The eventually resulting circuit is denoted by G_{E_i}, whereby i denotes the index of the root node of the expression E. This circuit requires a total of $(k \times w)$ additional circuits lines in order to buffer the intermediate results of the binary operations, whereby w denotes the bit-width of the circuit signals and k is the number of binary-operations (nodes) in the expression.

Example 3. Consider the expression E=(((a * b) + (a / b)) - ((a + c) / b)) which has been taken from Line 4 of the SyReC code shown in Fig. 2 and is composed of six binary expressions over 16-bit signals. The binary tree for this expression is shown in Fig. 5. Each node represents a binary operation (enumerated from O_1 to O_6) to be realized using the respective building blocks (i.e. G_{O_1}, $\ldots$, G_{O_6}). This leads to a reversible circuit $G_{E_6} = G_{O_1}\ G_{O_2}\ G_{O_3}\ G_{O_4}\ G_{O_5}\ G_{O_6}$ which requires a total of $6 \times 16 = 96$ additional circuits lines in order buffer the respective intermediate results.

When realizing such an expression, it is obvious that, eventually, only the result of the root operation is of interest. Circuit lines storing intermediate results can be re-computed back to their initial (constant) value as soon as they are not required anymore. Then, those lines would, in principle, be available to store other intermediate results needed in order to compute the overall expression. As a consequence, even large expressions could be realized with a significantly

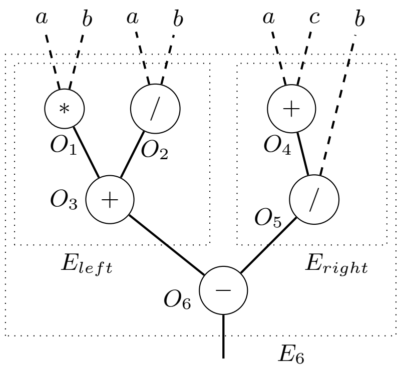

Fig. 5. The binary tree for the expression in Fig. 2 Line 5

smaller number of additional circuit lines compared to the currently applied synthesis scheme. Again, this is illustrated by means of an example.

Example 4. Consider again the expression E = (((a * b) + (a / b)) - ((a + c) / b)) from Example 3, which contains 6 operations. The actually desired result of the root operation (the subtraction O_6) is obtained using the intermediate results from the sub-expressions E_{left} = ((a * b) + (a / b)) and E_{right} = ((a + c) / b). The left sub-expression is realized as $G_{E_{left}} = G_{O_1}\ G_{O_2}\ G_{O_3}$ which requires a total of $3 \times 16 = 48$ additional circuit lines. However, once this sub-expression is realized and its result is buffered, the intermediate results of operations O_1 and O_2 are not needed anymore and can be recomputed back to their initial (constant) value – resulting in a circuit $G^{'}_{E_{left}} = G_{O_1}\ G_{O_2}\ G_{O_3} G^{-1}_{O_2}\ G^{-1}_{O_1}$. By this, 32 circuit lines with constant values become available and can be used in order to realize the right sub-expression. The entire expression E is then realized as

$$G_{E_6} = G^{'}_{E_{left}}\ G_{E_{right}}\ G_{O_6} = G_{O_1}\ G_{O_2}\ G_{O_3}\ G^{-1}_{O_2}\ G^{-1}_{O_1}\ G_{O_4}\ G_{O_5}\ G_{O_6}$$

and requires a total of 64 additional circuit lines (a significant reduction compared to the 96 additional circuit lines needed in Example 3).

Note that, in the following, we denote (sub-)circuits which immediately re-compute all not needed intermediate results back to the initial (constant) value by $G^{'}$.

In this work, we are aiming for investigating this potential in more detail. For this purpose, we propose a revised synthesis scheme for hardware description languages which re-computes circuit lines buffering intermediate results back to their initial (constant) value as soon as the respective intermediate value is not needed anymore. Afterwards, the effect as well as the possibilities of these changes in HDL-based synthesis are discussed in Section 5 and experimentally evaluated in Section 6.

4 Line-aware Synthesis of Expressions

Left and right operands in a binary expression E are independently considered as shown in Fig. 5. Therefore, they can be synthesized as two different building blocks $G_{E_{left}}$ and $G_{E_{right}}$, respectively. Afterwards, the corresponding results are fed to the building block G_{O_k} realizing the root-operation O_k. Now, realizing the left sub-expression not according to the original synthesis scheme (i.e. as $G_{E_{left}}$), but according to the ideas sketched in the previous section (i.e. as $G^{'}_{E_{left}}$), circuit lines applied in order to store intermediate results from E_{left} can be re-used for the realization of E_{right}. Then, the overall expression E can be realized as follows:

$$G_{E_k} \; = \; G^{'}_{E_{left}} \; G_{E_{right}} \; G_{O_k}$$

This scheme can recursively be applied for all sub-expressions eventually leading to the following (proposed) synthesis procedure:

```
Given  An expression $E$ to be realized,
       An indication whether a circuit $G$ or a circuit $G^{'}$ shall be realized
  1. **IF** ($E$ is a circuit signal only), **THEN** terminate the execution of this algorithm (base case of the recursion).
     **ELSE**, consider $E = E_{left} \odot E_{right}$ with $\odot$ being the root-operation $O_k$ realized by $G_{O_k}$.
  2. Recursively invoke this algorithm for expression $E_{left}$ in order to generate a sub-circuit $G^{'}_{E_{left}}$ realizing the left-operand.
  3. Recursively invoke this algorithm in order to to generate a sub-circuit $G_{E_{right}}$ realizing the right-operand.
  4. Combine the resulting sub-circuits to the following cascade:
     $G := \; G^{'}_{E_{left}} \; G_{E_{right}} \; G_{O_k}$
  5. **IF** a circuit $G'$ shall be realized **THEN**, re-compute intermediate results by adding the respective building blocks in a reverse fashion, i.e. extend the circuit to the following cascade:
     $G^{'} := G \; G^{-1}_{E_{right}} \; G^{'-1}_{E_{left}}$
```

5 Observations and Discussion

Having the scheme proposed in the previous section, a detailed analysis and discussion on the potential of re-computing intermediate results as soon as possible can be conducted. This section is devoted to that. More precisely, several cases are discussed showcasing when the application of the proposed algorithm is beneficial and when it may turn out to be disadvantageous. This can be used to get inspirations for best practices as well as an understanding of the characteristics that apply when synthesizing HDL descriptions as reversible circuits.

5.1 Reducing the number of lines is not always rewarding

The ideas in the proposed synthesis scheme are based on the desire of reducing the number of lines in the resulting circuit. The total number of additionally required circuit lines is still bounded by the number of lines required for synthesizing the largest statement. Hence, improving the number of lines for a "smaller" statement does not really help in reducing the total number of lines in the circuit. Moreover, reducing the number of lines for this smaller statement leads to additional circuit costs, since re-computing intermediate results is conducted by adding further building blocks in a reverse fashion.

Example 5. Consider again the program from Fig. 2 as well as the sketch of its realization in Fig. 4. As can be seen, the statement in Line 4 has the largest expression and would require 6 additional circuit lines when synthesized using the original synthesis approach[2]. The statement in Line 5 has the second-largest expression and would require 4 additional circuit lines.

Now, if the synthesis approach proposed in Section 4 is applied in order to realize the largest expression, a reduction from 6 additional circuit lines to 4 circuit lines can be achieved. This is worthwhile as it indeed reduces the total number of circuit lines required for the entire circuit. However, applying the same scheme in order to improve the the second-largest expression does not lead to further global reductions. Although the expression itself could be realized with 3 rather than 4 additional circuit lines, the number of lines for the entire circuit would not change. Moreover, this reduction would increase the number of building blocks required for re-computing. Hence, this expression should be realized using the original synthesis scheme.

5.2 The shape of the expression tree has an impact

In contrast to the original synthesis scheme (reviewed in Section 2.3), the proposed synthesis scheme from Section 4 depends on the operation precedence within the expression. In the worst case, the proposed procedure results in the same result as when the original scheme would have been applied. This worst

[2] Note that the number of circuit lines has to be multiplied by the bit-width w of the circuit signals, however, is assumed to be constant and, hence, omitted for sake of clarity.

case occurs whenever all operations in the expression have a primary value (i.e. a signal or a number) as an operand. Then, the circuit lines can not be re-used until the root operation is calculated. In this case, a circuit with a linear number of lines results, i.e. with k additional circuit lines and k building blocks (k being the number of operations). An example of such case is shown in Fig. 6a. This case occurs when the precedence of operations is ordered either from left to right, or vice versa.

On the other hand, an expression which can be represented by a completely balanced tree can be realized with a logarithmic number of lines. Then, both operands always require the same number of lines which can frequently been re-computed and, hence, re-used. However, as already mentioned above, this comes with the price of larger gate costs as additional building blocks are required[3]. More precisely, this case would result in a circuit with $2 \cdot (\lceil log_2(k+1)\rceil) - 1$ additional circuit lines and $3^{(\lceil log_2(k+1)\rceil - 1)}$ building blocks. An example of such a case is shown in Fig. 6b.

The best case, which shows the biggest potential for the proposed procedure with respect to the number of lines, is observed when, for all the stages of the expression, the right operand requires exactly one circuit line less than the left operand. Then, the right operand can always reuse the buffers lines from the left operand and does not need to allocate an own one. An example of such a case is shown in Fig. 6c. The number of resulting circuit lines for this best can be approximated by the function $f(n) = f(n-1) + f(n-2) + 1$, whereby $f(0) = 0$ and $f(1) = 1$ (this sequence is related to the *Fibonacci sequence*), for example, if there are $n = 6$ lines, then $f(6) = 20$, i.e. an expression arranged in the best case with up to 20 operations, can be calculated by using 6 lines. With increasing k, the reduction ratio in the number of lines becomes even better – although, it is, practically, unlikely for such a single expression to occur in typical HDL statements.

Example 6. The following three expressions are actually equivalent to each other, each has 7 multiplication operations and the result is simply the product:

1. $(((((((a * b) * c) * d) * e) * f) * g) * h)$
   This case represent the worst case. It requires 7 additional circuit signals and requires 7 cascaded blocks to be realized.
2. $(((a * b) * (c * d)) * ((e * f) * (g * h)))$
   This case is the completely balanced case, which requires 5 additional circuit signals, but 9 cascaded blocks.

[3] In this sense, the proposed synthesis scheme goes in line with previous observations on the trade-off between circuit lines and gate costs as e.g. conducted in [23].

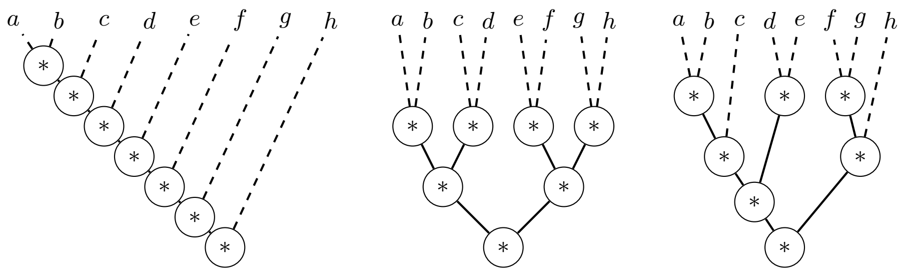

Fig. 6. Expressions with seven multiplication in different orders of precedence

3. $((((a * b) * c) * (d * e)) * ((f * g) * h))$
   This case is the best case where only 4 additional circuit signals are sufficient to realize the circuit by cascading 12 basic blocks.

For a designer writing HDL programs to be synthesized as a reversible circuit, it is important to be aware of the synthesizer features when writing the expression in order to write expressions, whenever possible, in the way that result in better circuits.

5.3 Exchanging the sub-expressions has an impact

As long as the left and right sub-expressions of an expression E are calculated independently, it is possible to exchange the order by which the respective sub-circuits are synthesized. This can be exploited when the right sub-expression requires a larger number of additional circuit lines for buffering intermediate results. Then, the expression E should be realized as $G := G'_{E_{right}} \ G_{E_{left}} \ G_{O_k}$. This has a slight benefit (precisely one signal is saved) compared to the original order: If the larger sub-expression is realized first, more signals can be re-computed. One of them can be used to buffer the result of the larger sub-expression. This signal is not needed to realize the other sub-expression since, as assumed before, this sub-expression is smaller. Because of the recurrent nature of the procedure, this one line reduction can be accumulated and result in a tangible reduction in the number of lines.

If the two expressions require the same number of signals, then no improvement with respect to the number of signals can be gained. Nevertheless, even then it might be beneficial to switch the sub-expressions. In fact, the sub-expression realized first is subject to an early re-computation. This gets more expensive for more "costly" operations. Hence, in case both sub-expressions require the same number of signals, the sub-expression with the "cheaper" building blocks should be realized first. This is illustrated by the following example.

Example 7. Consider E_{left} =((a*b)*(c*d)) and E_{right} =((aˆb)ˆ(cˆd)). If E_{left} is synthesized first, a circuit $G^{'}_{E_{left}} := G_{O_1}\ G_{O_2}\ G_{O_3}\ G^{-1}_{O_2}\ G^{-1}_{O_1}$ has to be generated, i.e. five building blocks for multiplication are required. The right sub-expression is then realized by $G_{E_{right}} := G_{O_1}\ G_{O_2}\ G_{O_3}$, i.e. three building blocks for the XOR operation are required. Exchanging the order would reverse that, i.e. result in five building blocks for the XOR operation and three building blocks for the multiplication. Since the realization of the multiplication requires a more expensive building block compared to the realization of the XOR operation, this would result in a much cheaper circuit.

Note that if the realization of the sub-expression is switched, this exchange must also be reflected in the respective re-computing cascade. That is, a corresponding circuit would have to be defined as $G^{'} := G\ G^{-1}_{E_{left}}\ G^{'-1}_{E_{right}}$, i.e. again with the right and left sub-circuit interchanged.

6 Experimental Case Studies

In order to experimentally evaluate the proposed concepts as well as the considered cases, the proposed synthesis approach has been implemented in C++. The resulting algorithm can be applied to various expressions and determines the number of lines as well as the number of required building blocks of the respectively resulting realization.

A main problem for the evaluation is that, thus far, not a very huge variety of HDL descriptions which are useful for benchmarking are available.

Hence, we manually created an initial benchmark set composed of two types of expressions which cover different cases, namely a *polynomial factored form* and a *majority function.*

These cases offer properties allowing to evaluate the behavior of representations e.g. in terms of a balanced tree or in the best possible case as discussed in Section 5.

In the following, the respective cases will explicitly be discussed. In addition to that, Table 1 provides a numerical summary. The first columns denote thereby the name of the respective case (*Case*), its order (*Order*, i.e. the size of the respective instantiation, and the number of operations in the resulting expression (*Op.*). Afterwards, the number of additionally required lines (*Lines*[4]) as

[4] Note that, again, the number of circuit lines has to be multiplied by the bit-width w of the circuit signals which, however, is assumed to be constant and, hence, omitted for sake of clarity.

well as the number of Blocks (*Blocks*) are provided for (1) the original synthesis approach as reviewed in Section 2.3 (*Orig. synth.*), (2) the proposed synthesis approach assuming the expression is/can be represented in terms of a balanced tree (*Balanced tree*), and (3) the proposed synthesis approach assuming the expression is/can be represented in the best case (*Best case*). In addition to the absolute values, also the percentual difference to the original synthesis approach is provided in columns labeled by *%*.

Already the numerical evaluation shows the potential of the proposed synthesis approach. In fact, significant reductions in the number of lines can be achieved. As discussed above, this comes at the price of an increased number of building blocks. In this sense, the proposed synthesis scheme goes in line with previous observations on the trade-off between circuit lines and gate costs as e.g. conducted in [23]. More detailed discussions follow with respect to the considered cases.

6.1 Polynomial Factored Form

The first case considers polynomials in the factored form, i.e. expressions of the form

$$(x + a_1)(x + a_2)\dots(x + a_m),$$

where m is the order of the polynomial. This form contains $(2 \times m - 1)$ operations and has been chosen to demonstrate a fully balanced-tree expression. In fact, this expression can be structured in a fashion so that sub-expressions of equal length result. This allows to represent the entire expression in terms of a balanced-tree. We can see from that the tendency to make both operands of the same size can dramatically decrease the number of additionally required lines. Furthermore, it can be noticed from Table 1 that the best case is not that much better in terms of the number of lines. In contrast, this may lead to significantly higher cost with respect to the needed building blocks.

6.2 Majority Function

The second case is considered in order to evaluate the algorithm on a logical expression that lacks the possibility to get represented either in form of a balanced tree or a best case tree. This case is carried out with the majority function, i.e. a Boolean function defined to determine if the majority of inputs are set to 1 or not. Two sub-cases are considered: the first is the majority of three inputs which is defined in the sum-of-products form as $(a\&b)|(a\&c)|(b\&c)$ and an according version for five inputs. For the first sub-case, this function shows only a slight change in the number of additionally required circuit lines, while the second sub-case unveils drastic reductions.

Table 1. Experimental case studies

Case	Order	Op. (k)	Orig. synth.		Balanced tree			Best case		
			Lines	Blocks	Lines	%	Blocks	Lines	%	Blocks
	4	7	7	7	5	29%	9	5	29%	15
Factored Polynommial	8	15	15	15	7	53%	27	6	60%	41
	16	31	31	31	9	71%	81	8	74%	205
Majority function	3	5	5	5	4	20%	7	4	20%	7
	5	29	29	29	8	72%	102	8	72%	170

7 Conclusion

In this work, an alternative procedure for HDL-based synthesis has been proposed which focused on a line-aware realization of expressions. The general idea was to re-compute intermediate results as soon as they are not needed anymore. By this, a significant amount of circuit lines can be saved. Nevertheless, the applicability of the proposed scheme significantly depends on the respectively applied expressions. Hence, we discussed possible cases and, by this, provided a better insight into the possible potential. Experimental case studies confirmed the findings. Future work will focus on the development of strategies for code optimization, e.g. term rewriting techniques that best exploit the potential of the proposed synthesis method. Besides that, how to reduce the number of required building blocks and, hence, the resulting gate costs of the obtained circuit remains an open issue to be addressed.

References

1. Nielsen, M., Chuang, I.: Quantum Computation and Quantum Information. Cambridge Univ. Press (2000)
2. Berut, A., Arakelyan, A., Petrosyan, A., Ciliberto, S., Dillenschneider, R., Lutz, E.: Experimental verification of Landauer's principle linking information and thermodynamics. Nature **483** (2012) 187–189
3. Wille, R., Drechsler, R., Oswald, C., Garcia-Ortiz, A.: Automatic design of low-power encoders using reversible circuit synthesis. In: Design, Automation and Test in Europe. (2012) 1036–1041
4. Miller, D.M., Maslov, D., Dueck, G.W.: A transformation based algorithm for reversible logic synthesis. In: Design Automation Conf. (2003) 318–323
5. Shende, V.V., Prasad, A.K., Markov, I.L., Hayes, J.P.: Synthesis of reversible logic circuits. IEEE Trans. on CAD **22**(6) (2003) 710–722
6. Wille, R., Le, H.M., Dueck, G.W., Große, D.: Quantified synthesis of reversible logic. In: Design, Automation and Test in Europe. (2008) 1015–1020
7. Wille, R., Drechsler, R.: BDD-based synthesis of reversible logic for large functions. In: Design Automation Conf. (2009) 270–275

8. Soeken, M., Wille, R., Hilken, C., Przigoda, N., Drechsler, R.: Synthesis of reversible circuits with minimal lines for large functions. In: ASP Design Automation Conf. (2012) 85–92
9. Feinstein, D.Y., Thornton, M.A., Miller, D.M.: Partially redundant logic detection using symbolic equivalence checking in reversible and irreversible logic circuits. In: Design, Automation and Test in Europe. (2008) 1378–1381
10. Soeken, M., Wille, R., Dueck, G.W., Drechsler, R.: Window optimization of reversible and quantum circuits. In: IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems. (2010)
11. Viamontes, G.F., Markov, I.L., Hayes, J.P.: Checking equivalence of quantum circuits and states. In: Int'l Conf. on CAD. (2007) 69–74
12. Wang, S.A., Lu, C.Y., Tsai, I.M., Kuo, S.Y.: An XQDD-based verification method for quantum circuits. IEICE Transactions **91-A**(2) (2008) 584–594
13. Wille, R., Große, D., Frehse, S., Dueck, G.W., Drechsler, R.: Debugging of Toffoli networks. In: Design, Automation and Test in Europe. (2009) 1284–1289
14. Polian, I., Fiehn, T., Becker, B., Hayes, J.P.: A family of logical fault models for reversible circuits. In: Asian Test Symp. (2005) 422–427
15. Wille, R., Zhang, H., Drechsler, R.: ATPG for reversible circuits using simulation, boolean satisfiability, and pseudo boolean optimization. In: IEEE Annual Symposium on VLSI. (2011) 120–125
16. Wille, R., Offermann, S., Drechsler, R.: SyReC: A programming language for synthesis of reversible circuits. In: Forum on Specification and Design Languages. (2010) 184–189
17. Wille, R., Soeken, M., Große, D., Schönborn, E., Drechsler, R.: Designing a RISC CPU in reversible logic. In: Int'l Symp. on Multi-Valued Logic. (2011) 170–175
18. Wille, R., Soeken, M., Schönborn, E., Drechsler, R.: Circuit line minimization in the HDL-based synthesis of reversible logic. In: IEEE Annual Symposium on VLSI. (2012) 213–218
19. Maslov, D., Dueck, G.W.: Reversible cascades with minimal garbage. IEEE Trans. on CAD **23**(11) (2004) 1497–1509
20. Wille, R., Keszöcze, O., Drechsler, R.: Determining the minimal number of lines for large reversible circuits. In: Design, Automation and Test in Europe. (2011) 1204–1207
21. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Symp. on Partial evaluation and semantics-based program manipulation. (2007) 144–153
22. Wille, R., Große, D., Teuber, L., Dueck, G.W., Drechsler, R.: RevLib: an online resource for reversible functions and reversible circuits. In: Int'l Symp. on Multi-Valued Logic. (2008) 220–225 RevLib is available at http://www.revlib.org.
23. Wille, R., Soeken, M., Miller, D.M., Drechsler, R.: Trading off circuit lines and gate costs in the synthesis of reversible logic. INTEGRATION, the VLSI Jour. **47**(2) (2014) 284–294