# Make It Reversible: Efficient Embedding of Non-reversible Functions

Alwin Zulehner[1] Robert Wille[1,2]
[1]Institute for Integrated Circuits, Johannes Kepler University, Linz, Austria
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
alwin.zulehner@jku.at robert.wille@jku.at

***Abstract*—Reversible computation became established as a promising concept due to its application in various areas like quantum computation, energy-aware circuits, and further areas. Unfortunately, most functions of interest are non-reversible. Therefore, a process called embedding has to be conducted to transform a non-reversible function into a reversible one – a coNP-hard problem. Existing solutions suffer from the resulting exponential complexity and, hence, are limited to rather small functions only. In this work, an approach is presented which tackles the problem in an entirely new fashion. We divide the embedding process into matrix operations, which can be conducted efficiently on a certain kind of decision diagram. Experiments show that improvements of several orders of magnitudes can be achieved using the proposed method. Moreover, for many benchmarks exact results can be obtained for the first time ever.**

## I. Introduction

In the last years, reversible computation became established as a promising concept for future directions in hardware design as well as a basis for emerging and future technologies. Following reversible computation means that any function to be realized must not only allow to determine the output assignment from a given input assignment, but also vice versa. Typical operations such as addition do not satisfy this criteria: having the sum only, the values of the summands cannot uniquely be determined. This characteristic allows for plenty of applications in various areas.

*Quantum computation* [9] represents one of the most important ones. This technology is theoretically capable to solve problems in polynomial time for which conventional computation only offers solutions with exponential complexity (e.g. factorization; see [11]). First physical accomplishments [14] further motivate a consideration of this technology. Reversible computation is of interest here, since every quantum computation is inherently reversible and, hence, certain parts of corresponding quantum circuits can directly be realized as reversible circuit [3].

A relation to *energy consumption of hardware* exists as shown by the seminal work of Landauer and Bennett [7], [4]. They proved that energy consumption of computations is closely related to information loss. Since reversible computation never loses information (in contrast e.g. to a non-reversible AND operation, where two bits are transformed into a single one – losing one bit of information), this paradigm (theoretically) allows for chips with close to zero energy dissipation. Also here, recent physical results confirmed these observations [5] and corresponding concepts such as energy recovery logic are currently considered [18], [6]. Although the gains obtained in this regard are still negligible at the moment, this may become a key factor for future hardware technologies where shrinking feature sizes pose a significant thread because of energy dissipation.

Besides that, reversible computation has successfully been applied in *further areas* such as the design of encoders (in general as done in [19] or for on-chip interconnects as done in [15], [17]), adiabatic computation [2], the exploitation of reversibility in conventional design tasks such as verification [1], and many more.

However, in order to actually exploit these benefits, many of the applications reviewed above require the respectively desired functionality – which usually includes non-reversible arithmetic operations such as addition, multiplication, etc. or logic non-reversible operations such as AND, OR, etc. – to be realized in a *reversible* fashion. To this end, a process called *embedding* [8] is be applied. Here, the desired (non-reversible) function is enriched by additional inputs and outputs to facilitate reversibility. Unfortunately, determining an embedding for a non-reversible function turned out to be a coNP-hard problem (as proven in [13]). In the worst case, all input/output relations have to be considered – yielding an exponential complexity.

First methods aiming for the determination of embeddings for non-reversible Boolean functions considered truth tables [8] – which obviously is not feasible for larger functions. Recently, more efficient approaches have been proposed, which rely on function representations such as binary decision diagrams [13]. But also here, the exponential complexity cannot really be tackled and, hence, results for functions with up to 30 variables only have been obtained thus far. Moreover, in many cases it even cannot be determined how many additional inputs and outputs are actually required in order to make a function reversible. Consequently, many large reversible functions are currently realized with a number of inputs/outputs that is magnitudes away from the actual optimum [16].

In this work, we propose an embedding method which tackles the underlying exponential complexity in an entirely new fashion. To this end, we apply a matrix representation of Boolean functions to derive the minimal number of additionally required outputs and, based on this number, determine a reversible embedding. Thereby, the embedding process is divided into matrix operations which can easily be employed on compact data-structures such as Quantum Multiple-valued Decision Diagrams (QMDDs [10]). Experimental evaluations show, that the proposed approach outperforms state of the art methods by several orders of magnitudes in terms of runtime. Moreover, exact results on the number of additionally required outputs as well as on corresponding embeddings can be obtained for some of the benchmarks for the first time ever.

The remainder of this paper is structured as follows. Section II briefly recapitulates reversible and non-reversible functions and their representation. In Section III, we give an overview of the embedding process and its main challenges. The proposed approaches for determining the minimal number of additional outputs as well as for deriving a corresponding reversibly embedded function are described in Section IV. Section V presents the experimental evaluation of the proposed approach compared to the state-of-the-art. The paper is concluded in Section VI.

## II. Background

We briefly recapitulate the basics of Boolean functions and their representation in this section.

| $x$ | $y$ | $x'$ | $y'$ |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 |

(a) Truth table

| Outputs \ Inputs ($x$ $y$) | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 00 | 0 | 0 | **1** | 0 |
| 01 | 0 | 0 | 0 | **1** |
| 10 | 0 | **1** | 0 | 0 |
| 11 | **1** | 0 | 0 | 0 |

(b) Permutation matrix

Fig. 1: Representations for reversible functions

| $x$ | $y$ | $x'$ | $y'$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

(a) Truth table

| Outputs \ Inputs ($x$ $y$) | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 00 | **1** | 0 | 0 | 0 |
| 01 | 0 | **1** | **1** | 0 |
| 10 | 0 | 0 | 0 | **1** |
| 11 | 0 | 0 | 0 | 0 |

(b) Function matrix

Fig. 2: Representations for the half adder function

### A. Reversible Functions and Their Representation

**Definition 1.** *A Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$ is* reversible*, iff $n = m$ and the function ensures a unique mapping from inputs to outputs, i.e. iff it forms a bijection.*

Reversible functions can be represented by permutation matrices.

**Definition 2.** *Let $f : \mathbb{B}^n \to \mathbb{B}^n$ be a Boolean reversible function. Then, the* permutation matrix *$M$ of $f$ is a $2^n \times 2^n$ matrix with elements $m_{i,j}$, $0 \leq i, j < 2^n$ such that*

$$m_{i,j} = \begin{cases} 1 & \text{if } f(j) = i \\ 0 & \text{otherwise} \end{cases}.$$

The columns (rows) of the permutation matrix represent the inputs (outputs). If an input maps to an output, the entry in the according row of the matrix is set to 1. All other entries in the permutation matrix are set to 0. Since there exists a unique mapping from inputs to outputs (this is always the case for reversible functions), each column and each row contains exactly one 1-entry.

**Example 1.** *Fig. 1a shows the truth table of a reversible function $f$. One can easily see that $f$ is reversible, since the numbers of inputs and outputs are equal and there exists a unique mapping from inputs to outputs, i.e. there exist no distinct input patterns which are mapped to the same output pattern. An alternative to the truth table is a permutation matrix as shown in Fig. 1b. For example, the second column of the permutation matrix represents the input 01 which, according to $f$, is supposed to map to the output* 10*. Hence, the second column contains its 1-entry in the third row (representing output* 10*).*

### B. Non-reversible Functions

Reversible functions are only a subset of all Boolean functions. Since a common representation for reversible and non-reversible function is desirable, *function matrices* have been introduced.

**Definition 3.** *Let $f : \mathbb{B}^n \to \mathbb{B}^m$ be a Boolean function. Then, the* function matrix *$M$ of $f$ is a $2^k \times 2^k$, $k = \max(n, m)$, matrix with elements $m_{i,j}$, $0 \leq i, j < 2^k$ such that*

$$m_{i,j} = \begin{cases} 1 & \text{if } f(j) = i \\ 0 & \text{otherwise} \end{cases}.$$

Each column of a function matrix contains one 1-entry only, but a row may contain multiple 1-entries, because more than one input combination may be mapped to the same output pattern.

**Example 2.** *Consider the function of a half adder as depicted in Fig. 2a. Two input combinations, namely 01 and 10, map to the same output, namely 01. Hence, the function is not reversible. Fig. 2b shows the corresponding function matrix. The non-reversibility can be seen in the second row (output 01) of the matrix: two 1-entries in a single row. These entries are in the second and third columns and, therefore, represent input combination 01 as well as 10. Additionally, there are only 0-entries in the last row of the functions matrix, which is also a violation of reversibility, because no input combination is mapped to output 11.*

## III. Existing Embedding Process

As discussed in Section I, reversible computation employs several benefits for conventional and particularly emerging technologies, but most functions of interest are not reversible. Hence, these functions have to be embedded into reversible ones. This section reviews this process and its main challenges.

We consider Boolean functions $f : \mathbb{B}^n \to \mathbb{B}^m$ with $n$ primary inputs and $m$ primary outputs. If $f$ is non-reversible, multiple input combinations are mapped to the same output pattern. Since reversibility requires a unique mapping from inputs to outputs, the output pattern must be made distinguishable. Therefore, additional outputs – so called garbage outputs – are added to the primary outputs. Assuming that the most frequent output pattern occurs $\mu$ times, $k = \lceil \log_2 \mu \rceil$ additional outputs are required to distinguish all occurrences of the pattern.

**Example 3.** *Consider again the half adder function as depicted in Fig 2a. The function is non-reversible since 2 input combinations, namely 01 and 10, map to the same output pattern (01). Since 01 is the most frequent output pattern and occurs twice, $k = \lceil \log_2 2 \rceil = 1$ garbage output is required.*

Inserting garbage outputs results in extra columns of the truth table. Since we are not interested in the value of the garbage outputs, they can be assigned arbitrarily. However, there are dependencies in the assignment: they have to be chosen in a way, such that the garbage outputs are assigned differently for all occurrences of an output pattern. In the following, this is represented by an asterisk (*).

**Example 3** (continued)**.** *The garbage outputs of input patterns 01 and 10 depend on each other, because these inputs map to the same output. As soon as the garbage output for one of the input patterns is fixed to 1 (0), the garbage output for the other input pattern must be fixed to 0 (1) to ensure reversibility.*

In addition to a unique mapping from inputs to outputs, reversibility requires that the number of inputs and outputs have to be equal. Therefore, if $n$ is larger than $m+k$, $n-m-k$ further garbage outputs are added and marked with *. In the opposite case, $m+k-n$ additional inputs (so called *ancillary inputs*) have to be added to the function. Each additional input doubles the number of rows in the truth table. If all ancillary inputs are assigned 0, the reversible function evaluates to the originally specified output. For all other assignments to the ancillary inputs, again arbitrary output values can be applied – even for the primary outputs. However, also here dependencies have to be considered. In fact, while the output pattern indeed is don't care in these cases, after all, each pattern is supposed to be applied only once in order to ensure reversibility. In the following, this is represented by a dot ($\cdot$).

TABLE I Embedding of the half adder function

(a) Degree of freedom

| $x$ | $y$ | $a$ | $x'$ | $y'$ | $g$ |
|---|---|---|---|---|---|
| **0** | **0** | 0 | **0** | **0** | * |
| 0 | 0 | 1 | · | · | · |
| **0** | **1** | 0 | **0** | **1** | * |
| 0 | 1 | 1 | · | · | · |
| **1** | **0** | 0 | **0** | **1** | * |
| 1 | 0 | 1 | · | · | · |
| **1** | **1** | 0 | **1** | **0** | * |
| 1 | 1 | 1 | · | · | · |

(b) One possible embedding

| $x$ | $y$ | $a$ | $x'$ | $y'$ | $g$ |
|---|---|---|---|---|---|
| **0** | **0** | 0 | **0** | **0** | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| **0** | **1** | 0 | **0** | **1** | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| **1** | **0** | 0 | **0** | **1** | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| **1** | **1** | 0 | **1** | **0** | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |

**Example 3** (continued)**.** *One ancillary input is needed to ensure that the number of inputs is equal to the number of outputs. This yields a truth table of the embedded half adder function as shown in Table Ia. If the additional input $a$ is set to 0, the intended function can be obtained from the outputs $x'$ and $y'$. The values of the additional garbage output $g$ as well as for the remaining input assignments (i.e. for $a \neq 0$) can arbitrarily be chosen (represented by * and ·, respectively) as long as the dependencies discussed above are considered.*

Finally, the embedding process is completed by assigning precise values to all entries represented by * and · while considering the discussed dependencies.

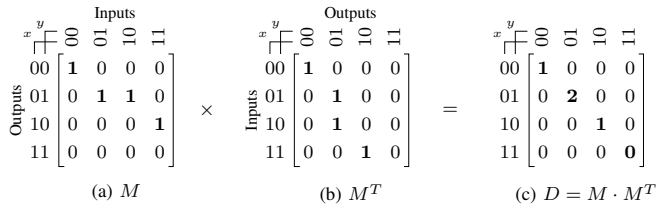
**Example 3** (continued)**.** *Assigning * and · with precise values eventually yields a reversible function as shown in Table Ib.*

The process reviewed above seems simple if small functions such as the half adder are considered. However, with increasing function size, embedding gets costly. In fact, it has been proven in [13] that embedding is coNP-complete. Two main challenges exist: First, determining the most frequent output pattern and, hence, the number $k = \lceil \log_2 \mu \rceil$ of additionally required garbage outputs requires a consideration of all possible output patterns in the worst case – an exponential complexity. Second, assigning all *- and ·-entries with precise values (while respecting the dependencies) is non-trivial.

In the past, embedding has mainly been conducted by means of truth tables [8] – obviously not feasible for large functions. Recently, researchers started investigations towards embedding for larger functions using *Binary Decision Diagrams* (BDDs). This led to improvements e.g. in the determination of the minimal number of actually required garbage outputs [16], [13] as well as more elaborated embedding methods [13]. However, while exact embedding still remained intractable for large functions, the work in [13] also solved this problem heuristically, i.e. an embedding where the number of additional outputs is not necessarily minimal. For these embeddings, the number of outputs is approximated by $n + m$, but it remains uncertain how far this if away from the actual minimum. Until today, no efficient method for the exact embedding of large functions nor a scalable method to determine the minimal number of additionally required outputs is available.

## IV. Alternative Embedding Scheme

In this section, we propose a method which, for the first time, allows for an exact embedding for larger functions. In contrast to previous approaches, which rely on truth tables (or representations of them like BDDs), our approach is based on function matrices. Our methodology is divided into two steps: First, the minimal number of additionally required garbage outputs is efficiently determined. Afterwards, the precise values for the respectively resulting *- and ·-entries are determined. Sections IV-A and IV-B show how these steps can be performed using function matrices. In Section IV-C, we discuss how the respective steps can efficiently be conducted (making the proposed method applicable for larger functions).




(a) $M$ (b) $M^T$ (c) $D = M \cdot M^T$

Fig. 3: Determine the minimal number of additional outputs

### A. Determining the Number of Garbage Outputs

As discussed in Section III, the number $\mu$ of occurrences of the most frequent output pattern has to be determined first. If the considered function $f$ is represented as a function matrix, $\mu$ is equal to the maximal number of 1-entries in a row of this function matrix. Since all entries in the function matrix are either 1 or 0, counting the 1-entries is equivalent to calculating the row sum.

**Example 4.** *Consider the function matrix of the half adder function as shown in Fig. 2b. The row sum for output pattern 01 is 2, since two input combinations (01 and 10) are mapped to this output pattern. The other row sums are either 0 or 1. Hence, the number of occurrences of the most frequent output pattern is $\mu = 2$.*

Forming the row sums of a function matrix $M$ turns out to be very simple: $M$ is a square matrix composed of Boolean values which contains exactly one 1 entry in each column. The product of $M$ with its transposed[1] matrix $M^T$ yields a diagonal matrix (i.e. a matrix where all entries off the main diagonal are zero) $D = M \cdot M^T$ with the row sums of $M$ in its main diagonal.

**Example 5.** *Fig. 3a shows the function matrix $M$ of the half adder function. Furthermore, Fig. 3b and 3c show its transposed matrix $M^T$ as well as the diagonal matrix $D = M \cdot M^T$, respectively. The product $M \cdot M^T$ contains the row sums of $M$, i.e. the number of occurrences of the output patterns, in its diagonal. Since no input maps to output 11 the fourth entry in the main diagonal is 0. The second entry of the diagonal is 2, since input pattern 01 as well as input pattern 10 map to output 01. Since this is the largest entry, $\mu = 2$.*

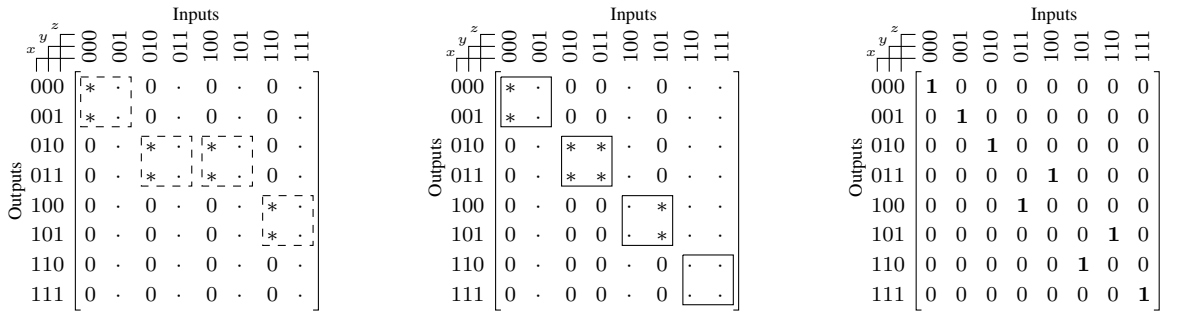
The proposed approach is fundamentally different from previous approaches, because the matrix multiplication allows to consider all output patterns concurrently. Experimental evaluation shows that our approach determines $\mu$ efficiently, whereas state of the art approaches require a substantial amount of run-time. More precisely, an improvement of several orders of magnitude in terms of run-time are observed.

### B. Assigning Precise Values

After determining $\mu$, it is simple to correspondingly extend the considered function matrix $M$ by the respective number $k = \lceil \log_2 \mu \rceil$ of additional inputs and outputs: Each entry of $M$ is simply replaced with a new $2^k \times 2^k$ matrix[2]. In order to keep the original function, we distinguish thereby between the corresponding 1-entries and 0-entires of $M$. More precisely, the 1-entries are replaced by a matrix $B_1$, whereas the 0-entries are replaced by a matrix $B_0$. The first column of the $2^k \times 2^k$ matrix $B_1$ contains *-entries only (representing that the desired output is determined when setting the additionally

[1] The transposed of a matrix is obtained by reflecting all elements along the main diagonal.

[2] If $m + k > n > k$, only $k + m - n$ inputs and outputs have to be added, since the function matrix already contains sufficient amount of garbage outputs.




(a) Extended function matrix (b) Block diagonal matrix (c) Resulting reversible embedding

Fig. 4: Embedding of the half adder using function matrices

added ancillary inputs to zero). In contrast, the first column of the $2^k \times 2^k$ matrix $B_0$ contains 0-entries only (representing that the input pattern must not be mapped to one of these output patterns). All remaining entries of $B_1$ and $B_0$ are filled with ·-entries. After replacing the 1-entries and 0-entries with the corresponding matrices, it remains open how to assign the respective *- and ·-entries with actual values (while, at the same time, respecting the dependencies). How to do this is covered in this section. To this end, we illustrate the remaining problem first.

**Example 6.** *In our running example, the half adder function is extended by one additional input/output (since $\lceil \log_2 2 \rceil = 1$). Therefore, the 1-entries and 0-entries of $M$ (cf. Fig. 3a) are replaced by $2^1 \times 2^1$ matrices $B_1$ and $B_0$, respectively, as shown in Fig. 4a ($B_1$-matrices are highlighted by dashed squares; all remaining $2 \times 2$-matrices are $B_0$ matrices). This ensures that e.g. the desired input pattern $xy = 10$ still maps to the desired output pattern $xy = 01$ while, at the same time, the flexibility of whether the extended input pattern $xyz = 100$ has to map to the extended output pattern $xyz = 010$ or the extended output pattern $xyz = 011$ is still guaranteed. Now, the problem remains what output pattern shall be chosen so that, eventually, a unique input/output mapping is guaranteed for the entire function.*

Hence, the challenge is how to assign precise values to the *- and ·-entries such that a reversible function results, i.e. such that each row and each column of the *function matrix* contains a single 1-entry only. A primitive solution would be to consider each row/column after each other. But again, this would tackle the exponential complexity in an enumerative and, thus, inefficient fashion. Hence, we aim for an alternative by focusing on the *-entries first (their assignment requires more dependencies to be considered). These dependencies can be resolved by transforming the matrix into a block diagonal matrix with blocks of dimension $2^k \times 2^k$ along the main diagonal. This can easily be achieved by swapping columns and results in a matrix where inherently all zero entries are located outside of the $2^k \times 2^k$ blocks along the main diagonal.

**Example 7.** *Consider again the extended matrix for the half adder function as shown in Fig. 4a. By swapping columns 100 and 011 as well as columns 110 and 101, a matrix as shown in Fig. 4b results, where all *-entires are located inside the $2 \times 2$ blocks (matrices) along the diagonal (highlighted by solid squares); all 0-entires are outside of these blocks.*

Recall that all *- and ·-entries have to be assigned in a reversible fashion (i.e. respecting the dependencies). Due to the last step, all these entries are now arranged along a diagonal. Hence, the desired assignment can easily be obtained by employing the identity function, i.e. setting all entries along the diagonal to 1. This clearly yields a reversible function (in fact, the identity function is the simplest reversible function), and implicitly satisfies all dependencies. At the same time, the originally desired function can easily be determined again by reverting the column swaps conducted before (possible since swapping of columns is a reversible operation). This eventually yields a complete matrix for the desired function which fully satisfies all dependencies and, hence, constitutes a full embedding.

**Example 7** (continued)**.** *Employing the identity function to the matrix in Fig. 4b yields an assignment to all *- and ·-entries which satisfies all dependencies and, hence, is reversible. By swapping again columns 100 and 011 as well as columns 110 and 101, the matrix as shown in Fig. 4c results which, eventually, describes the original function in a reversible fashion.*

### C. Implementation

In order to conduct the embedding process as proposed above, the following operations have to be executed on a function matrix: matrix multiplication, transposition, determining the largest value in the diagonal, extending the function matrix, and block diagoalization. Considering the exponential size of function matrices, this obviously is not feasible for large functions using this representation. However, we show that all these operations can efficiently be conducted on *Quantum Multiple-valued Decision Diagrams* (QMDDs) [10] – allowing to tackle the exponential complexity.This section briefly reviews QMDDs and shows how the additionally required operations have been implemented on top of it.

QMDDs as comprehensively described in [10] allow for an efficient representation and manipulation of function matrices. A QMDD represents a $2^n \times 2^n$ matrix with complex entries as a directed acyclic graph. Each node in this graph represents a (sub-)matrix

$$M = \begin{array}{c} \\ \text{Outputs} \end{array} \begin{array}{c} \overset{x}{\text{Inputs}} \\ \begin{array}{cc} 0 & 1 \end{array} \end{array} \begin{array}{c} \\ \begin{array}{c} 0 \\ 1 \end{array} \end{array} \begin{bmatrix} M_{00} & M_{01} \\ M_{10} & M_{11} \end{bmatrix}$$

which is recursively decomposed into four submatrices. This decomposition is conducted with respect to a variable $x$ (representing one bit of the input and the output) and yields four successors of the node as shown in Fig. 5: The left most edge represents a mapping of $x$ from input 0 to output 0 and, hence, leads to a node representing the submatrix $M_{00}$. The other edges represent, from left to right, the mapping of $x$ from 1 to 0, from 0 to 1, and from 1 to 1 (hence, leading to nodes representing the submatrices $M_{01}$, $M_{10}$, and $M_{11}$, respectively. If this decomposition yields a single matrix entry, a terminal node is reached.

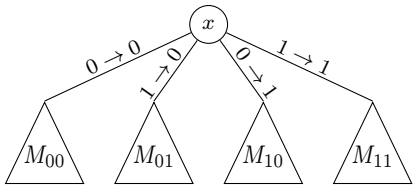


Fig. 5: QMDD node

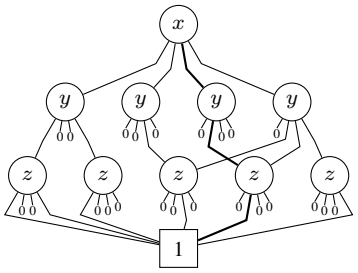


Fig. 6: QMDD of the embedded half adder

**Example 8.** *Fig. 6 shows a corresponding QMDD for the function matrix of the embedded half adder (i.e. for the function shown in Fig. 4c). Each path from the root of the QMDD to the 1-terminal represents a 1-entry of the function matrix, e.g. the path highlighted in bold, which represents the mapping from* $xyz = 011$ *to* $100$*. In contrast, a mapping from* $xy = 00$ *to* $01$ *terminates in a 0-stub, representing a* zero matrix *(i.e. a matrix consisting of 0-entries) only, independent of the variable* $z$*.*

This representation eventually allows for a compact representation, since many nodes (i.e. submatrices) are identical and, hence, can be shared (as can also be seen in the example). In fact, the number of nodes in the QMDDs is not exponential in most cases. Moreover, this representation also allows for an efficient matrix multiplication (as described in [10]) which is frequently applied during the embedding process proposed here. Furthermore, also all other operations needed for the proposed embedding scheme can efficiently be implemented on top of QMDDs as follows:

- The *transposition* of a matrix $M$ represented by a QMDD node can be realized as sketched in Fig. 7a. Here, the submatrices $M_{01}$ and $M_{10}$ are swapped and, afterwards, these swaps are recursively applied for all four submatrices.
- The *determination of the largest value* $\mu$ *along the diagonal* of a matrix $M$ can be realized by a post-order traversal of the QMDD nodes as sketched in Fig. 7b. Here, the maximal values $\mu_i$ of all submatrices is recursively determined and, afterwards, the maximum yields $\mu$. The maximal value of a terminal node is its value[3]. Since the submatrices $M_{01}$ and $M_{10}$ are *zero matrices* in a diagonal matrix, only submatrices $M_{00}$ and $M_{11}$ have to be considered.
- *Extending the function matrix* is easy for QMDDs if matrices $B_1$ and $B_0$ are modeled as QMDD as well. Then, all edges pointing to a 1- or 0-entry are replaced by edges pointing to $B_1$ and $B_0$, respectively. Without the loss of generality, we can represent all -entries as 0, since they are not considered during block diagonalization. Hence, $B_0$ is modeled as a *zero matrix*. Of course, 0-entries must not be replaced by $B_0$, because a 0-terminal already represents a *zero matrix*. Furthermore, we can chose the top most *-entry of $B_1$ to be set to 1, since the vertical position is changed after block diagonalization anyway. Thus, $B_1$ is represented by a QMDD which contains exactly one node for each garbage output. This eventually yields the extension as sketched in Fig. 7c[4].
- *Block diagonalization* of QMDDs can be accomplished by moving 1-entries from submatrices $M_{01}$ and $M_{10}$ to submatrices $M_{00}$ and $M_{11}$, respectively. This is sketched in Fig. 7d. This movement can be realized by swapping columns with Hamming distance of 1, which can be implemented as matrix multiplication as shown in [12]. Afterwards, the algorithm is recursively applied to the submatrices $M'_{00}$ and $M'_{11}$ until all 1-entries are located inside $2^k \times 2^k$ blocks along the main diagonal.

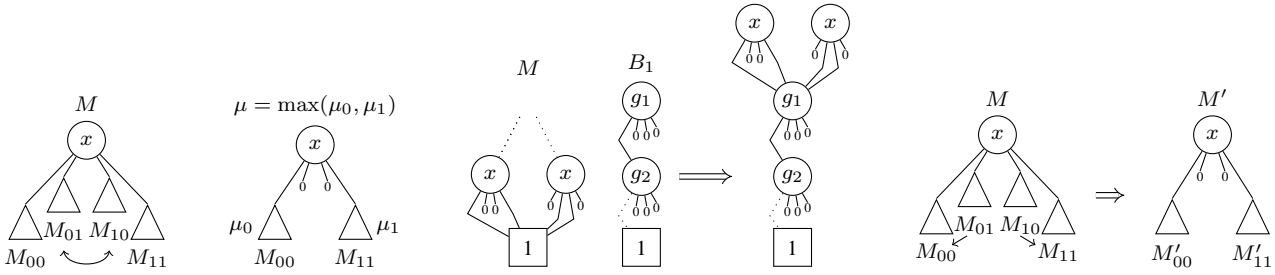
## V. Experimental Results

The method as well as the implementation described above eventually yields an alternative embedding tool which is significantly more efficient than the state-of-the-art. To confirm this, experimental evaluations have been conducted and their results have been compared to previous solutions. To this end, we implemented the methods described above in C++ on top of a QMDD implementation taken from [10] and applied the same benchmarks to it as used in the previous work: the ESOP-based and BDD-based approach of [13]. The experiments with the proposed approach have been carried out on a 3.20 GHz Intel i5 processor with 8 GB of main memory running Linux 4.2 – a machine which is similar to what has previously been used in [13]. That is, the obtained run-times are comparable. Table II summarizes all results.

As discussed above, the state of the art solutions can often not tackle the exponential complexity of the embedding problem. In many cases, they already fail in the first step of the embedding process, i.e. when determining the minimal number of garbage outputs.This is reviewed in the fourth and fifth column of Table II, which provide the run-times for the

[3]Note that QMDDs are implemented with a single terminal with value 1. Values other than 1 are represented by weighs attached to the edges. These weights serve as scalars the submatrices are multiplied with.

[4]Note that this process is a Kronecker multiplication of the original matrix with the corresponding $B_1$- and $B_0$- matrices.



(a) Transposition (b) Det. largest value (c) Extending the function matrix (d) Diagonalization

Fig. 7: Matrix operations on QMDDs

TABLE II Experimental results and comparison

| | | | | Garbage Outputs Determ. | | | Embedding Determ. | |
|---|---|---|---|---|---|---|---|---|
| Name | PI | PO | $k$[5] | ESOP [13] | BDD [13] | Proposed | ESOP [13] | Proposed |
| co14 | 14 | 1 | 14 | 72.8 | 0.0 | 0.2 | 118.6 | 0.2 |
| dc2 | 8 | 7 | 6 | 0.1 | 0.0 | 0.2 | 0.1 | 0.2 |
| example2 | 10 | 6 | 8 | 1.4 | 0.0 | 0.2 | 1.5 | 0.3 |
| inc | 7 | 9 | 5 | 0.0 | 0.0 | 0.2 | 0.0 | 0.2 |
| mlp4 | 8 | 8 | 5 | 0.2 | 0.0 | 0.2 | 0.2 | 0.2 |
| ryy6 | 16 | 1 | 16 | 157.7 | 0.0 | 0.2 | 258.8 | 0.3 |
| 5xp1 | 7 | 10 | 0 | 0.1 | 0.0 | 0.2 | 0.1 | 0.2 |
| t481 | 16 | 1 | 16 | 1717.8 | 0.0 | 0.2 | 2308.4 | 2.3 |
| x2 | 10 | 7 | 9 | 0.1 | 0.0 | 0.2 | 0.1 | 0.2 |
| add6 | 12 | 7 | 6 | 46.1 | 0.1 | 0.2 | 50.0 | 0.6 |
| cmb | 16 | 4 | 16 | 7.1 | 0.0 | 0.2 | 104.8 | 0.6 |
| ex1010 | 10 | 10 | 5 | 2.9 | 1.1 | 0.2 | 9.0 | 0.6 |
| pcler8 | 16 | 5 | 16 | 28.1 | 0.0 | 0.2 | 68.3 | 4.0 |
| tial | 14 | 8 | 11 | 1007.0 | 0.2 | 0.3 | 1047.3 | 8.3 |
| alu4 | 14 | 8 | 11 | 1270.3 | 0.1 | 0.3 | 1306.4 | 7.1 |
| apla | 10 | 12 | 10 | 0.0 | 0.0 | 0.2 | 0.5 | 0.5 |
| f51m | 14 | 8 | 11 | 556.5 | 0.2 | 0.3 | 595.7 | 6.9 |
| cu | 14 | 11 | 14 | 0.0 | 0.0 | 0.2 | 0.8 | 1.8 |
| in0 | 15 | 11 | 14 | 1.5 | 0.1 | 0.2 | 26.5 | 57.1 |
| 0410184 | 14 | 14 | 0 | 1227.8 | 7.4 | 0.4 | 1229.2 | 0.4 |
| apex4 | 9 | 19 | 7 | 1.0 | 25.0 | 0.3 | 39.8 | 0.6 |
| misex3 | 14 | 14 | 14 | 160.7 | 17.5 | 0.3 | 929.7 | 29.7 |
| misex3c | 14 | 14 | 7 | 327.2 | 2.8 | 0.2 | 472.0 | 28.9 |
| cm163a | 16 | 13 | 12 | 625.4 | 0.0 | 0.2 | 633.5 | 6.1 |
| bw | 5 | 28 | 4 | 0.0 | 0.0 | 0.3 | 0.1 | 0.3 |
| parity | 16 | 1 | 15 | >5000.0 | 0.9 | 0.4 | >5000.0 | 1.4 |
| cm150a | 21 | 1 | 21 | >5000.0 | 0.1 | 2.7 | >5000.0 | 2.7 |
| mux | 21 | 1 | 21 | >5000.0 | 0.1 | 2.5 | >5000.0 | 2.6 |
| cordic | 23 | 2 | 23 | >5000.0 | 0.1 | 0.3 | >5000.0 | 3783.8 |
| frg1 | 28 | 3 | 27 | >5000.0 | 0.0 | 0.3 | >5000.0 | 1522.3 |
| pdc | 16 | 40 | 15 | 31.1 | >5000.0 | 0.8 | >5000.0 | 1937.1 |
| spla | 16 | 46 | 15 | 32.7 | >5000.0 | 0.9 | >5000.0 | 738.7 |
| ex5p | 8 | 63 | 5 | 0.4 | >5000.0 | 1.2 | >5000.0 | 2.1 |
| apex2 | 39 | 3 | 39 | >5000.0 | 5.1 | 1.1 | >5000.0 | >5000.0 |
| e64 | 65 | 65 | 64 | 0.1 | >5000.0 | 1.5 | >5000.0 | >5000.0 |
| seq | 41 | 35 | 40 | >5000.0 | >5000.0 | 1.1 | >5000.0 | >5000.0 |
| cps | 24 | 109 | 23 | >5000.0 | >5000.0 | 4.5 | >5000.0 | >5000.0 |

PI: primary inputs PO: primary outputs $k = \lceil \log_2 \mu \rceil$: min. required garbage.
ESOP: Exact cube-based approach [13]; uses ESOP to det. $k$ and an exact embedding.
BDD: BDD-based approach [13] to det. $k$. Proposed: approach prop. in Section IV.

ESOP- and the BDD-based approach, respectively. In contrast, our solution can efficiently deal with this complexity – as shown in the sixth column of Table II providing the run-times for the solution as described in Section IV-A. While previous work often requires hundreds of CPU seconds or even times out in many cases, the solution proposed here is capable of determining the number of garbage outputs in a few seconds or often in a fraction of a second only. Moreover, for the first time ever this allows to obtain exact values for the benchmarks *seq* and *cps*; thus far, only a heuristic number of required garbage outputs was known for these cases (cf. [16]).

Besides that, major improvements can also be observed for the second step of the embedding process (the assignment of precise values). The last two columns of Table II provide the values for the determination of an exact embedding (including the determination of the number of garbage outputs). Again, the run-times needed by the current state of the art solution (i.e. the ESOP-based approach as proposed in [13]) as well as the run-times of the approach proposed in Section IV are provided[6]. Obviously, when already the determination of the number of garbage outputs failed, also no complete embedding can be created. Hence, no results could have been generated for all benchmarks which already timed out in the first step. Besides that, the state of the art solution significantly suffers from large run-times in the second step as well (e.g. *pdc*, *spla*, *ex5p*, and *e64* time out). Even though the proposed method requires some time to tackle the exponential complexity – in some cases even timeouts are reported – significant improvements compared to the state-of-the-art can be observed. In some cases, an embedding can be obtained up to four orders of magnitudes faster than with the state-of-the-art. For the benchmarks *parity*, *cm150a*, *mux*, *cordic*, *frg1*, *pdc*, *spla*, and *exp5p* an exact embedding was derived for the first time ever.

[5]Note that the total number of circuit lines is $\max(PI, PO + k)$.

[6]Note that the BDD-based approach presented in [13] is not capable of conducting the second embedding step in an exact fashion and, hence, is not considered here.

## VI. Conclusions

In this paper, we significantly improved the process of embedding non-reversible functions for reversible computation. This is a crucial step for many future directions in hardware design as well as for many emerging technologies which rely on this computing scheme. Although proven as a coNP-hard problem, a method has been presented which tackles the underlying exponential complexity in an efficient fashion. Experimental results showed that several major improvements can be accomplished with the proposed method: First, all embedding steps can been conducted significantly faster compared to the current state-of-the art – improvements in the runtime of up to four orders of magnitude have been observed. Besides that, exact results on the number of required garbage outputs or on the entire embedding have been determined for several benchmarks for the first time ever. This constitutes a major step forward considering that, thus far, many large reversible functions are currently realized with a number of inputs/outputs and/or an embedding which are magnitudes away from the actual optimum [16].

## Acknowledgements

This work has partially been supported by the European Union through the COST Action IC1405.

## References

[1] L. G. Amarù, P. Gaillardon, R. Wille, and G. D. Micheli. Exploiting inherent characteristics of reversible circuits for faster combinational equivalence checking. In *Design, Automation and Test in Europe*, pages 175–180, 2016.

[2] W. C. Athas and L. J. Svensson. Reversible logic issues in adiabatic cmos. In *Physics and Computation, 1994. PhysComp '94, Proceedings., Workshop on*, pages 111–118, Nov 1994.

[3] A. Barenco, C. H. Bennett, R. Cleve, D. DiVinchenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter. Elementary gates for quantum computation. *The American Physical Society*, 52:3457–3467, 1995.

[4] C. H. Bennett. Logical reversibility of computation. *IBM J. Res. Dev*, 17(6):525–532, 1973.

[5] A. Berut, A. Arakelyan, A. Petrosyan, S. Ciliberto, R. Dillenschneider, and E. Lutz. Experimental verification of Landauer's principle linking information and thermodynamics. *Nature*, 483:187–189, 2012.

[6] I. Hänninen, G. L. Snider, and C. S. Lent. Adiabatic CMOS: limits of reversible energy recovery and first steps for design automation. *Trans. Computational Science*, 24:1–20, 2014.

[7] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[8] D. Maslov and G. W. Dueck. Reversible cascades with minimal garbage. *IEEE Trans. on CAD*, 23(11):1497–1509, 2004.

[9] M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000.

[10] P. Niemann, R. Wille, D. M. Miller, M. A. Thornton, and R. Drechsler. QMDDs: Efficient quantum function representation and manipulation. *IEEE Trans. on CAD*, 35(1):86–99, 2016.

[11] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.

[12] M. Soeken, R. Wille, C. Hilken, N. Przigoda, and R. Drechsler. Synthesis of reversible circuits with minimal lines for large functions. In *ASP Design Automation Conf.*, pages 85–92, 2012.

[13] M. Soeken, R. Wille, O. Keszocze, D. M. Miller, and R. Drechsler. Embedding of large Boolean functions for reversible logic. *J. Emerg. Technol. Comput. Syst.*, 12(4):41:1–41:26, Dec. 2015.

[14] L. M. K. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang. Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414:883, 2001.

[15] R. Wille, R. Drechsler, C. Osewold, and A. G. Ortiz. Automatic design of low-power encoders using reversible circuit synthesis. In *Design, Automation and Test in Europe*, pages 1036–1041, 2012.

[16] R. Wille, O. Keszöcze, and R. Drechsler. Determining the minimal number of lines for large reversible circuits. In *Design, Automation and Test in Europe*, 2011.

[17] R. Wille, O. Keszocze, S. Hillmich, M. Walter, and A. G. Ortiz. Synthesis of approximate coders for on-chip interconnects using reversible logic. In *Design, Automation and Test in Europe*, 2016.

[18] Y. Ye and K. Roy. Energy recovery circuits using reversible and partially reversible logic. *Trans. on Circuits and Systems I: Fundamental Theory and Applications*, 43(9):769–778, 1996.

[19] A. Zulehner and R. Wille. Taking one-to-one mappings for granted: Advanced logic design of encoder circuits. In *Design, Automation and Test in Europe*, 2017.