# An Algorithm for Synthesis of Reversible Logic Circuits

Pallav Gupta, *Student Member, IEEE*, Abhinav Agrawal, and Niraj K. Jha, *Fellow, IEEE*

***Abstract*—Reversible logic finds many applications, especially in the area of quantum computing. A completely specified $n$-input, $n$-output Boolean function is called reversible if it maps each input assignment to a unique output assignment and vice versa. Logic synthesis for reversible functions differs substantially from traditional logic synthesis and is currently an active area of research. The authors present an algorithm and tool for the synthesis of reversible functions. The algorithm uses the positive-polarity Reed–Muller expansion of a reversible function to synthesize the function as a network of Toffoli gates. At each stage, candidate factors, which represent subexpressions common between the Reed–Muller expansions of multiple outputs, are explored in the order of their attractiveness. The algorithm utilizes a priority-based search tree, and heuristics are used to rapidly prune the search space. The synthesis algorithm currently targets the generalized $n$-bit Toffoli gate library. However, other algorithms exist that can convert an $n$-bit Toffoli gate into a cascade of smaller Toffoli gates. Experimental results indicate that the authors' algorithm quickly synthesizes circuits when tested on the set of all reversible functions of three variables. Furthermore, it is able to quickly synthesize all four-variable and most five-variable reversible functions that were in the test suite. The authors also present results for some benchmark functions widely discussed in literature and some new benchmarks that the authors have developed. The algorithm is shown to synthesize many, but not all, randomly generated reversible functions of as many as 16 variables with a maximum gate count of 25.**

***Index Terms*—Quantum computing, reversible computing, reversible logic synthesis.**

## I. Introduction

With the computer industry keeping pace with Moore's law, energy consumption has become the center of attention in digital circuit design after performance. Landauer's principle [1] states that any logic computation that is not reversible (i.e., information lossless) will dissipate a certain amount of energy for every bit of information lost regardless of the technology chosen for implementation. Today, this energy is small compared with other forms of heat dissipation. However, with the exponential packing of transistors on a chip, this energy is expected to play a dominant role in the next one to two decades [2].

Manuscript received May 3, 2004; revised December 16, 2004, May 31, 2005, and November 7, 2005. This work was supported by the National Science Foundation under Grant CCF-0429745. This paper was recommended by Associate Editor S. Nowick.

P. Gupta and N. K. Jha are with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA (e-mail: pgupta@princeton.edu; jha@princeton.edu).

A. Agrawal is with McKinsey and Company, New York, NY 10022 USA.

Digital Object Identifier 10.1109/TCAD.2006.871622

Computation in which there is no information loss is called reversible and the gates that perform such computation are called reversible gates. Bennett [3] showed that zero energy dissipation is possible only if a circuit contains reversible gates; hence, reversibility will be an important issue in future circuit design. Reversible logic finds many applications, especially in the area of quantum computing. It is believed that quantum computing has the ability to considerably speed up some classical problems such as factorization (using Shor's algorithm) or search (using Grover's algorithm). Quantum gates are reversible by nature [4], which provides a powerful motivation to study reversible circuits.

The problem of reversible logic synthesis is concerned with the ability to automatically generate a reversible circuit given a reversible specification. It is different from traditional Boolean logic synthesis in five major ways [5]. 1) Unlike Boolean circuits, reversible circuits have an equal number of inputs and outputs. 2) Fanout is not allowed, and the circuit structure is constrained to a cascade of reversible gates. 3) Every output of a gate that is not used in the circuit is a garbage signal. A good synthesis method minimizes the number of garbage signals. 4) The total number of constants at inputs of the gates should be kept as low as possible. 5) There are no feedback paths, and hence, the circuit is acyclic. These differences make traditional synthesis methods inapplicable.

In this paper, we present Reed–Muller reversible logic synthesizer (RMRLS), an algorithm and tool that uses the positive-polarity Reed–Muller (PPRM) expansion of a reversible function to synthesize a reversible circuit. A naive algorithm would simply use as many gates as there are terms in the Reed–Muller expansion of the function. Clearly, such a method fails to take advantage of any shared functionality that exists between multi-output functions. In our algorithm, candidate factors, which are subexpressions common between Reed–Muller expansions of multiple outputs, are identified. The factors are then substituted into the Reed–Muller expansions to determine if they will be favorable in leading to a solution (i.e., a synthesized circuit). The primary objective of the algorithm is to minimize the number of gates (i.e., the number of factors needed to convert a Reed–Muller expansion into the identity function), whereas its secondary objective is to minimize the size of the individual gates (i.e., the number of literals in the factors).

The remainder of this paper is organized as follows. Section II presents background material that is required to understand the ideas presented in this paper, whereas previous work is discussed in Section III. The synthesis algorithm is

| $c$ | $b$ | $a$ | $c_o$ | $b_o$ | $a_o$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 |

Fig. 1. Reversible function of three variables.

| $c$ | $b$ | $a$ | $c_o$ | $s_o$ | $p_o$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | $1^\dagger$ |
| 0 | 1 | 0 | 0 | 1 | $1^\dagger$ |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | $1^\dagger$ |
| 1 | 1 | 0 | 1 | 0 | $1^\dagger$ |
| 1 | 1 | 1 | 1 | 1 | 0 |

(a)

| $d$ | $c$ | $b$ | $a$ | $c_o$ | $s_o$ | $p_o$ | $g_o$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |

(b)

Fig. 2. Augmented full-adder. (a) Original truth table. (b) Possible reversible specification.

described in detail in Section IV. Our experimental results are presented in Section V. The conclusions are given in Section VI.

## II. Background

We present some preliminary concepts in this section.

### A. Reversible Functions

A completely specified $n$-input, $n$-output Boolean function is reversible if it maps each input assignment to a unique output assignment and vice versa [6]. A reversible function of $n$ variables can be defined as a truth table or as a permutation on the set of integers $\{0, 1, \ldots, 2^n - 1\}$. For example, the reversible function in Fig. 1 can also be specified as {1, 0, 7, 2, 3, 4, 5, 6}. An irreversible function can be converted into a reversible function by adding extra outputs, called garbage outputs, such that the input–output mapping is unique. If the maximum number of identical output vectors is $p$, then the total number of garbage outputs needed is equal to $\lceil \log_2 p \rceil$ [2]. Of course, constant garbage inputs must then be added, as necessary, to balance the number of inputs and outputs.

To demonstrate the process of converting an irreversible function into a reversible one, consider the augmented full-adder, which produces carry ($c_o$), sum ($s_o$), and propagate ($p_o$) signals. Its truth table is shown in Fig. 2(a). The function is not reversible because there exist repeated output vectors (marked with † in the figure). An extra garbage output $g_o$ set equal to input $a$ or $b$ will make the mapping unique. A garbage input $d$ must also be added to make the number of inputs and outputs equal. Fig. 2(b) shows one possible reversible specification of the augmented full-adder [7].

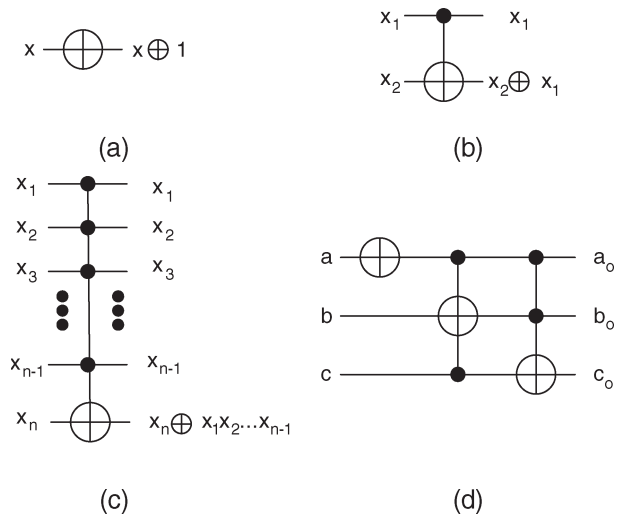


Fig. 3. (a) NOT gate, (b) CNOT gate, (c) $n$-bit Toffoli gate, and (d) circuit for function in Fig. 1.

### B. Reversible Gates

A reversible gate implements a reversible function. There are two main types of reversible gates, namely: 1) Toffoli [8] and 2) Fredkin [9]. Since our algorithm currently does not utilize the Fredkin gate, we do not discuss it further in this paper. An $n$-bit Toffoli gate, denoted by TOF$n(x_1, x_2, \ldots, x_n)$, passes the first $n-1$ inputs (referred to as control bits) to the output unaltered and inverts the $n$th input (referred to as target bit) if the first $n-1$ inputs are all one. That is

$$\begin{aligned} y_i &= x_i, \quad \text{for } 1 \le i < n \\ y_n &= x_n \oplus x_1 x_2 \cdots x_{n-1}. \end{aligned} \tag{1}$$

A one-bit Toffoli gate inverts the input unconditionally and is called the NOT gate. A two-bit Toffoli gate is called the Feynman or CNOT gate. Fig. 3 presents a graphical representation of NOT, CNOT, and $n$-bit Toffoli gates and the circuit implementation of the reversible function in Fig. 1.

### C. Reed–Muller Expansions

Any Boolean function can be described using an EXOR sum-of-products (ESOP) expansion [10], [11]. The PPRM expansion uses only uncomplemented variables and can be derived from the function's sum-of-products expression. The PPRM expansion of a function is canonical and of the form

$$\begin{aligned} f(x_1, x_2, \ldots, x_n) = {} & a_0 \oplus a_1 x_1 \oplus \cdots \oplus a_n x_n \oplus a_{12} x_1 x_2 \\ & \oplus a_{13} x_1 x_3 \cdots \oplus a_{n-1,n} x_{n-1} x_n \\ & \oplus \cdots \oplus a_{12\cdots n} x_1 x_2 \cdots x_n \end{aligned} \tag{2}$$

where $a_i \in \{0, 1\}$ and $x_i$ are all uncomplemented (positive polarity). For example, the PPRM expansion of the function in Fig. 1 is as follows:

$$\begin{aligned} a_o &= a \oplus 1 \\ b_o &= b \oplus c \oplus ac \\ c_o &= b \oplus ab \oplus ac. \end{aligned} \tag{3}$$

### D. Quantum Cost

The quantum cost of a reversible circuit is the sum of the quantum cost of its gates. The quantum cost of a gate $G$ is the number of elementary quantum operations required to realize the function given by $G$ [2]. These elementary operations are performed by the NOT, CNOT, and three-bit Toffoli gates. NOT and CNOT gates have a quantum cost of one. However, they are not complete because they only realize linear functions. The addition of the three-bit Toffoli gate makes the set of gates complete (i.e., have the ability to implement any reversible function). However, the three-bit Toffoli gate cannot be realized as a single elementary operation. Fortunately, a realization for the three-bit Toffoli gate with a quantum cost of five has been found [12]. Larger Toffoli gates have a higher quantum cost due to the number of elementary quantum operations required for their realizations. We use the cost table available from [13] to calculate the cost for such gates.

Given the current state of technologies for implementing quantum circuits, it is believed that an $n$-bit Toffoli $(n > 3)$ gate will have a high technological cost. These gates are expected to be macros that will be implemented by elementary gates. Theoretical lower and upper bounds on the number of elementary gates required to implement an $n$-bit Toffoli gate are given in [12] and [14].

### E. Generating PPRM Expansions

Generating the PPRM expansion of a Boolean function is nontrivial. There are two issues that complicate the process. First, most functions encountered in practice are irreversible and/or incompletely specified. To make a function reversible, it is necessary to add garbage inputs and outputs and assign bit values (i.e., 0/1) to them. Determining which combination, among the possible assignments, will lead to the best synthesized circuit, given the optimization criteria (e.g., least number of Toffoli gates, minimum quantum cost, etc.), is a challenging and open problem. The second problem is that it is necessary to convert a reversible function into the ESOP form before it can be expanded into the PPRM form. Fortunately, this problem has been addressed, and researchers have developed a state-of-the-art tool called EXORCISM-4 [15] that uses efficient heuristics and look-ahead strategies to quickly find the ESOP form of a Boolean function. Once the ESOP form has been obtained, it can be transformed into the PPRM form by making the substitution $\bar{a} = a \oplus 1$ on all the complemented variables, algebraically expanding the product terms, and canceling out an even number of identical product terms.

In this paper, the PPRM expansion of a function was obtained in one of the following ways.

1) Completely specified reversible functions: EXORCISM-4 was used to convert the specification into the ESOP form, which was then converted into the PPRM form.
2) Benchmark functions: For the benchmarks for which the specification was incomplete, the PPRM expansion was derived manually by examining the synthesized circuits reported by other researchers [13]. For the benchmarks that we have developed, we provide the complete specification.

## III. Previous Work

In this section, we present a survey of some of the other reversible logic synthesis algorithms that have been proposed.

An exhaustive algorithm is presented in [16], which generates all possible circuits containing $k$ gates for increasing values of $k$ until a circuit is found that implements the given specification. This is an iterative-deepening approach, and it can be seen that the result will be optimal. However, the applicability of this method is restricted to reversible functions of at most three or four variables that require eight or fewer gates in their implementation. Important theoretical results on the synthesis properties of even and odd permutation functions are also presented. In [17], the authors introduce local optimization (similar to peephole optimization in compilers) for reversible circuits where suboptimal subcircuits are replaced with smaller counterparts to simplify and reduce the overall size of the circuit.

An algorithm based on the Rademacher–Walsh spectrum is presented in [18]. At any given stage, the circuit is synthesized from inputs to outputs or vice versa depending upon the best translation (i.e., an application of a generalized $n$-bit Toffoli (GT) gate) that is possible. The best translation is determined to be that which results in the maximum positive change in the complexity measure of the function. Because there is no backtracking or look-ahead, an error is declared if no translation can be found. The authors are working on a formal proof to show that the algorithm will always synthesize a valid circuit given enough time and memory. Their experimental results indicate that the method holds promise and needs to be further investigated.

In [5], the authors present an iterative algorithm to implement an incompletely specified Boolean function as a cascade of reversible complex Maitra terms (also known as reversible wave cascades). The remarkable feature about this algorithm is that it requires at most one constant input and no garbage outputs. The basic idea is that a cascade implementation of the original incompletely specified function is equivalent to the cascade implementation of a completely specified function that has the same ON-set (minterms) and OFF-set (maxterms) as the original function. First, the set of completely specified functions representing a stage of a cascade is computed. Next, the remainder function is calculated assuming that the completely specified functions will be used in the cascade. The stages are added to the synthesized circuit if the remainder function is independent of at least one (or more) variable(s).

A bidirectional algorithm is described in [7], in which synthesis proceeds by adding gates to the circuit either at its inputs or outputs. Synthesis is complete when the reversible specification has been transformed into the identity function (i.e., $a_{\text{out}} = a$, $b_{\text{out}} = b$, $c_{\text{out}} = c$, ...). This is achieved by inspecting the truth table in lexicographical order until the first output assignment is encountered, which is not equal to the input assignment located in the same row of the table. A series of Toffoli and/or Fredkin gates must be added in such a way that the output assignment becomes the same as the input assignment. However, mapping each output back to its corresponding input is often not the best mapping. An output permutation, which maps output $v_{\text{out}}$ to a different input rather than its corresponding input $v$ (e.g., $a_{\text{out}} = b$, $b_{\text{out}} = a$, $c_{\text{out}} = c$, ...), is used to find a better mapping. An output permutation is useful because it may lead to a simpler specification that needs to be synthesized. The interesting feature about this algorithm is that it is guaranteed to synthesize a valid circuit given enough time and memory. However, the method synthesizes circuits that frequently contain sequences of gates that can be simplified. Thus, a processing step is applied postsynthesis where such sequences, called templates, are identified and simplified. Templates were first introduced in [19] and later generalized in [20]–[22].

Finally, the algorithm in [6] has the ability to utilize any arbitrary gate library (i.e., arbitrary subsets of NOT, CNOT, $n$-bit Toffoli, SWAP, and $n$-bit Fredkin gates) during synthesis. The main idea is that given a reversible specification, all gates in a given library are possible candidates for application. For each gate, assuming it is used, the remainder function is calculated using shared binary decision diagrams with complementary edges. The complexity measure of the remainder function is calculated, and the gate that results in the lowest complexity measure is chosen to be added to the synthesized circuit. Synthesis then proceeds on the remainder function. If two or more gates yield the lowest complexity, multiple paths are explored simultaneously.

## IV. Synthesis Algorithm

We describe our synthesis algorithm in this section. We describe the basic algorithm first. We then describe some heuristics that are used to improve the performance of the basic algorithm. The input to our algorithm is a PPRM expansion of a reversible function $f(v_1, v_2, v_3, \ldots, v_n)$ that is to be synthesized. The output is a network of Toffoli gates that realizes $f$. We illustrate the application of our algorithm with an example and comment on the data structures that we use and whether the algorithm converges or not.

### A. Algorithm

Fig. 4 outlines the pseudocode for the main steps in the basic version of our synthesis algorithm. Initialization takes place in lines 1–13. Variable $bestDepth$, which stores the number of gates in the best circuit synthesized for $f$ so far, is set to infinity. Variable $bestSolNode$ stores a pointer to a leaf node, which represents the last gate of the synthesized circuit. The total number of terms in the PPRM expansion of $f$ is stored in $initTerms$, and a timer $(Timer)$ is created that specifies the time limit for synthesis.

The root node $(rootNode)$ of the search tree is initialized next. The depth and factor of this node are set to 0 and NULL, respectively. The PPRM expansions of all output variables $v_{\text{out},i}$ in $f$ are obtained in terms of all its input variables $v_i$ and stored in $rootNode.pprm$. $rootNode.terms$ and $rootNode.elim$ contain the total number of terms in the current PPRM expansion and the total number of terms that have been eliminated from the original PPRM expansion once a substitution is made, respectively. Because this is the root node, $rootNode.elim$ is set to zero. In addition, its priority $rootNode.priority$ is set to infinity. Finally, an empty priority queue $PQ$ is initialized, and the root node is pushed onto the priority queue. This will be the first node that will be explored during synthesis. The priority queue maintains a list of nodes sorted with respect to their priorities.

After the initialization phase, the algorithm enters a loop. The most promising node for further exploration is removed from the priority queue and stored in $parentNode$ (line 15). Lines 16 and 17 check to see whether $parentNode$ is worth exploring or not. If the depth of $parentNode$ is greater than or equal to $bestDepth - 1$, then $parentNode$ can be ignored as it cannot possibly lead to a better solution than the best one seen so far.

We explore $parentNode$ by examining each output variable $v_{\text{out},i}$ in the PPRM expansion of $f$ (lines 18 and 19). For each input variable $v_i$, we search for factors in the PPRM expansion of $v_{\text{out},i}$ contained in $parentNode.pprm$ that do not contain $v_i$ (line 20). For example, if $a_{\text{out}} = a \oplus 1 \oplus bc \oplus ac$, then the appropriate factors are 1 and $bc$, as neither contains literal $a$. For each factor $(factor)$ that has been identified in this manner, the substitution $v_i = v_i \oplus factor$ is made in the PPRM expansions of $parentNode$. A new node is created, which is a child of $parentNode$. The depth of the child node is incremented by one (line 23), and a copy of $factor$ is stored (line 24). The PPRM of the child node is set to be the PPRM obtained once the substitution has been made (line 25). The number of terms in the new PPRM and the number of terms eliminated by making the substitution are stored in $childNode.terms$ and $childNode.elim$, respectively (lines 26 and 27). Finally, $childNode$ is analyzed, and one of the following actions is taken.

1) If the synthesis of $f$ has been completed (i.e., the PPRM expansions for all $v_{\text{out},i}$ contain only $v_i$), then the values of $bestDepth$ and $bestSolNode$ are updated if this solution improves upon the best solution found so far (lines 28–30).
2) If the number of terms in the PPRM expansion has not decreased by making the substitution (i.e., $childNode.elim \leq 0$), then the node is also disregarded (line 31). This guarantees that we only explore those nodes where the number of terms in the PPRM expansion is decreasing monotonically with the application of each substitution. Otherwise, the priority of $childNode$ is calculated, and the node is inserted into the priority queue.

**Require:** PPRM expansion of function $f(v_1, v_2, v_3, \ldots, v_n)$ which is to be synthesized
// initialize depth of best solution, pointer to best solution
$bestDepth \leftarrow \infty$
$bestSolNode \leftarrow$ NULL
$initTerms \leftarrow$ number of terms in the PPRM expansion of $f(v_1, v_2, v_3, \ldots, v_n)$
$Timer \leftarrow$ new Timer with a pre-specified time limit

// create root node of search tree and initialize it
$rootNode \leftarrow$ new node with no parent
$rootNode.depth \leftarrow 0$
$rootNode.factor \leftarrow$ NULL
$rootNode.pprm \leftarrow$ PPRM expansion of $f(v_1, v_2, v_3, \ldots, v_n)$
$rootNode.terms \leftarrow initTerms$
$rootNode.elim \leftarrow initTerms - rootNode.terms$
$rootNode.priority \leftarrow \infty$

// create priority queue and push the root node on the priority queue
$PQ \leftarrow$ initialize empty priority queue
$PQ.push(rootNode, rootNode.priority)$

**repeat**
  $parentNode \leftarrow PQ.pop()$
  **if** $parentNode.depth \geq bestDepth - 1$ **then**
    continue

  // for each output variable in the PPRM expansion of the node that is being explored
  **for all** $v_i$ in $v_{out,i}$ in $parentNode.pprm$ **do**
    $v_i.pprm \leftarrow parentNode.pprm.expansion(v_i)$

    // consider all factors in the PPRM expansion of $v_{out,i}$ that do not contain $v_i$
    **for all** $factor$ in $v_i.pprm$ **do**
      **if** $factor$ does not contain variable $v_i$ **then**
        $childNode \leftarrow$ new node with $parentNode$ as its parent
        $childNode.depth \leftarrow parentNode.depth + 1$
        $childNode.factor \leftarrow v_i = v_i \oplus factor$
        $childNode.pprm \leftarrow$ substitute $v_i = v_i \oplus factor$ in all PPRM expansions of
                all output variables $v_{out}$ that contain $v_i$ in $parentNode.pprm$
        $childNode.terms \leftarrow$ number of terms in the PPRM expansion of $childNode.pprm$
        $childNode.elim \leftarrow parentNode.terms - childNode.terms$

        // if we have found a better solution, update the best solution
        **if** $childNode$ is a solution && $childNode.depth < bestDepth$ **then**
          $bestDepth \leftarrow childNode.depth$
          $bestSolNode \leftarrow childNode$

        // if this node is a candidate for further exploration, calculate its priority, and insert in priority queue
        **if** $childNode.elim > 0$ **then**
          $childNode.priority \leftarrow \alpha * childNode.depth + \frac{\beta * childNode.elim}{childNode.depth} - \gamma * factor.literalCount$
          $PQ.push(childNode, childNode.priority)$
  // while there are candidates in the priority queue and timer has not expired
**until** ($PQ.isEmpty()$ == FALSE && $Timer.isExpired()$ == FALSE)

// the path from $rootNode$ to $bestSolNode$ represents the synthesized network
// for each node $n$ along the path, $n.factor$ contains the actual Toffoli gate (i.e. the substitution that was made)
**return** $bestSolNode$

Fig. 4. Pseudocode for our basic reversible logic synthesis algorithm.

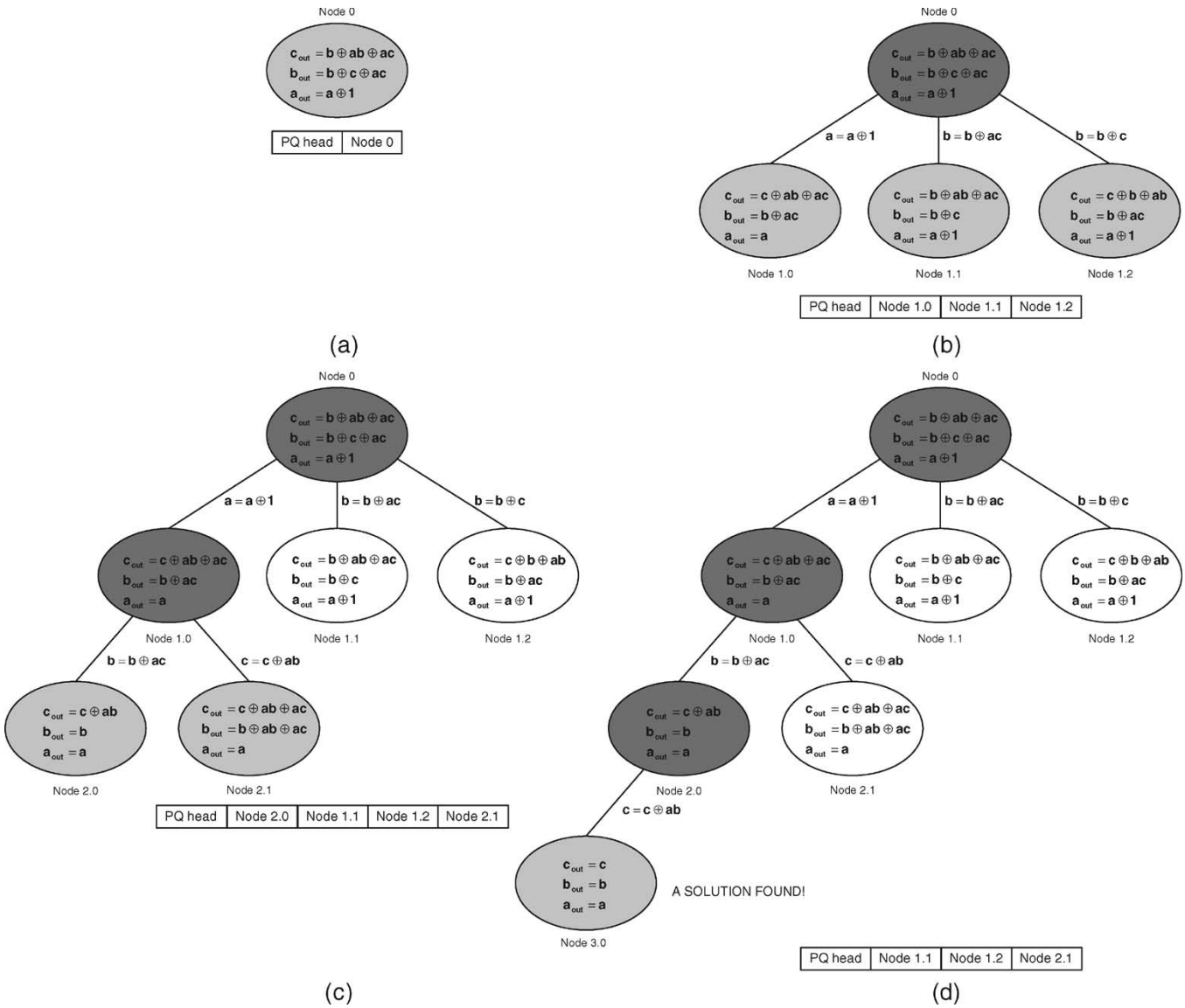


Fig. 5. Application of algorithm to reversible function in Fig. 1.

The priority of $childNode$ is calculated as follows:

$$childNode.priority = \alpha * childNode.depth + \frac{\beta * childNode.elim}{childNode.depth} - \gamma * factor.literalCount \quad (4)$$

where $\alpha$, $\beta$, and $\gamma$ are weights that sum up to one. In (4), the first term gives preference to nodes at a larger depth (i.e., depth-first search) as all things being equal, they are more likely to be closer to a solution. The second term addresses the primary objective of minimizing the number of Toffoli gates. The number of terms eliminated per stage is used to measure a node's effectiveness. Finally, the third term addresses the secondary objective of minimizing the number of control bits of the individual Toffoli gates. After careful experimentation, values of 0.3, 0.6, and 0.1 for $\alpha$, $\beta$, and $\gamma$, respectively, were used in this paper.

The algorithm repeats the above process until the priority queue becomes empty or the timer expires. The former condition implies that there are no more candidate nodes left to explore. The latter condition states that we have reached the time limit for synthesis. Upon termination, $bestSolNode$ contains a pointer to the leaf node, which represents the last gate of the synthesized circuit. The path from $rootNode$ of the search tree to $bestSolNode$ represents the series of Toffoli gates in the synthesized circuit. The edges of the path represent the substitutions that were made. For each node $n$ along this path, $n.factor$ contains a copy of the substitution $v_i = v_i \oplus factor$. Hence, $v_i$ is the target bit, and the literals in $factor$ represent the control bits of the Toffoli gate.

### B. Example

Fig. 5 illustrates the application of our basic synthesis algorithm to the reversible function in Fig. 1. In the figure, the darkly shaded nodes have already been explored, lightly shaded nodes have been added in the current stage, and nodes with no shading are yet to be considered. Initially, the PPRM expansion of the function is stored in $Node\ 0$, which is inserted into the priority queue. Fig. 5(a) shows the search tree and priority

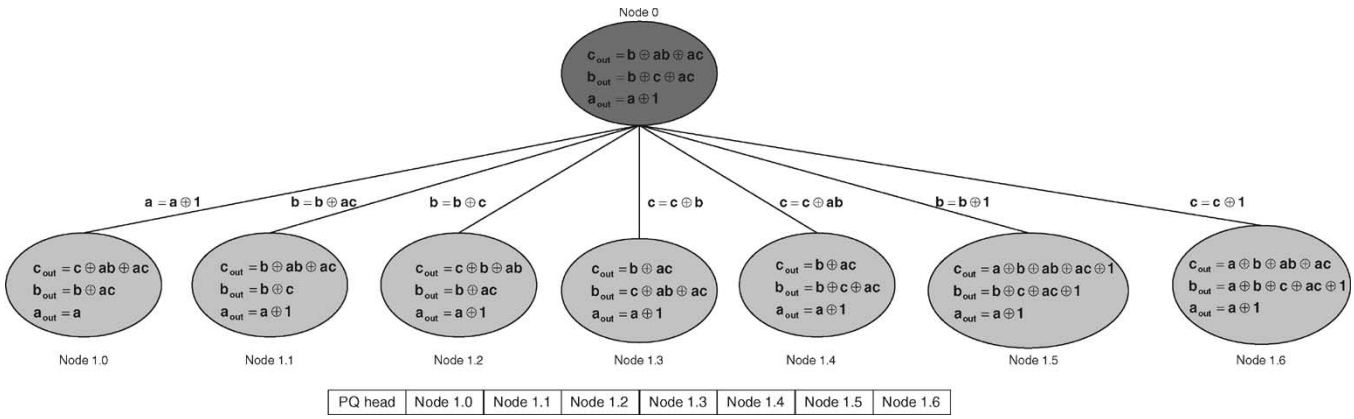


Fig. 6. Initial search tree expanded to contain additional substitutions.

queue at this point. In the next step, $Node\ 0$ is popped from the priority queue (it is the only item present) and examined for possible substitutions. The algorithm identifies three possible substitutions, namely: 1) $a = a \oplus 1$; 2) $b = b \oplus c$; and 3) $b = b \oplus ac$. For each substitution, a new node is created, the factor that is identified is substituted in the PPRM expansion to get the new PPRM expansion. Because all three substitutions result in fewer terms in the new PPRM expansion, all the nodes are added to the priority queue. $Node\ 1.0$ has a higher priority than $Node\ 1.1$ and $Node\ 1.2$ [see Fig. 5(b)]. In the next step, $Node\ 1.0$ is popped from the priority queue and analyzed. $b = b \oplus ac$ and $c = c \oplus ab$ are the two possible substitutions identified for this node. $Node\ 2.0$ and $Node\ 2.1$ corresponding to these substitutions are inserted into the priority queue with $Node\ 2.0$ at the head of the queue with the highest priority [see Fig. 5(c)]. In the next iteration, $Node\ 2.0$ is popped from the priority queue. The only possible substitution is $c = c \oplus ab$, which leads to a solution [see Fig. 5(d)]. The solution (i.e., the substitutions shown on the path from the root of the search tree to $Node\ 3.0$) and its depth (i.e., the length of the path) are cached. At this point, no new nodes are inserted into the priority queue. $Nodes\ 1.1$ and $1.2$ are popped in that order, respectively, but fail to give better solutions from any of their substitutions. Their children are not added to the queue because we have already found a solution of depth three, which they will not be able to beat. Lastly, $Node\ 2.1$ is popped but discarded because its depth is too large to be useful. The solution illustrated in Fig. 3(d) is the best solution that the algorithm finds.

### C. Data Structures

The data structures that are used in our algorithm are well known. A priority queue, implemented as a max heap, is utilized to determine which node is processed next. Doubly linked lists are used to store the PPRM expansion of a function, and substitutions are made by traversing the list. The way the algorithm is implemented requires repetitive searching in the forward direction of the linked list only. To speed up this process, a pointer is saved at the last match, and the search continues from this point onward the next time. A tree data structure is utilized to keep track of the search space. Because leaf nodes represent the only candidates for further exploration, we do not need to store the PPRM expansion at the intermediate nodes. We only need to store the substitution that was made in the nonleaf nodes. This memory optimization is very useful when synthesizing larger functions.

### D. Additional Substitutions

So far, the substitutions that we have considered are of the form $v_i = v_i \oplus factor$, which require variable $v_i$ to be present in the PPRM expansion for the corresponding output variable $v_{\text{out},i}$. This requirement is somewhat stringent and needs to be relaxed to be able to synthesize functions of larger variables. Enlarging the set of substitutions presents a drawback in that more substitutions need to be considered when exploring a node. However, it also increases the likelihood of good substitutions being found that eliminate more terms in the PPRM expansion, thus leading to a solution quicker. We allow for two additional types of substitutions.

- We no longer require that output variable $v_{\text{out},i}$ contains variable $v_i$ for the algorithm to consider factors from the PPRM expansion of $v_{\text{out},i}$. For example, in the synthesis for the reversible function in Fig. 1 just presented, the algorithm would also select $c = c \oplus b$ and $c = c \oplus ab$ as possible substitutions in the first stage.
- For any variable $v_i$, we also allow the substitution $v_i = v_i \oplus 1$ even if the PPRM expansion of $v_{\text{out},i}$ does not contain 1. In the synthesis of the reversible function in Fig. 1, substitutions $b = b \oplus 1$ and $c = c \oplus 1$ would be added as possible substitutions in the first stage. For this special substitution only, we make an exception in that we allow the number of terms in the PPRM expansion to increase.

Fig. 6 shows what the initial search tree would look like if the additional substitutions identified above are also considered.

### E. Heuristics for Basic Algorithm

Given enough time and memory, the basic algorithm in Fig. 4 will always find a valid solution. We prove this claim in the next section. In practice, however, the applicability of the basic algorithm is limited to reversible functions of at most five variables before it exceeds the available memory. Consequently, we

introduce a few heuristics to the basic algorithm to improve its performance on reversible functions with a much larger number of variables. The price we pay is that these heuristics can no longer guarantee that a solution will be found if one exists.

We abandon the entire search process and backtrack to the first level of the search tree if we have not been able to find a solution in a user-specified number (e.g., $\sim$ 10 000) of steps. The intuition behind this is that if we made a poor substitution, chances are that we might have made such a substitution very early (i.e., at higher depths of the search tree) during synthesis. Given that we have not been able to find a solution after considerable effort, we choose to abandon the search and restart the search from the top of the search tree with a different substitution (i.e., an alternative path).

There are many substitutions that need to be considered at each level of the search tree. Consequently, the branching factor in the basic algorithm will be large for larger functions. To resolve this issue, we apply the greedy method in that only the best substitution or the best $k$ substitutions with the highest score are added to the priority queue at every iteration for each input variable $v_i$. The score of each substitution is given by its priority, which is calculated using (4). If there are $n$ input variables, then either $n$ or $kn$ factors are added to the priority queue at each level of the search tree. The value of $k$ used in our heuristics varies from three to five. This heuristic considerably reduces the memory requirements, speeds up the search, and enables the basic algorithm to handle large functions.

### F. Algorithm Convergence

The remaining question that needs to be addressed is whether the algorithm will always synthesize a valid circuit, i.e., terminate with a valid solution. We claim that this is indeed the case for the basic algorithm, and the proof is given next.

Given a PPRM expansion, a solution is found if and only if for each input variable $v_i$, $v_{\text{out},i} = v_i$. That is the PPRM expansion, through repeated applications of substitutions, has been reduced to the identity function. The substitutions that are applied are of the form $v_i = v_i \oplus factor$, where $v_i$ is the target bit of a Toffoli gate and the literals in $factor$ are the control bits of the Toffoli gate. Note that the literals in $factor$ cannot contain $v_i$ (i.e., the substitution $v_i = v_i \oplus v_i \cdot factor$ is illegal) because $v_i$ cannot both be the target and control bit in a Toffoli gate.

The basic algorithm allows for three types of substitutions:

1) $v_i = v_i \oplus factor$, where $v_i$ *is* present in the PPRM expansion of output variable $v_{\text{out},i}$;
2) $v_i = v_i \oplus factor$, where $v_i$ *is not* present in the PPRM expansion of output variable $v_{\text{out},i}$;
3) $v_i = v_i \oplus 1$.

The union of these types of substitutions are *all* the possible substitutions that can be made in a PPRM expansion. There are no other valid substitutions that can be applied. Consequently, the algorithm considers all the possible substitutions that can be applied at each stage. All of these candidates will be stored in the priority queue. Given enough time and memory, each of the possible substitutions will eventually be analyzed. Although this occurs at every stage, in the worst case scenario, the entire search space will be searched. Therefore, the algorithm will always find a valid solution. This concludes the proof.

As mentioned in the previous section, heuristics must be utilized to improve the performance of the basic algorithm on larger examples. The heuristics choose a subset of the list of possible substitutions based on their order of attractiveness. Depending upon the heuristic, it is possible to miss making a critical substitution that is required to find a solution for a given example. Because the heuristics described above are greedy in nature, they may prune such a critical substitution because other substitutions may have higher priorities than the one that is pruned. Consequently, if the basic algorithm is used in conjunction with the heuristics described above, there is no guarantee that a valid solution will be found.

## V. Experimental Results

In this section, we present our experimental results. The synthesis algorithm has been implemented in C and is a part of a tool called RMRLS [23]. All experiments were conducted on a Dell PowerEdge 600SC server featuring a Pentium IV 1.6-GHz processor, 512-kB cache, 768-MB RAM, and running RedHat Linux 8.0.

### A. Three-Variable Reversible Functions

We first synthesized all reversible functions of three variables, which number 8! or 40 320. Although our algorithm targets the GT gate library, circuits of three variables can contain gates from the NCT library, which contains the NOT, CNOT, and three-bit Toffoli gates only. Consequently, we take the liberty of saying that we are using the NCT library for this case only. Table I compares our results with those reported by other researchers. The column "No. gates" shows the number of gates in the circuit, whereas the data in other columns indicate the number of such circuits that were synthesized. The synthesis methods in [6] and [7] utilize the more general NCTS library, which, in addition to the gates in the NCT library, contains a SWAP gate. A SWAP gate unconditionally exchanges a pair of inputs. Optimal results for three-variable reversible functions, using both the NCT and NCTS libraries, are from [16]. As can be seen from Table I, our algorithm synthesizes good quality circuits for the case of three-variable functions. Furthermore, it took less than half a second to synthesize each function.

By extending our algorithm to include the SWAP gate, it may be possible to achieve results comparable to those in [6]. The reason why our algorithm achieves better results than [7] despite the lack of a SWAP gate is mainly because substitution of common subexpressions in a PPRM expansion seems to be a more powerful technique in reducing a reversible function to the identity function than trying to map each output back to its corresponding input. Of course, templates [20]–[22] provide a very useful method for improving the synthesized circuit further and should be utilized as a postprocessing step by any synthesis algorithm. In fact, it was reported in a personal communication [24] that the average circuit size was improved from 6.10 to 6.05 when our results were postprocessed through a template-based simplification tool [21]. However, because we

TABLE I
ALL REVERSIBLE FUNCTIONS OF THREE VARIABLES

| No. gates | Ours NCT | Miller [7] NCTS | Kerntopf [6] NCTS | Optimal [16] NCT | Optimal [16] NCTS |
|---|---|---|---|---|---|
| 11 | | 5 | | | |
| 10 | | 110 | | | |
| 9 | 36 | 792 | 86 | | |
| 8 | 3,351 | 4,726 | 2,740 | 577 | 32 |
| 7 | 12,476 | 11,199 | 11,774 | 10,253 | 6,817 |
| 6 | 13,596 | 12,076 | 13,683 | 17,049 | 17,531 |
| 5 | 7,479 | 7,518 | 8,068 | 8,921 | 11,194 |
| 4 | 2,642 | 2,981 | 3,038 | 2,780 | 3,752 |
| 3 | 625 | 767 | 781 | 625 | 844 |
| 2 | 102 | 130 | 134 | 102 | 134 |
| 1 | 12 | 15 | 15 | 12 | 15 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| Avg. | 6.10 | 6.18 | 6.01 | 5.87 | 5.63 |

do not have access to this tool, the other results we report do not have the benefit of this postprocessing step.

### B. Four- and Five-Variable Reversible Functions

We attempted to synthesize completely specified four- and five-variable reversible functions next. We generated a sample of 50 000 and 3000 random PPRM specifications of four and five variables, respectively. This was done because synthesizing all possible functions was beyond reach (there are 16! and 32! reversible functions of four and five variables, respectively). The following synthesis options were specified.

1) For four variables, a time limit of 60 s per function, maximum circuit size of 40 gates, and the greedy option for substitution pruning was set.
2) For five variables, a time limit of 180 s per function, maximum circuit size of 60 gates, and the greedy option for substitution pruning was set.

The synthesis results are shown in Tables II and III. Here, circuit size refers to the number of Toffoli gates. The majority of four-variable functions could be synthesized in 5–20 s, whereas some took longer. For five-variable functions, the majority of functions that could be synthesized took 40–75 s.

We observe that the algorithm successfully synthesized all four-variable functions in our test suite. However, this was not true for five-variable functions, where 194 examples (6.5%) failed to synthesize. We believe this could be due to many reasons. First, the algorithm might have needed a longer time to synthesize the functions. Second, it could be that these functions require more than 60 gates. Since these were randomly generated functions, we had no way of knowing *a priori* the number of gates needed for their implementation. However, if we look at Table III, we see that the number of gates required for the majority of synthesized functions is in the 30–45 range. Thus, it seems that the heuristics in our algorithm are not able to prune the search space well enough to handle five-variable functions, which require more than 45 gates. The search space at a depth of 45 is $s^{45}$, where $s$ is the number of substitutions possible. At every node, $s$ can be in the range 0–20. This is a huge search space. To be able to handle examples that require more than 45 gates, it seems to be necessary to improve upon the current heuristic of choosing which substitutions to explore further. It should be mentioned that the algorithm can still synthesize some functions that require more than 45 gates and also sometimes fail to synthesize some functions that require fewer than 30 gates.

### C. Examples From Literature and New Benchmarks

We now consider several examples from the literature and introduce some new ones for the first time here. A time limit of 60 s and the greedy option for substitution pruning was specified for synthesis. Some of the examples were synthesized in a fraction of a second, others took a few seconds, and the remaining, namely, *rd53*, *alu*, and the shifters took longer. However, all examples were successfully synthesized within the specified time limit. The first two examples are from [7].

*Example 1:*

Specification: {1, 0, 3, 2, 5, 7, 4, 6}.
Toffoli circuit: TOF3$(c, a, b)$ TOF3$(c, b, a)$ TOF3$(c, a, b)$ TOF1$(a)$.

This means that there are four Toffoli gates cascaded together, as shown in Fig. 7.

*Example 2:* This represents a wraparound shift to the right of one position for a three-variable function.

Specification: {7, 0, 1, 2, 3, 4, 5, 6}.
Toffoli circuit: TOF1$(a)$ TOF2$(a, b)$ TOF3 $(b, a, c)$.

The next series of examples is from [18].

*Example 3:* This represents the realization of a Fredkin gate using Toffoli gates.

Specification: {0, 1, 2, 3, 4, 6, 5, 7}.
Toffoli circuit: TOF3$(c, a, b)$ TOF3$(c, b, a)$ TOF3$(c, a, b)$.

*Example 4:* This represents a simple swap between two positions in a three-variable function.

Specification: {0, 1, 2, 4, 3, 5, 6, 7}.
Toffoli circuit: TOF2$(c, b)$ TOF3$(c, b, a)$ TOF3$(b, a, c)$ TOF3$(c, b, a)$ TOF2$(c, b)$.

*Example 5:* This is an extension of Example 4 to four variables.

Specification: {0, 1, 2, 3, 4, 5, 6, 8, 7, 9, 10, 11, 12, 13,14, 15}.
Toffoli circuit: TOF2$(d, b)$ TOF3$(d, b, a)$ TOF4$(d, b, a, c)$ TOF4$(c, b, a, d)$ TOF4$(d, b, a, c)$ TOF3$(d, b, a)$ TOF2$(d, b)$.

*Example 6:* This represents a wraparound shift to the left of one position for a three-variable function.

Specification: {1, 2, 3, 4, 5, 6, 7, 0}.
Toffoli circuit: TOF3$(b, a, c)$ TOF2$(a, b)$ TOF1$(a)$.

*Example 7:* This represents a wraparound shift to the left of one position for a four-variable function.

Specification: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,15, 0}.
Toffoli circuit: TOF4$(c, b, a, d)$ TOF3$(b, a, c)$ TOF2$(a, b)$ TOF1$(a)$.

TABLE II
RANDOM FOUR-VARIABLE REVERSIBLE FUNCTIONS

| Circuit size | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| No. of circuits | 3 | 34 | 159 | 604 | 1,753 | 3,917 | 6,726 | 8,704 | 9,053 | 7,665 | 5,435 | 3,225 | 1,631 | 728 | 264 | 77 | 20 | 1 | 1 |

**Note:** All 50,000 functions could be synthesized.

TABLE III
RANDOM FIVE-VARIABLE REVERSIBLE FUNCTIONS

| Circuit size | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 51 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| No. of circuits | 1 | 3 | 8 | 29 | 45 | 82 | 130 | 202 | 206 | 310 | 344 | 307 | 304 | 297 | 176 | 151 | 117 | 47 | 27 | 15 | 4 | 1 |

**Note:** 194 (6.5%) out of 3,000 functions failed to synthesize within the given time limit.

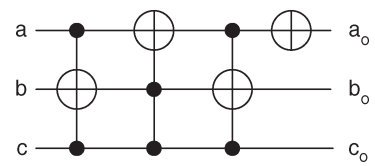


Fig. 7. Example 1 realization.

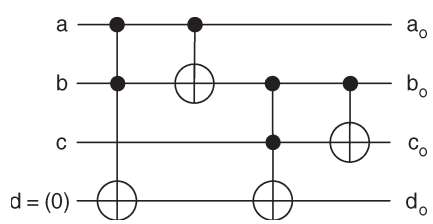


Fig. 8. Augmented full-adder realization.

*Example 8:* This represents the augmented full-adder discussed in Section II.

Specification: {0, 7, 6, 9, 4, 11, 10, 13, 8, 15, 14, 1, 12, 3, 2, 5}.
Toffoli circuit: TOF3$(b, a, d)$ TOF2$(a, b)$ TOF3$(c, b, d)$ TOF2$(b, c)$.

The circuit implementation is shown in Fig. 8.

*Example 9:* The *rd53* example is from the MCNC [25] benchmark suite and has five inputs and three outputs. The output vector is the binary encoding of the number of ones in the input vector. Thus, $\{00000\}$ yields $\{000\}$, $\{00101\}$ yields $\{010\}$, and $\{11111\}$ yields $\{101\}$. Output vectors $\{010\}$ and $\{011\}$ both occur ten times, and hence, we need to add $\lceil \log_2 10 \rceil = 4$ garbage outputs for a total of seven outputs. This requires the addition of two more constant inputs to the five existing ones.

Specification: We use the same specification as that in [18].
Toffoli circuit: TOF3$(a, b, f)$ TOF2$(b, a)$ TOF3$(a, c, f)$ TOF2$(c, a)$ TOF5$(a, b, c, d, g)$ TOF3$(a, d, f)$ TOF2$(a, d)$ TOF4$(b, d, e, g)$ TOF2$(c, b)$ TOF3$(d, e, f)$ TOF5$(a, b, d, e, g)$ TOF5$(b, c, d, e, g)$ TOF2$(d, e)$.

The next set of examples is introduced for the first time in this work.

*Example 10:* The *majority5* function outputs one if three or more of the five inputs are one. Otherwise, it outputs zero. Four garbage outputs are necessary to make the function reversible.

Specification: {0, 1, 2, 3, 4, 5, 6, 27, 7, 8, 9, 28, 10, 29, 30, 31, 11, 12, 13, 16, 14, 17, 18, 19, 15, 20, 21, 22, 23, 24, 25, 26}.
Toffoli circuit: TOF3$(d, e, a)$ TOF4$(a, d, e, b)$ TOF2$(e, d)$ TOF2$(d, a)$ TOF3$(a, d, b)$ TOF3$(c, d, a)$ TOF3$(b, d, c)$ TOF4$(a, b, c, d)$ TOF4$(a, b, d, c)$ TOF3$(c, d, b)$ TOF2$(e, c)$ TOF3$(c, d, e)$ TOF4$(a, d, e, b)$ TOF2$(e, d)$ TOF4$(a, b, d, e)$ TOF3$(c, e, d)$.

*Example 11:* The *decod24* function is a 2:4 decoder with two inputs and four outputs. To make the function reversible, it is necessary to add two garbage inputs.

Specification: {1, 2, 4, 8, 0, 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15}.
Toffoli circuit: TOF2$(c, a)$ TOF2$(d, b)$ TOF2$(c, b)$ TOF3$(a, d, b)$ TOF2$(d, a)$ TOF2$(b, c)$ TOF4$(a, b, c, d)$ TOF3$(b, d, c)$ TOF2$(c, a)$ TOF2$(a, b)$ TOF1$(a)$.

*Example 12:* The *5one013* function outputs one if the number of ones in the binary encoding of the input vector is equal to zero, one, or three. Otherwise, it outputs zero. Four garbage outputs are necessary to make the function reversible.

Specification: {16, 17, 18, 3, 19, 4, 5, 20, 21, 6, 7, 22, 8, 23, 24, 9, 25, 10, 11, 26, 12, 27, 28, 13, 14, 29, 30, 15, 31, 0, 1, 2}.
Toffoli circuit: TOF2$(d, a)$ TOF1$(e)$ TOF2$(e, d)$ TOF2$(e, c)$ TOF2$(e, b)$ TOF2$(a, b)$ TOF2$(c, a)$ TOF2$(c, e)$ TOF5$(a, b, c, d, e)$ TOF1$(a)$ TOF2$(d, c)$ TOF4$(a, b, e, c)$ TOF1$(e)$ TOF1$(d)$ TOF2$(b, e)$ TOF2$(e, b)$ TOF3$(b, c, d)$ TOF3$(a, e, b)$ TOF5$(a, b, c, e, d)$.

Function *5one245* is similar but outputs one if the number of ones in the binary encoding of the inputs is equal to two, four, or five. Functions *6one135* and *6one0246* are an extension of Example 12 to six variables. Results for these functions will be presented later.

*Example 13:* The Boolean specification for the *alu* function is given in Fig. 9. There are three control signals, i.e.: 1) $C_0$; 2) $C_1$; and 3) $C_2$, and two data inputs, i.e.: 1) $A$ and 2) $B$. The control signals determine the logic operation performed on the data inputs. Four garbage outputs are necessary to make the function reversible.

| $C_0$ | $C_1$ | $C_2$ | $F$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | $A + B$ |
| 0 | 1 | 0 | $\bar{A} + \bar{B}$ |
| 0 | 1 | 1 | $A \cdot \bar{B} + \bar{A} \cdot B$ |
| 1 | 0 | 0 | $A \cdot B + \bar{A} \cdot \bar{B}$ |
| 1 | 0 | 1 | $A \cdot B$ |
| 1 | 1 | 0 | $\bar{A} \cdot \bar{B}$ |
| 1 | 1 | 1 | 0 |

Fig. 9. Boolean specification for *alu* benchmarck.

Specification: {16, 17, 18, 19, 0, 20, 21, 22, 23, 24, 25, 11, 12, 26, 27, 15, 28, 13, 14, 29, 8, 9, 10, 30, 31, 1, 2, 3, 4, 5, 6, 7}.

Toffoli circuit: TOF2$(e, c)$ TOF2$(c, e)$ TOF2$(e, d)$ TOF1$(e)$ TOF4$(a, b, d, e)$ TOF3$(b, c, a)$ TOF3$(a, c, e)$ TOF3$(d, e, a)$ TOF2$(c, d)$ TOF4$(a, d, e, b)$ TOF3$(b, c, a)$ TOF4$(a, c, e, b)$ TOF2$(d, c)$ TOF5$(a, b, c, e, d)$ TOF1$(c)$ TOF2$(e, c)$ TOF3$(d, e, c)$ TOF4$(b, d, e, c)$.

*Example 14:* The *shifter* function has two control signals, i.e.: 1) $s_0$ and 2) $s_1$, and $n$ inputs. Depending on the value of the control signals, the function does a wraparound shift of zero, one, two, or three positions on the input. The control signals are passed unchanged to the output. For example, if the control signals are 10, then $\{0, 1, 2, 3, \ldots, 2^n - 1\}$ will become $\{2, 3, \ldots, 2^n - 1, 0, 1\}$. When $n$ is 10, 15, and 28, respectively, the synthesized circuit contains 27, 30, and 56 Toffoli gates, respectively.

### D. Benchmark Functions

We next present synthesis results for many of the benchmarks available from [13]. Note that the results presented in [13] are the best available for each benchmark from various sources in the literature, i.e., these results are not all due to the same algorithm from the literature. Table IV shows the circuit name, how many real and garbage inputs it contains, the number of gates required for its implementation, and its quantum cost. The quantum cost is calculated by using the cost table for Toffoli gates available in [13]. To make a fair comparison, we compare our results with the best published results for the GT library available from [13]. In cases where the circuit contains gates from the NCT library only, we compare our results with the best published results for the NCT library. The † symbol annotating the benchmark name in Table IV highlights such cases.

In general, our results are on par with those from [13]. In some examples, we get identical results. In other cases, we synthesize a circuit with fewer gates but with a higher quantum cost or vice versa. There are only two cases, namely, *5mod5* and *shift10*, in which our tool synthesizes a circuit that contains more Toffoli gates and a higher quantum cost. With the aid of the postprocessing step using the template-based simplification tool [21], it may be possible to further reduce both the gate count and quantum cost of the circuits synthesized by our tool. Note that no comparison can be made for the examples that we have developed. We will make these examples available to the reversible logic community. Due to memory constraints, our algorithm was not able to find a solution to some examples, namely, in the *ham#*, *hwb#*, and *#symm* family of functions given in [13].

TABLE IV
REVERSIBLE LOGIC BENCHMARKS

| Benchmark | Real inputs | Garbage inputs | Gates | Cost | Gates [13] | Cost [13] |
|---|---|---|---|---|---|---|
| 2of5 | 5 | 2 | 20 | 100 | 15 | 107 |
| rd32† | 3 | 1 | 4 | 8 | 4 | 8 |
| 3_17† | 3 | 0 | 6 | 14 | 6 | 12 |
| 4_49 | 4 | 0 | 13 | 61 | 16 | 58 |
| alu | 5 | 0 | 18 | 114 | – | – |
| rd53 | 5 | 2 | 13 | 116 | 16 | 75 |
| xor5† | 5 | 0 | 4 | 4 | 4 | 4 |
| 4mod5† | 4 | 1 | 5 | 13 | 5 | 13 |
| 5mod5 | 5 | 1 | 11 | 91 | 10 | 90 |
| ham3† | 3 | 0 | 5 | 9 | 5 | 7 |
| ham7 | 7 | 0 | 24 | 68 | 23 | 81 |
| hwb4† | 4 | 0 | 15 | 35 | 17 | 63 |
| decod24 | 4 | 0 | 11 | 31 | – | – |
| shift10 | 12 | 0 | 27 | 1,469 | 19 | 1,198 |
| shift15 | 17 | 0 | 30 | 3,500 | – | – |
| shift28 | 30 | 0 | 56 | 14,310 | – | – |
| 5one013 | 5 | 0 | 19 | 95 | – | – |
| 5one245 | 5 | 0 | 20 | 104 | – | – |
| 6one135† | 6 | 0 | 5 | 5 | – | – |
| 6one0246† | 6 | 0 | 6 | 6 | – | – |
| majority3† | 3 | 0 | 4 | 16 | – | – |
| majority5 | 5 | 0 | 16 | 104 | – | – |
| graycode6 | 6 | 0 | 5 | 5 | 5 | 5 |
| graycode10 | 10 | 0 | 9 | 9 | 9 | 9 |
| graycode20 | 20 | 0 | 19 | 19 | 19 | 19 |
| mod5adder | 6 | 0 | 19 | 127 | 21 | 125 |
| mod32adder | 10 | 0 | 15 | 154 | – | – |
| mod15adder | 8 | 0 | 10 | 71 | – | – |
| mod64adder | 12 | 0 | 26 | 333 | – | – |

†Comparison made using NCT library.

– Results not available from [13].

### E. Scalability

In the final set of experiments, we wanted to test the scalability of our algorithm on reversible functions with a relatively large number of inputs. Experiments with four- and five-variable functions indicate that the algorithm can synthesize circuits with gate counts up to around 45. Consequently, we generated random reversible logic circuits of 6–16 variables with a prespecified number of gates. The circuit was constructed by picking a gate at random from a given library (GT or NCT). The gate was then concatenated to the end of the circuit. The process was repeated until the specified number of gates had been selected and added. In the case of the GT library, the number of control bits for each Toffoli

TABLE V
RANDOM REVERSIBLE FUNCTIONS WITH MAXIMUM GATE COUNT OF 15

| | Circuit size | | | | | | | | Failed | |
|---|---|---|---|---|---|---|---|---|---|---|
| Variables | $1-5$ | $6-10$ | $11-15$ | $16-20$ | $21-25$ | $26-30$ | $31-35$ | $36-40$ | No. | % |
| 6 | 173 | 155 | 110 | 46 | 11 | 3 | 1 | 0 | 1 | 0.2 |
| 7 | 159 | 147 | 105 | 58 | 18 | 12 | 1 | 0 | 0 | 0 |
| 8 | 181 | 134 | 93 | 51 | 27 | 5 | 4 | 1 | 4 | 0.8 |
| 9 | 160 | 116 | 115 | 63 | 23 | 10 | 6 | 1 | 6 | 1.2 |
| 10 | 152 | 132 | 114 | 68 | 16 | 11 | 4 | 0 | 3 | 0.6 |
| 11 | 176 | 127 | 106 | 53 | 17 | 10 | 3 | 1 | 7 | 1.4 |
| 12 | 152 | 117 | 108 | 66 | 20 | 13 | 5 | 5 | 14 | 2.8 |
| 13 | 161 | 132 | 98 | 56 | 25 | 9 | 3 | 0 | 16 | 3.2 |
| 14 | 145 | 151 | 95 | 44 | 27 | 16 | 6 | 1 | 15 | 3.0 |
| 15 | 167 | 131 | 89 | 55 | 19 | 11 | 5 | 0 | 23 | 4.6 |
| 16 | 160 | 141 | 95 | 48 | 28 | 7 | 1 | 2 | 18 | 3.6 |

**Note:** 500 random examples each were generated for a given number of variables.

TABLE VI
RANDOM REVERSIBLE FUNCTIONS WITH MAXIMUM GATE COUNT OF 20

| | Circuit size | | | | | | | | Failed | |
|---|---|---|---|---|---|---|---|---|---|---|
| Variables | $1-5$ | $6-10$ | $11-15$ | $16-20$ | $21-25$ | $26-30$ | $31-35$ | $36-40$ | No. | % |
| 6 | 260 | 231 | 171 | 153 | 113 | 48 | 17 | 6 | 1 | 0.1 |
| 7 | 218 | 215 | 170 | 146 | 122 | 70 | 32 | 22 | 5 | 0.5 |
| 8 | 227 | 202 | 167 | 122 | 109 | 81 | 40 | 26 | 26 | 2.6 |
| 9 | 240 | 177 | 166 | 130 | 98 | 73 | 34 | 26 | 56 | 5.6 |
| 10 | 223 | 219 | 153 | 119 | 86 | 68 | 32 | 34 | 66 | 6.6 |
| 11 | 227 | 213 | 150 | 116 | 81 | 55 | 35 | 33 | 90 | 9.0 |
| 12 | 233 | 225 | 164 | 107 | 69 | 48 | 25 | 18 | 111 | 11.1 |
| 13 | 223 | 222 | 153 | 120 | 75 | 37 | 28 | 17 | 125 | 12.5 |
| 14 | 238 | 224 | 154 | 90 | 46 | 49 | 27 | 21 | 151 | 15.1 |
| 15 | 237 | 205 | 178 | 81 | 68 | 37 | 14 | 18 | 162 | 16.2 |
| 16 | 258 | 182 | 172 | 89 | 58 | 32 | 22 | 27 | 160 | 16.0 |

**Note:** $1,000$ random examples each were generated for a given number of variables.

gate was determined randomly as well. The circuits were then simulated to obtain their reversible specifications. Next, the PPRM expansions for this specification were derived. Finally, synthesis was performed on the PPRM specification. The CPU time limit for synthesis for each function was again set to 60 s, and the greedy option for substitution pruning was used.

Tables V–VII show the results for three different scenarios. In Table V, 500 examples each, containing 6–16 variables, with at most 15 gates, were generated randomly. In Tables VI and VII, 1000 such examples each were generated requiring at most 20 and 25 gates, respectively. For testing the scalability of our algorithm, all we were interested in observing was whether a solution (not necessarily the best) could be found or not. As soon as a solution was found, we chose to move on to the next example. That is why, for example, Table V contains results that require more than 15 gates, although we know that all the examples require at most 15 gates. The column "Failed" indicates the number of examples that failed to synthesize in the allotted time.

From Tables V and VI, we see that a vast majority of the circuits could be synthesized in 60 s. If more time were allowed, the circuit size would improve. However, we see a large proportion of circuits failing to synthesize in Table VII in the allotted time. This is again because we are pruning a lot of potential substitutions at each iteration with the greedy option. Better pruning strategies need to be developed to improve this algorithm. Nonetheless, it is comforting to see that the algorithm can still quickly synthesize more than half (50%) of the circuits in such cases.

## VI. CONCLUSION

We have described an algorithm and tool that uses a PPRM expansion of a reversible function to synthesize a network of Toffoli gates. The algorithm searches the tree of possible factors in priority order to try to find the best possible solution. The use of extensive pruning leads to very fast synthesis. We applied our algorithm to all 40 320 reversible functions of three

TABLE VII
RANDOM REVERSIBLE FUNCTIONS WITH MAXIMUM GATE COUNT OF 25

| | Circuit size | | | | | | | | Failed | |
|---|---|---|---|---|---|---|---|---|---|---|
| Variables | $1-5$ | $6-10$ | $11-15$ | $16-20$ | $21-25$ | $26-30$ | $31-35$ | $36-40$ | No. | % |
| 6 | 189 | 202 | 158 | 132 | 103 | 76 | 57 | 72 | 11 | 1.1 |
| 7 | 215 | 152 | 132 | 119 | 88 | 73 | 83 | 84 | 54 | 5.4 |
| 8 | 179 | 167 | 129 | 122 | 84 | 70 | 74 | 78 | 97 | 9.7 |
| 9 | 191 | 166 | 128 | 101 | 68 | 64 | 68 | 57 | 157 | 15.7 |
| 10 | 201 | 156 | 121 | 106 | 61 | 62 | 35 | 39 | 219 | 21.9 |
| 11 | 202 | 163 | 117 | 87 | 73 | 49 | 32 | 47 | 230 | 23.0 |
| 12 | 164 | 156 | 146 | 106 | 56 | 36 | 36 | 25 | 275 | 27.5 |
| 13 | 201 | 176 | 122 | 74 | 57 | 40 | 42 | 25 | 263 | 26.3 |
| 14 | 197 | 160 | 138 | 76 | 45 | 35 | 22 | 32 | 295 | 29.5 |
| 15 | 166 | 172 | 103 | 50 | 29 | 13 | 8 | 7 | 452 | 45.2 |
| 16 | 173 | 183 | 128 | 60 | 37 | 17 | 11 | 8 | 383 | 38.3 |

**Note:** $1,000$ random examples each were generated for a given number of variables.

variables and obtained good quality results. We also showed that it can handle all four-variable functions and most five-variable functions that were in our test suite. We presented results on several benchmarks and also introduced some new ones. Experiments on scalability indicate that our algorithm can quickly find solutions to a good proportion of the randomly generated functions with 6–16 inputs. However, it has most success in synthesizing those functions that require no more than 25 gates on average.

As part of future work, we would like to incorporate Fredkin gates into our algorithm. A Fredkin gate is equivalent to three Toffoli gates. Thus, the use of Fredkin gates could yield a significant improvement in circuit quality. We also want to improve the pruning process, so that we can handle circuits that are currently beyond the capabilities of our algorithm. In particular, there is a need for more powerful heuristics. We are also working on ways to efficiently synthesize functions with "don't cares." We currently preassign values to "don't care" outputs. It would be better if we could find a way to dynamically assign these values during synthesis.

## ACKNOWLEDGMENT

The authors would like to thank D. Maslov for pointing out an error in a circuit presented in the conference version [26] of this paper.

## REFERENCES

[1] R. Landauer, "Irreversibility and heat generation in the computing process," *IBM J. Res. Develop.*, vol. 5, no. 3, pp. 183–191, Jul. 1961.
[2] D. Maslov, "Reversible logic synthesis," Ph.D. dissertation, Dept. Comput. Sci., Univ. New Brunswick, Fredericton, NB, Canada, 2003.
[3] C. Bennett, "Logical reversibility of computation," *IBM J. Res. Develop.*, vol. 17, no. 6, pp. 525–532, Nov. 1973.
[4] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*. Cambridge, U.K.: Cambridge Univ. Press, 2000.
[5] A. Mishchenko and M. Perkowski, "Logic synthesis of reversible wave cascades," in *Proc. Int. Workshop Logic Synthesis*, Jun. 2002, pp. 197–202.
[6] P. Kerntopf, "A new heuristic algorithm for reversible logic synthesis," in *Proc. Des. Autom. Conf.*, Jun. 2004, pp. 834–837.
[7] D. M. Miller, D. Maslov, and G. W. Dueck, "A transformation based algorithm for reversible logic synthesis," in *Proc. Des. Autom. Conf.*, Jun. 2003, pp. 318–323.
[8] T. Toffoli, "Reversible computing," MIT, Cambridge, MA, 1980. Tech. Rep.
[9] E. Fredkin and T. Toffoli, "Conservative logic," *Int. J. Theor. Phys.*, vol. 21, no. 3/4, pp. 219–253, 1982.
[10] T. Sasao, *Logic Synthesis and Optimization*. Dordrecht, The Netherlands: Kluwer, 1993.
[11] M. Thornton, *Fixed Polarity Reed–Muller Forms*. [Online]. Available: http://engr.smu.edu/~mitch/class/8391/week12/esop.pdf
[12] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter, "Elementary gates for quantum computation," *Phys. Rev. A, Gen. Phys.*, vol. 52, no. 5, pp. 3457–3467, Nov. 1995.
[13] D. Maslov. *Reversible Logic Synthesis Benchmarks Page.* (2005, May). [Online]. Available: http://www.cs.uvic.ca/~dmaslov/
[14] V. V. Shende, I. L. Markov, and S. S. Bullock, "Smaller two-qubits circuits for quantum communication and computation," in *Proc. Des. Autom. Test Eur. Conf.*, Feb. 2004, pp. 980–985.
[15] A. Mishchenko and M. Perkowski, "Fast heuristic minimization of exclusive sum-of-products," in *Proc. Int. Workshop Appl. Reed–Muller Expansion Circuit Des.*, Aug. 2001, pp. 242–250.
[16] V. V. Shende, A. K. Prasad, I. L. Markov, and J. P. Hayes, "Synthesis of reversible logic circuits," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 22, no. 6, pp. 710–722, Jun. 2003.
[17] V. V. Shende, A. K. Prasad, K. N. Patel, I. L. Markov, and J. P. Hayes, "Scalable simplification of reversible circuits," in *Proc. Int. Workshop Logic Synthesis*, May 2003.
[18] D. M. Miller and G. W. Dueck, "Spectral techniques for reversible logic synthesis," in *Proc. Int. Symp. Represent. Methodol. Future Comput. Technol.*, Mar. 2003, pp. 56–62.
[19] K. Iwama, Y. Kambayashi, and S. Yamashita, "Transformation rules for designing CNOT-based quantum circuits," in *Proc. Des. Autom. Conf.*, Jun. 2002, pp. 419–425.
[20] D. Maslov, G. W. Dueck, and D. M. Miller, "Simplification of Toffoli networks via templates," in *Proc. Symp. Integr. Circuits Syst. Des.*, Sep. 2003, pp. 53–58.
[21] ——, "Fredkin/Toffoli templates for reversible logic synthesis," in *Proc. Int. Conf. Comput.-Aided Des.*, Nov. 2003, pp. 256–261.
[22] ——, "Templates for Toffoli network synthesis," in *Proc. Int. Workshop Logic Synthesis*, May 2003, pp. 320–326.
[23] RMRLS. [Online]. Available: http://www.princeton.edu/~cad/projects.html
[24] D. Maslov, Private communication, Dec. 2004.
[25] N.C.S.U. *Benchmark Archives at CBL* Collaborative Benchmarking Lab., Dept. of Comput. Science, North Carolina State Univ. [Online]. Available: http://www.cbl.ncsu.edu/benchmarks/
[26] A. Agrawal and N. K. Jha, "Synthesis of reversible logic," in *Proc. Des. Autom. Test Eur. Conf.*, Feb. 2004, pp. 21384–21385.

**Pallav Gupta** (S'99) received the B.S. degree in computer engineering (*summa cum laude*) from the University of Arizona, Tucson, in 2002, and is currently working toward the Ph.D. degree in electrical engineering at Princeton University, Princeton, NJ.

His research interests include computer-aided design for emerging nanotechnologies such as quantum cellular automata, resonant tunneling diodes, carbon nanotubes, reversible logic, and computing paradigms for emerging technologies.

Mr. Gupta was the recipient of the Wallace Memorial Fellowship for the 2005–2006 academic year given to top graduate students at Princeton University.

**Abhinav Agrawal** received the B.S. degree in electrical engineering (*summa cum laude*) from Princeton University, Princeton, NJ, in 2004.

His undergraduate research work examined the vulnerabilities of the Intel NetBurst microarchitecture to heat and voltage fluctuations, which he presented at the Intel Undergraduate Research Symposium. Currently, he is a Business Analyst with McKinsey and Company, New York, NY, working with clients on strategy, operational improvement, sales, and marketing.

Mr. Agrawal was elected to Phi Beta Kappa and Tau Beta Pi, upon graduation.

**Niraj K. Jha** (S'85–M'85–SM'93–F'98) received the B.Tech. degree in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, India, in 1981, the M.S. degree in electrical engineering from the State University of New York, Stony Brook, in 1982, and the Ph.D. degree in electrical engineering from the University of Illinois, Urbana, in 1985.

He is a Professor of electrical engineering at Princeton University, Princeton, NJ. He served as the Director of the Center for Embedded System-on-a-Chip Design funded by the New Jersey Commission on Science and Technology. He has coauthored three books entitled *Testing and Reliable Design of CMOS Circuits* (Norwell, MA: Kluwer, 1990), *High-Level Power Analysis and Optimization* (Norwell, MA: Kluwer, 1998), and *Testing of Digital Systems* (Cambridge, U.K.; Cambridge University Press, 2003). He has also authored six book chapters. He has authored or coauthored more than 280 technical papers. He has received 11 U.S. patents. His research interests include low-power hardware and software design, CAD of integrated circuits and systems, digital-system testing, and distributed computing.

Dr. Jha is an ACM Fellow. He is the recipient of the AT&T Foundation Award and NEC Preceptorship Award for research excellence, NCR Award for teaching excellence, and Princeton University Graduate Mentoring Award. He has coauthored six papers which have won the Best Paper Award at ICCD'93, FTCS'97, ICVLSID'98, DAC'99, PDCS'02, and ICVLSID'03. Another paper of his was selected for "The Best of ICCAD: A collection of the best IEEE International Conference on Computer-Aided Design papers of the past 20 years." He has served as an Associate Editor of IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS II: ANALOG AND DIGITAL SIGNAL PROCESSING. He is currently serving as an Editor of IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN, IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, *Journal of Electronic Testing: Theory and Applications* (JETTA), and *Journal of Embedded Computing*. He has served as the Guest Editor for the JETTA special issue on high-level test synthesis. He has also served as the Program Chair of the 1992 Workshop on Fault-Tolerant Parallel and Distributed Systems and the 2004 International Conference on Embedded and Ubiquitous Computing.