Quantum Logic Circuit Simulation Based on the QMDD Data Structure

David Goodman, Mitchell A. Thornton, David Y. Feinstein
Department of Computer Science and Engineering
Southern Methodist University
Dallas, TX, USA
{dgoodman,mitch,dfeinste}@engr.smu.edu

D. Michael Miller
Department of Computer Science
University of Victoria
Victoria, BC, Canada
mmiller@cs.uvic.ca

Abstract

Two approaches for a quantum logic circuit simulator based on Quantum Multiple-valued Decision Diagrams (QMDDs) are formulated and compared. The two approaches for quantum logic circuit simulation are performed by explicit multiplication using QMDD representations of the circuit and input vector, and through implicit multiplication where the QMDD representing the circuit undergoes a guided traversal based upon the input vector. These methods are implemented and compared in terms of memory usage and runtime.

1. Introduction

Recently there has been significant interest in quantum computations since many operations that are intractable using classical computers can be solved efficiently using quantum algorithm formulations including Shor's factorization algorithm [1] and Grover's search algorithm [2]. Additionally, all quantum logic operations are necessarily both logically and physically reversible. As proven by Landauer in 1961, only reversible circuits can approach the theoretical limit of no heat dissipation and hence zero power dissipation [3]. For this reason, there is also a large amount of interest in the development of practical quantum logic circuits for the purpose of low-power dissipation.

Quantum logic circuits represent transformations of the state of one or more qubits over time or space and these circuits are modeled as cascades of quantum logic gates. Because quantum logic gates and circuits are represented by unitary transformation matrices, the transformation of the initial state of a set of qubits into a resulting output state can be mathematically computed as a matrix-vector product where the matrix represents the quantum logic circuit and the initial qubit state is represented as a column vector. Unfortunately, for an n-qubit circuit, the size of the transformation matrix is $2^n \times 2^n$ and the column vector representing the initial state of the n qubits is of size 2^n. For these reasons, the implementation of a quantum logic circuit simulator using explicit matrices and vectors is only useful for very small circuits and we are motivated to develop more efficient approaches for quantum logic circuit simulation.

Linear algebra computations have been shown to be efficiently implemented using decision diagram structures in the past. In [4] an efficient means for computing the Walsh spectrum of a Boolean function using *Binary Decision Diagrams* (BDDs) [5] is shown that, as a side-effect, illustrates how the tensor or Kronecker matrix product can be computed using graph algorithms. Also, in [6] an efficient means for the computation of a matrix-vector product using BDDs is described. These ideas and others were used for the development of a quantum logic simulator using BDD operations in the QuiDDPro software [7,8] that uses the very efficient BDD software package CUDD [9].

A new decision diagram structure for the representation of quantum logic circuits called the *Quantum Multiple-valued Decision Diagram* (QMDD) is described in [10,11,12]. Although this is also a decision diagram structure, it represents a matrix in a very different way than is usually accomplished through the use of BDDs. The QMDD structure is also easily extended to handle quantum circuits with multiple-valued (i.e. non-binary) qudits as well as binary-valued qubits [10,12].

In this paper, two different approaches for quantum logic circuit simulation based on QMDD manipulations are described. The first technique performs explicit matrix-vector multiplication by representing the quantum logic circuit as a QMDD, the initial state of the qubits as a QMDD, and then invoking a recursive multiplication function that produces a product QMDD. The second method also represents the quantum logic circuit as a QMDD, but then performs a guided traversal where the product vector is implicitly computed during the traversal. The QMDD traversal is guided by the value of the qubit initial state vector.

The paper is organized as follows. Quantum logic circuit matrix representations and QMDDs are described in more detail in Section 2. In Section 3, a description of the two simulation approaches is provided. Section 4 contains experimental results. Conclusions and areas of future improvement are presented in Section 5.

2. Matrix Representations of Quantum Circuits and QMDDs

Many different quantum gates have been developed, and they can all be modeled with a unitary transformation matrix. Some of the gates we consider are binary reversible gates and are represented by 0/1-valued permutation matrices. In the general case, a quantum logic gate is represented as a vector rotation within the Bloch sphere and corresponding transformation matrices contain complex-valued components. Quantum logic circuits are formulated as cascades of quantum logic gates and the transformation matrix representing the entire quantum circuit is computed as the product of the matrices representing the individual gates. Furthermore, an n-qubit circuit is formed utilizing n Kronecker matrix products. The QMDD structure is formulated such direct matrix and Kronecker product algorithms are implemented as recursive graph traversal functions.

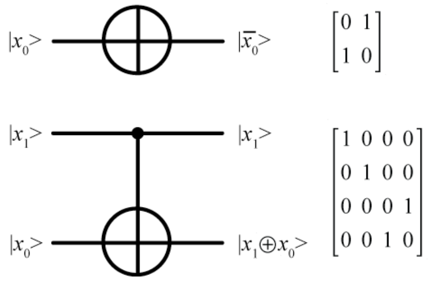
2.1. Quantum Logic Gates

Although numerous quantum logic gates have been defined, here we focus on the following subset of quantum logic gates since that are currently supported by the QMDD software.

- Quantum NOT gates
- Feynman (Controlled-NOT) gates
- Toffoli (Controlled-Controlled-NOT) gates
- Generalized (Multiple Control) Toffoli gates
- Controlled-V (square root of NOT) gate
- Controlled-V† (inverse of V) gate

Figure 1 shows the standard diagrams for the NOT and Controlled-NOT gates with their corresponding transformation matrices. The quantum NOT gate causes a single qubit initially in eigenstate (or basis state) |0> to be transformed to the |1> state and vice-versa. The controlled-NOT logic gate is a 2-qubit circuit that transforms one qubit from its initial eigenstate to the opposite basis state if, and only if, the other 'control' qubit is in eigenstate |1>. The control qubit remains unchanged. The Toffoli gate is an extension of the controlled-NOT gate in that it utilizes two control qubits and the generalized Toffoli gate operates with more than two control qubits. These operations can also be described with Boolean relationships as is also shown in Figure 1.

The 2-qubit controlled-V gate transforms the target qubit by using the transformation given by the matrix $\mathbf{V} = \frac{1+i}{2}\begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix}$ if the control qubit is in state |1>. Similarly, the controlled-V† gate transforms the target qubit using the matrix $\mathbf{V}^\dagger = \mathbf{V}^{-1} = \frac{1-i}{2}\begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix}$, if the control qubit is |1>. The $\mathbf{V}$ transformation matrix is sometimes referred to as the square root of NOT gate since a cascade of two $\mathbf{V}$ gates produces the same transformation as a quantum NOT gate. $\mathbf{V}^\dagger$ of course has the same property since $\mathbf{V}^\dagger = \mathbf{V}^{-1}$ and $\mathbf{V}$ is a unitary matrix.



$|x_0>$
$|\overline{x}_0>$
$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
$|x_1>$
$|x_1>$
$|x_0>$
$|x_1 \oplus x_0>$
$\begin{bmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&1 \\ 0&0&1&0 \end{bmatrix}$

Figure 1: NOT and Controlled-NOT Gates

2.2. Matrix Representations

Since a quantum gate can be represented by a matrix, an n-qubit gate results in a matrix of size $2^n \times 2^n$. A cascade of gates forming a quantum logic circuit can also be represented by a single matrix formed by the direct multiplication of the matrices representing the individual gates.

As an example, a single Toffoli gate with variables x_0 and x_2 serving as control qubits and the variable x_1 as the target qubit (denoted by T(x_2,x_0;x_1)) is shown in Figure 2. The corresponding transformation matrix for T(x_2,x_0;x_1) is given in (1) where the qubit ordering is x_2 serving as the most-significant qubit and x_0 being the least significant qubit:

$$\begin{bmatrix} 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&1&0&0 \end{bmatrix} \qquad (1)$$

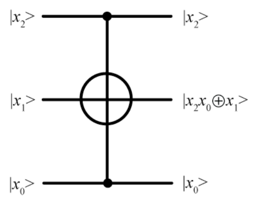


Figure 2. Toffoli Logic Gate: T(x_2,x_0;x_1)

2.3. The QMDD Representation

The main motivation for creating the QMDD structure is the regular structure of the matrices that represent quantum and reversible circuits. Since, for binary circuits, the dimensions of these matrices are always powers of 2, the matrices can always be partitioned into four quadrants. For example, the matrix **M** can be partitioned as

$$\mathbf{M} = \begin{bmatrix} \mathbf{M}_0 & \mathbf{M}_1 \\ \mathbf{M}_2 & \mathbf{M}_3 \end{bmatrix} \qquad (2)$$

and each of the partitions has dimensions $2^{n-1} \times 2^{n-1}$. Each of these submatrices can also be further partitioned into four smaller matrices. This process can recursively continue until the four submatrices become single values. The selection variables in a QMDD are ordered [5] indicating that each variable appears no more than once on any path through the QMDD based on a predefined ordering that is identical among all possible paths. Also, standard decision diagram techniques [5] are used to reduce the QMDD by sharing common subdiagrams and by removing redundant subdiagrams. A complex-valued multiplier constant annotates each edge in a QMDD allowing for the extraction of common factors from the corresponding matrices. QMDDs contain a single terminal vertex annotated with value 1.

In order to evaluate a QMDD for a particular selection variable assignment, the path from the starting vertex to the terminal vertex determined by a particular assignment is followed. For example, when encountering the variable x_i one follows edge number j where j is the value that is assigned to x_i. It is possible that some of the variables will not appear on the designated path. This simply means that this particular variable does not affect the result for this particular valuation of x_i. From the results in [12], it is proven that skipped variables only occur when the exiting edge points directly to the terminal vertex of the QMDD and has an edge multiplier value of 0. The value associated with a path in a QMDD is the product of the edge multipliers encountered on the path including the edge leading to the start vertex.

As an example, Figure 3 shows the QMDD representation of the quantum Toffoli gate, T(x_2,x_0;x_1), represented by the transformation matrix previously given in (1). The edges 0, 1, 2 and 3 are labeled from left to right in the diagram and correspond to the subscripts of the submatrices in Equation (2). A complete traversal of the QMDD in Figure 3 allows for the formulation of the matrix representation of the gate.

QMDDs are also easily generalized for the representation of multiple-valued quantum circuits. For r-valued logic, each matrix has dimension $r^n \times r^n$ and each nonterminal vertex in the corresponding QMDD has r^2 outgoing edges (see [10] for details).

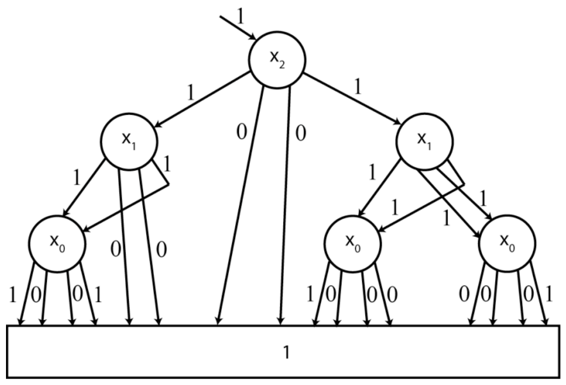


Figure 3. QMDD for Toffoli Gate T($\mathbf{x_2}$,$\mathbf{x_0}$;$\mathbf{x_1}$)

2.4. Matrix Operations

Another crucial aspect of a quantum circuit simulator is that operations performed on the matrices represented as QMDDs be realized efficiently. The major operations that are needed are matrix addition, matrix multiplication, and the Kronecker product. Each of these operations are implemented as recursive QMDD graph algorithms as described in [10,11].

3. Quantum Logic Circuit Simulation

Our circuit simulators are designed to accept two different types of circuit description formats. These formats are those used by Maslov in his benchmark circuit set [13], and the QASM format [14,15]. The QASM format is a text-format language that is used to describe quantum circuits (or, from another point of view, a quantum algorithm). The ease of describing quantum circuits in QASM is the principal motivation for its use.

Since the QMDD software previously developed uses the `tfc` file format developed by Maslov [13], we wrote a simple program to translate a QASM specification into the `tfc` format. Given this tool, a QMDD can be built for a circuit described in either QASM or the `tfc` format using the approach described in [10,11].

3.1. Input Vector Formulation and Simulation

Our simulators assume that the input vectors are represented by an n-qubit register with all qubits initialized to eigenstate values of |0> or |1>. In order to apply matrix multiplication to compute the resultant transformed states of the n-qubit register, a column vector is first formed using the Kronecker product relationship that transforms the Dirac-ket notation [16] into an appropriate column vector.

Recall that the Kronecker product is defined as

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}$$

The Kronecker product is not a commutative operation, however, it is associative. In our simulators, we implemented the operation as a sequence of Kronecker products evaluated from right to left.

As an example, for the initial qubit register values of |001>, the column vector is formed as shown in Equation (2).

$$|001> = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\\0\\0\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\\0\\0\\0\\0\end{bmatrix} \quad (3)$$

The resulting column vector is the "input vector" for the circuit.

The next step in the simulation process is to perform a matrix multiplication with the column vector generated from the initial state of the qubit register and the circuit transformation matrix. The resultant product is then a column vector of the same dimension as the input vector that represents the state of the qubit register after it is transformed by the quantum circuit.

The final step is to convert the output vector back to Dirac-ket notation. In general, this can be a difficult process since mathematically it requires formulating a Kronecker product factorization. However since our simulators assume all simulation results will be in eigenstate form, a simple algorithm suffices to convert the output vector back into Dirac-ket form.

3.2. Explicit Multiplication Based Simulator

The explicit multiplication based simulator represents the input vector as a QMDD, the quantum logic circuit as a QMDD, and then calls the QMDD function that performs matrix multiply. In representing the input vector as a QMDD, the column vectors representing |0> and |1> as shown in Equations (4) utilize the 0- and 2-edges of a QMDD vertex and the 1- and 3-edges point to a null value. Figure 4 shows QMDD representations of |0> and |1> respectively.

$$|0> = \begin{bmatrix}1\\0\end{bmatrix} \qquad |1> = \begin{bmatrix}0\\1\end{bmatrix} \qquad (4)$$

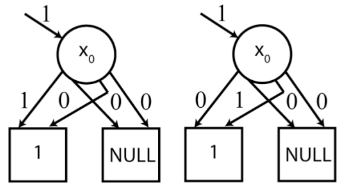


Figure 4. QMDDs for |0> and |1>

Once the qubits are represented as QMDD structures, they are combined into a single QMDD representing the entire input vector using computations as shown in Equation (3). This is accomplished by invoking the Kronecker multiplication function for QMDDs.

After the QMDD representing the input vector has been created, the actual simulation can take place through a call to the QMDD matrix multiplication function.

The following example shows a simulation computation by using matrices and vectors in Equation (5) instead of the actual QMDDs that represent them. In this example, the matrix represents the circuit QMDD and the vector on the lefthand side of the equation represents the input column vector. The product column vector on the right-hand side of the equation represents the QMDD that is returned by the multiplication function that is then factored into Kronecker products of 2-dimensional vectors. This example consists of a single Toffoli gate (T(x_2,x_0;x_1)) with target variable x_1 and control variables x_2 and x_0. The initial state of the qubits in this example is |111> resulting in a transformed qubit state of |101>.

$$\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&1&0\\0&0&0&0&0&1&0&0\end{bmatrix}\begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\1\end{bmatrix} = \begin{bmatrix}0\\0\\0\\0\\0\\1\\0\\0\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} \quad (5)$$

3.3. Implicit Multiplication Based Simulator

An alternative approach for quantum circuit simulation involving the QMDD structure is a recursive QMDD traversal guided by the values of the initial state of the n-qubit register. As is done in the explicit multiplication simulation method, a QMDD is first built that represents the quantum circuit and it is then traversed in a guided manner based on the input vector component values. The output vector components of the circuit are computed during the traversal. This implicit multiplication traversal method is described as follows.

Let *e* denote an edge pointing to a QMDD describing a quantum circuit and let *vector* be an array consisting of the initial qubit states. The algorithm is stated where T(*e*) represents a function that is true if *e* points to a terminal vertex of the QMDD and is false otherwise.

1. If T(*e*) is true, then the weight of *e* in the *i*th position is stored in the ith position of the output vector.
2. If T(*e*) is false, then if the value in *vector* corresponding to the variable of the current vertex = 0, the following procedure is followed.
   a. Call the recursive function with the *0th* edge of the current vertex.
   b. Call the recursive function with the *2nd* edge of the current vertex.

   Otherwise,

   c. Call the recursive function with the *1st* edge of the current vertex.
   d. Call the recursive function with the *3rd* edge of the current vertex.

The reasoning for using these particular edges (either the 0 and 2 edge or the 1 and 3 edge) is due to the structure of the matrices and the form of the resulting product of the matrices with vectors representing |0> and |1>. As mentioned before, a matrix can be partitioned into four equal parts. When this partitioned matrix is multiplied by a zero or one qubit, only half of the matrix results in the product vector. Examples are shown below:

$$\begin{bmatrix} \mathbf{M}_0 & \mathbf{M}_1 \\ \mathbf{M}_2 & \mathbf{M}_3 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{M}_1 \\ \mathbf{M}_3 \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{M}_0 & \mathbf{M}_1 \\ \mathbf{M}_2 & \mathbf{M}_3 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{M}_0 \\ \mathbf{M}_2 \end{bmatrix}$$

4. Experimental Results

All of the experimental results were generated using a Dell PE 2650 computer with a Dual Intel Xeon processor running at 3.2GHz with 4 GB of RAM running a Linux operating system.

Table 1 contains results pertaining to the initial processing phase of the simulators where a circuit netlist is parsed and converted into a QMDD representation. These results provide information regarding the runtime and memory usage required for generating the QMDD circuit representation when garbage collection versus no garbage collection is invoked. Garbage collection refers to the process of periodically freeing up memory by eliminating QMDD vertices present in the unique table created during circuit netlist parsing that are no longer needed for final representation. Garbage collection would normally always be invoked in normal operation of the simulator and the results in Table 1 with no garbage collection are included to show the effectiveness of the process. In these results, garbage collection is invoked whenever more than 50,000 QMDD vertices are created.

The first column in Table 1 contains the name of the circuit netlist followed by the size of each circuit in terms of the number of qubits in column 2 and the number of gates in column 3. Each of the example circuits were obtained from Maslov's quantum circuit benchmark website [13].

Columns 4 and 5 of Table 1 contain the peak number of vertices created during the process of building the QMDD as the circuit netlist is parsed both when the garbage collection feature is used (column 4) and not used (column 5). We note that the qubit or variable order utilized in these results is the order in which the qubits are specified in the benchmark circuit netlist. An alternative QMDD size would generally result if a different ordering were used. In these results, no variable reordering techniques are used.

Columns 6 and 7 of Table 1 contain the total runtime required for building each QMDD both with and without the garbage collection feature being utilized. Column 8 contains the final vertex count required for the representation of each benchmark circuit as a QMDD.

Table 2 contains experimental results for the simulation of each benchmark circuit for a set of input stimuli or initial input vectors. Each input vector is specified as a set of initial qubit values in Dirac-ket form, thus the simulation runtimes include the conversion of the input vectors into an appropriate column vector and the conversion of the resultant column vector back into Dirac-ket form. As is the case with Table 1, columns 1, 2, and 3 contain the benchmark circuit name and size in terms of number of qubits and gates.

Column 4 in Table 2 contains the number of randomly generated input vectors used in each simulation. Columns 5, 6, and 7 contain the runtime in seconds for computing the resultant qubit register state for the input vectors. The runtime results in these columns are the average runtime over the set of input vectors for the simulation. We also implemented a simulator that forms the actual matrix and vector and performs a linear algebra matrix-vector product computation. The results in column 5 for the linear algebra computations are included for the purpose of comparing the use of QMDDs to the linear algebra approach for simulation. Column 6 contains the average runtime for the implicit multiplication approach whereas column 7 contains the runtime for the explicit multiplication approach.

In the explicit multiplication approach, a product QMDD is produced representing the circuit output. Column 8 contains the average number of vertices in the unique table after the explicit multiplication is performed and a garbage collection is invoked. Thus, column 8 contains the number of vertices to represent both the original circuit and the product vector. Column 9 contains the peak number of vertices representing the circuit and product vector after the explicit multiplication is performed. This is the peak node count for the input vector that produced the largest peak node count.

5. Conclusions and Future Improvements

The first set of experimental results in Table 1 show the effectiveness of representing the example circuit netlists in QMDD form both with and without garbage collection. For those circuits requiring a relatively large number of QMDD vertices, it is observed that the garbage collection feature can significantly affect runtime. As an example, for the 11 qubit hidden weighted circuit, **`hwb11`**, runtime is only 17% of that required when no garbage collection is used.

Table 2 provides simulator performance results for the two different approaches of producing the circuit response through the implicit versus the explicit multiplication approach. In the implicit multiplication approach, the QMDD representing the benchmark circuit undergoes a traversal that is guided by the input vector component values. This traversal results in many subpaths of the QMDD being traversed multiple times and thus, for large QMDDs, the overall runtime can be larger than the explicit multiplication approach although no runtime is expended for creation of new QMDD vertices since none are created in this method. The explicit multiplication approach actually creates a resultant product QMDD representing the circuit response.

In the explicit multiplication approach, the creation of the product QMDD also results in a traversal of the circuit QMDD during the formation of the product QMDD although subpaths of the circuit QMDD are not traversed multiple times, however, runtime is expended for the creation of new QMDD vertices.

For all but 2 of the example circuits, the two simulation approaches have approximately the same runtime. In the case of benchmarks **`0410184`** and **`Cycle17_3`**, the implicit multiplication method required significantly larger runtimes. This is attributed to the fact that these two examples required many subpaths within the circuit QMDDs to be traversed multiple times.

In the future, we plan to investigate the development of alternative guided QMDD traversal algorithms that can allow for the use of a cache that can be used to avoid repeated traversals of the same subpaths when the implicit multiplication method is used. In these results, the traversal functions are implemented as recursive routines and thus there is no way to predict the entire path that will be traversed during the intermediate computations. If the traversal could be implemented in a way where a table lookup operation could be performed based upon a current vertex and the input vector, repeated path traversals could be avoided and runtimes should be improved.

We also plan to incorporate automatic variable reordering in the simulator as described in [12] that will result in increased performance for both simulation approaches. After these improvements are made to the simulator, we plan to disseminate our simulator programs through the creation of a webpage where source and executables are available for download.

References

[1] P. Shor, Polynomial-time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer, *SIAM Journal of Computing*, vol. 26, 1997, pp. 1484-1509.

[2] L. Grover, A Fast Quantum Algorithm for Database Search, In Proceedings of the *ACM Symposium on Theory of Computing*, 1996, pp. 212-219.

[3] R. Landauer, Irreversibility and Heat Generation in the Computing Process, *IBM Journal of Research Development*, vol. 5, 1961, 183.

[4] D.M. Miller, Graph Algorithms for the Manipulation of Boolean Functions and their Spectra, In Proceedings of the *Congressus Numerantium*, 1987, pp. 177-199.

[5] R.E. Bryant, Graph-based Algorithms for Boolean Function Manipulation, *IEEE Transactions on Computers*, Vol. 35, no. 8, 1986, pp. 677-691.

[6] E.M. Clarke, K.L. McMillan, X. Zhao, M. Fujita, and J. Yang, Spectral Transforms for Large Boolean Functions with Application to Technology Mapping, In Proceedings of the *Design Automation Conference*, 1993, pp. 54-60.

[7] G.F. Viamontes, I.L. Markov, and J.P. Hayes, Graph-based Simulation of Quantum Computation in the Density Matrix Representation, *Quantum Information & Computation*, vol. 5, no. 2, 2005, pp. 113-130.

[8] G.F. Viamontes, I.L. Markov, and J.P. Hayes, QuiDDPro: High-Performance Quantum Circuit Simulation, http://vlsicad.eecs.umich.edu/Quantum/qp/, 2006.

[9] F. Somenzi, The CUDD Package, University of Colorado at Boulder, http://vlsi.colorado.edu/~fabio/, 1995.

[10] D.M. Miller and M.A. Thornton, QMDD: A Decision Diagram Structure for Reversible and Quantum Circuits, In Proceedings of the *IEEE International Symposium on Multiple-Valued Logic*, 2006.

[11] D.M. Miller, M.A. Thornton, and D. Goodman, A Decision Diagram Package for Reversible and Quantum Circuits, In Proceedings of the *IEEE World Congress on Computational Intelligence*, 2006.

[12] D.M. Miller, D.Y. Feinstein, and M.A. Thornton, Variable Reordering and Sifting for QMDD, In Proceedings of the *IEEE International Symposium on Multiple-Valued Logic*, 2007.

[13] D. Maslov. Reversible logic synthesis benchmarks page. http://www.cs.uvic.ca/~dmaslov, 2005.

[14] I. Chuang. Quantum Circuit Viewer: qasm2circ, Massachusetts Institute of Technology, http://www.media.mit.edu/quanta/qasm2circ/.

[15] S. Blaha, Quantum Computers and Quantum Computer Languages: Quantum Assembly Language and Quantum C Language, http://www.citebase.org/abstract?id=oai:arXiv.org:quant-ph/020182.

[16] M.A. Nielsen and I.L. Chuang, **Quantum Computation and Quantum Information**, Cambridge University Press, 2000.

TABLE 1: Experimental Results for Conversion of the Circuit Netlist into a QMDD

Circuit Name	Number of Qubits	Number of Gates	Peak Node Count to Build Circuit (vertices)		Time to Build Circuit (seconds)		Resultant QMDD Size (vertices)
			GC on	GC off	GC on	GC off	
Ham3	3	5	50	50	<0.000	<0.000	10
3_17	3	6	53	53	<0.000	<0.000	10
rd32	4	4	52	52	<0.000	<0.000	9
mod5adders	6	21	303	303	<0.000	<0.000	38
5mod5tc	6	17	394	394	<0.000	<0.000	28
Hwb7	7	289	8746	8746	0.025	0.025	179
rd53rcmg	7	30	1220	1220	<0.000	<0.000	103
Hwb9-1541	9	1541	50000	128731	0.494	0.535	683
Hwb11	11	9314	50000	2205109	11.536	66.544	2639
0410184.nct	14	46	1156	1156	<0.000	<0.000	39
ham15a	15	132	50000	161493	0.496	0.475	26346
Cycle17_3	20	48	3040	3040	0.004	0.004	236
mod1048576adder	40	210	-	-	-	-	-

TABLE 2: Experimental Results Comparing the Explicit and Implicit Multiplication Method Simulators

Circuit Name	Number of Qubits	Number of Gates	Number of Input Vectors	Average CPU Runtime for Linear Algebra Approach (seconds)	Average CPU Runtime for Implicit Multiplication Approach (seconds)	Average CPU Runtime for QMDD Explicit Multiplication Approach (seconds)	Average Resultant QMDD Size for Explicit Multiplication Approach (vertices)	QMDD Peak Node Count for Explicit Multiplication Approach (vertices)
Ham3	3	5	8	< 0.000	0.00005	0.00004	11	20
3_17	3	6	8	< 0.000	0.00005	0.00007	12	21
rd32	4	4	16	<0.000	0.00006	0.00006	18	28
mod5adders	6	21	64	<0.000	0.00007	0.00007	46	51
5mod5tc	6	17	64	<0.000	0.00007	0.00006	38	52
Hwb7	7	289	128	0.040	0.00016	0.00014	220	309
rd53rcmg	7	30	128	<0.000	0.00008	0.00010	127	188
Hwb9-1541	9	1541	512	timeout	0.00084	0.00086	851	1176
Hwb11	11	9314	2048	timeout	0.01028	0.00988	3312	4557
0410184.nct	14	46	16384	timeout	0.00207	0.00009	75	105
ham15a	15	132	328	timeout	0.99800	1.00677	36617	49594
Cycle17_3	20	48	1048	timeout	0.12637	0.00026	255	277
mod1048576adder	40	210	-	timeout	timeout	timeout	-	-