Gate-Level Simulation of Quantum Circuits

George F. Viamontes, Manoj Rajagopalan,
Igor L. Markov and John P. Hayes

The University of Michigan, Advanced Computer Architecture Laboratory
Ann Arbor, MI 48109-2122, USA
{gviamont,rmanoj,imarkov,jhayes}@eecs.umich.edu

Abstract— Simulating quantum computation on a classical computer is a difficult problem. The matrices representing quantum gates, and vectors modeling qubit states grow exponentially with an increase in the number of qubits. However, by using a new data structure called the Quantum Information Decision Diagram (QuIDD) that exploits the structure of quantum operators, many of these matrices and vectors can be represented in a form that grows polynomially. Using QuIDDs, we implemented a general-purpose quantum computing simulator in C++ called QuIDDPro and tested it on Grover's algorithm. Our QuIDD technique asymptotically outperforms other known simulation techniques.

I. Introduction

In modern computers (referred to as 'classical' to distinguish them from their quantum counterparts) binary information is stored in a bit that is physically a voltage signal in a solid-state electronic circuit. Mathematically, a bit is represented as a Boolean value or variable. In the quantum domain, binary information is stored in a quantum state such as the polarization (horizontal/vertical) of a photon or the spin (up/down) of an electron or atomic nucleus. Unlike a classical bit, a quantum bit or *qubit* can exist in a superposition of its classical binary states that is disturbed, or in most cases, destroyed by any external stimulus (typically in a measurement operation). In Dirac notation, the quantum states corresponding to the classical logic zero and one are denoted as $|0\rangle$ and $|1\rangle$, respectively [4]. However, unlike a classical bit, a qubit can store zero and one simultaneously, using values represented by 2-element vectors of the form, $\alpha|0\rangle + \beta|1\rangle$ where α and β are complex numbers and $|\alpha|^2 + |\beta|^2 = 1$. A fundamental postulate of quantum mechanics dictates that these individual state vectors can be combined, via the tensor product, with other state vectors [4]. These tensored qubit states provide a kind of massive parallelism since superposition allows an n-qubit state $|\Psi\rangle = \sum_{i=0}^{2^n-1} c_i|b_{i,n-1}b_{i,n-2},\ldots,b_{i,0}\rangle$ to store 2^n binary numbers simultaneously. Here each c_i is a complex number such that $\sum_{i=0}^{2^n-1}|c_i|^2 = 1$, and $b_{i,n-1}b_{i,n-2},\ldots,b_{i,0}$ is the binary expansion of i. For $n = 2$, we have $|\Psi\rangle = c_0|00\rangle + c_1|01\rangle + c_2|10\rangle + c_3|11\rangle$.

Richard Feynman observed in the early 1980s that simulating quantum processes on classical hardware seems to require super-polynomial (in the number of qubits) memory and time. Consequently, Feynman proposed *quantum computing*, a means of computation involving quantum mechanical states which can in principle simulate quantum processes much more efficiently [3]. Such computation can be represented on a classical computer as a series of matrix and vector operations [4]. Given n qubits, the data is a vector, or *state* vector, with 2^n complex numbers. Operations on these qubits can be simulated through matrix-vector multiplications by square unitary matrices, also called *operators* with 2^{2n} complex numbers. At a circuit level, qubits are analogous to bits, and operator matrices are analogous to logic gates. Given an n-qubit and k-qubit state vector or operator, an $n+k$-qubit state vector or operator can be constructed via the tensor (Kronecker) product. In this work we assume only a minimal knowledge of quantum computing and abstract computational problems in terms of linear algebra.

Since both the state vectors and operators grow exponentially in n qubits, it is computationally difficult for classical computers to simulate these systems. Such simulation, however, can be extremely useful to physicists who want to simulate quantum errors in their quantum computer designs and to computer scientists who want to devise new quantum algorithms. Equally interesting would be to use classical simulation to explore the *limitations* of quantum computing. Classical computers after all do not have to deal with quantum errors and, as we demonstrate, can simulate some instances of quantum computation in polynomial time and memory. The purpose of this paper is to demonstrate a novel approach to more efficient simulation of quantum computing based on well-known graph techniques from classical CAD. The next section describes previous attempts at better classical simulation, followed by Section III which presents our approach. Lastly, we offer experimental verification of our techniques in Section IV and conclude with final remarks in Section V.

II. Previous Work

Traditional array-based representations are often insensitive to the actual values stored, and even sparse matrix storage offers little improvement for quantum operators with no zero matrix elements (e.g. Hadamard operators). Gottesman [9] identified a number of special-case quantum circuits for which tailor-made simulation techniques require only polynomial memory and runtime. However, he noted that these "restricted types of quantum circuits fall short of the full power of quantum computation." Thus, in cases of major interest, such as Shor's and Grover's algorithms, quantum simulation is still performed with straightforward linear-algebraic tools and requires astronomic resources.

A number of "programming environments" for quantum computing were proposed recently that are mostly front-ends to quantum circuit simulators. Their back-ends typically use linear algebra methods to multiply matrices and require super-polynomial computational resources in the number of qubits. The potential benefits of efficient linear-algebraic operations on compressed arguments are immense.

Our approach uses graph-based techniques from classical CAD to improve asymptotic time and memory complexity of quantum simulations by exploiting the structure of quantum operators. Although abstract worst-case complexity is still exponential, our approach achieves very substantial performance gains in many important cases.

Other advanced simulation techniques, e.g., MATLAB's "packed" representation, include data compression, but often must decompress the operands of matrix-vector multiplication. A notable exception is Greve's simulation [7] of Shor's algorithm that uses *Binary Decision Diagrams* (BDDs) [2]. Probability amplitudes of individual qubits are modeled by single decision nodes. This only captures superpositions where every participating qubit is rotated by ± 45 degrees from $|0\rangle$ towards $|1\rangle$. Another BDD-based technique was recently proposed by Al-Rabadi et al. [8] which can perform multi-valued quantum logic. The drawback of this technique is that it too is incapable of capturing all types of quantum computation and instead emphasizes synthesis of quantum circuits rather than computational simulation. Further, there are no documented implementations or theoretical performance measures of Al-Rabadi et al.'s technique, making experimentation with this model difficult.

Though Greve's and Al-Rabadi et al.'s BDD representations cannot simulate the computation of arbitrary quantum circuits, the idea of modeling quantum states with a BDD-*like* structure is appealing and motivates our approach. However, unlike these previous models, our approach *is* capable of simulating arbitrary quantum circuits while offering performance improvements as demonstrated by our C++ implementation and experimental results described in Section IV.

III. QuIDD Theory

The *Quantum Information Decision Diagram* (QuIDD) was born out of the observation that vectors and matrices which arise in quantum computing exhibit a lot of structure. More complex operators obtained from the tensor product of simpler matrices continue to exhibit these common substructures which BDDs can capture. BDDs and operations for manipulating them were originally developed by Lee [1] and extended to Reduced Ordered BDDs (ROBDDs) by Bryant [2] to handle large Boolean functions efficiently. An ROBDD is a *directed acyclic graph* (DAG) with up to two outgoing edges per node, labeled *then* and *else*. Algorithms that perform operations on ROBDDs are typically recursive traversals. While not improving worst-case asymptotics, in practice ROBDDs achieve exponential space compression and runtime improvements by exploiting various types of structure found in practical applications.

Beyond the domain of digital logic design, ROBDD variants have been adopted in many contexts. Multi-Terminal Binary Decision Diagrams (MTBDDs) [5] and Algebraic Decision Diagrams (ADDs) [6] with their integrated linear-algebraic operations are particularly relevant to the task of simulating quantum systems. Both MTBDDs and ADDs compress explicit representations of matrices and vectors, with the amount of compression achieved being proportional to the frequency of repeated values in a given matrix or vector.

We have developed a further refinement, the QuIDD, to compress complex-valued matrices and vectors and operate on them in compressed form. The differences between QuIDDs and ADDs/MTBDDs are four-fold:

1. QuIDD terminals are restricted to the set of complex numbers

2. To facilitate efficient operations on the terminals, such as multiplication by a scalar, QuIDD terminals represent indices which index into a separate array of complex numbers containing the terminal values. This eliminates the need for an *Apply1* operation that must traverse the entire graph before performing operations on the terminals (see Subsection III.C and Fig. 1)

3. The variable ordering of QuIDDs interleaves row and column variables, which favors compression of block patterns (see Subsection III.B)

4. Bahar et al. note that ADDs can be padded with 0's to represent arbitrarily sized matrices [6]; no such padding is necessary in the quantum domain where all vectors and matrices have sizes that are a power of 2 (see Subsection III.B)

Although these optimizations may enhance ADD/MTBDD performance in other applications, we find that they are particularly useful in the quantum domain. A major contribution of this work is the observation that ADD/MTBDD-based structures with the properties listed above greatly enhance performance of quantum computational simulation. Our C++ implementation, QuIDDPro, is based on the CUDD package [13] and uses ADDs with these properties, though in principle it can also use MTBDDs. Space and time complexities of our simulations of n-qubit systems range from $O(1)$ to $O(2^n)$, but the worst case is not typical.

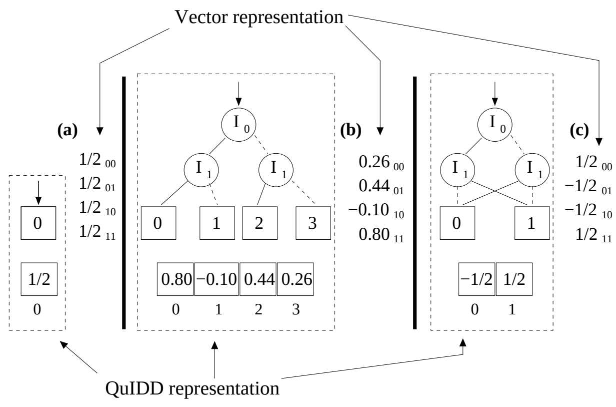

Fig. 1. Sample QuIDDs of (a) best, (b) worst and (c) mid-range size.

A. Vectors and Matrices

Figure 1 shows an abstract view of the QuIDD structure for three 2-qubit states. We consider the indices of the four vector elements to be binary numbers, and define their bits as decision variables of QuIDDs. A similar definition is used in ADDs [6]. For example, traversing the *then* edge (solid line) of node I_0 in Figure 1c is equivalent to assigning the value 1 to the first bit of the 2-bit vector index. Traversing the *else* edge (dotted line) of node I_1 in the same figure is equivalent to assigning the value 0 to the second bit of the index. These traversals bring us to the terminal value $-\frac{1}{2}$, which is precisely the value at index 10 in the vector representation.

QuIDD representations of matrices extend those of vectors by adding a second type of variable node and enjoy the same reduction rules and compression benefits. Consider the 2-qubit Hadamard matrix annotated with binary row and column indices shown in Figure 2a. In this case there are two sets of indices: The first (vertical) set corresponds to the rows, while the second (horizontal) set corresponds to the columns. We assign the variable name R_i and C_i to the row and column index variables respectively. This distinction between the two sets of variables was originally noted in several works including that of Bahar et al [6]. Figure 2b shows the QuIDD form of this sample matrix where it is used to modify the state vector $|00\rangle = (1,0,0,0)$ via matrix-vector multiplication.

B. Variable Ordering

Variable ordering can drastically affect the level of compression achieved in BDD-based structures such as QuIDDs. The CUDD package offers sophisticated dynamic variable-reordering techniques, such as shifting, that are typically greedy in nature, but achieve significant improvements in various BDD applications. However, dynamic variable reordering has significant time overhead, whereas finding a good static ordering in advance may be preferable in some cases. Good variable orderings are highly dependent upon the structure of the problem at hand, and therefore one way to seek out a good ordering is to study the problem domain. In the case of quantum computing, we notice that all matrices and vectors contain 2^n elements where n is the number of qubits represented. Additionally, the matrices are square and non-singular [4]. As noted in the work of Bahar et al., matrices and vectors that do not have sizes which are a power of two require padding with zeros [6], which can complicate real implementations. Fortunately, no such padding is required in quantum computing applications.

Hachtel et al. demonstrated that ADDs representing non-singular matrices can be operated on efficiently with interleaved row and column variables [12]. Interleaving implies the following variable ordering: $R_0 \prec C_0 \prec R_1 \prec C_1 \prec \ldots \prec R_n \prec C_n$. Intuitively, the interleaved ordering causes compression to favor regularity in block sub-structures of the matrices. Such regularity is created by tensor products that are required to allow multiple quantum gates to operate in parallel and also to extend smaller quantum gates to operate on larger numbers of qubits. The tensor product $A \otimes B$ multiplies each element of A by the whole matrix B to create a larger matrix which has dimensions $M_A \cdot M_B$ by $N_A \cdot N_B$. By definition, the tensor product will propagate patterns in its operands. As a result, the static interleaved variable ordering property of QuIDDs scales quite nicely as the number of qubits in the circuit increases, offering an efficient and robust representation for quantum circuit operators and state vectors.

C. QuIDD Operations

Most operations defined for ADDs also work on QuIDDs with only slight modification. A key example is matrix multiplication, which is an extension of the dot-product operation and implemented as a recursive procedure adapted from the well-known *BDD-Apply* function [2]. Matrix multiplication operations with ADDs are treated as *quasi-rings* which, among other properties, means that they have some operator $\flat$ which distributes over some commutative operator $\sharp$ [6]. This property is critical for computing the dot-products required in matrix multiplication, where terminal values are multiplied ($\flat$) to produce products that are then added ($\sharp$) to create the new terminal values of the resulting matrix. Another important issue in matrix multiplication is compression. To avoid the same problem that MATLAB encounters with its "pack" representation, ADDs must not be decompressed when performing this operation. In the work of Bahar et al., this is addressed by tracking the number i of "skipped" variables between the parent (caller) and its newly expanded child in each recursive call. A factor of 2^i is multiplied by the terminal-terminal product that is reached at the end of a recursive traversal [6]. For more details and pseudocode, see our technical report [10]. An example circuit with the QuIDD representation of its computational behavior is shown in Figure 3.

Tensor products can be implemented as follows. As noted in Subsection B, $A \otimes B$ multiplies each element of A by the entire matrix (or vector) B producing a new matrix (or vector). Multiplication of the terminal values is done by first shifting the variable numbers in B after those in A followed by a call to the recursive *Apply* function with an argument that directs *Apply* to multiply when it reaches the terminals of both operands [10]. Since QuIDD operations are variants of *Apply*, they have

$$
R_0R_1 \quad \begin{array}{c|cccc}
 & 00 & 01 & 10 & 11 \\
\hline
00 & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\
01 & \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \\
10 & \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\
11 & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2}
\end{array}
$$

C_0C_1

(a)

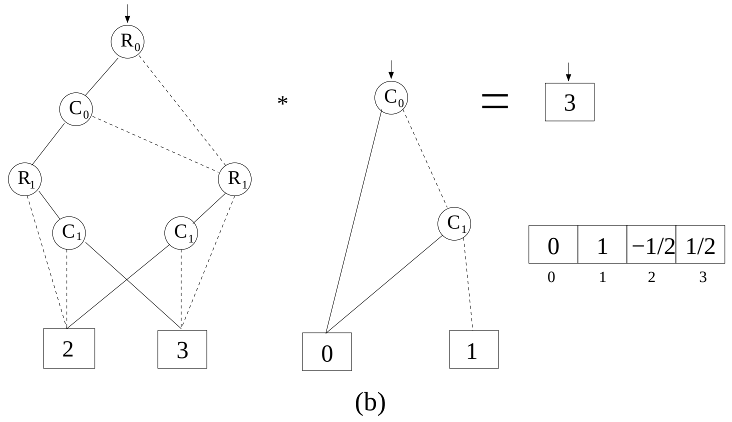

Fig. 2. (a) 2-qubit Hadamard, and (b) its QuIDD representation multiplied by $|00\rangle = (1,0,0,0)$. Note that the vector and matrix QuIDDs share the entries in a terminal array that is global to the computation.

complexity $O(|A| \cdot |B|)$, which is polynomial in the number of qubits if both $|A|$ and $|B|$ are.

Other key operations include quantum measurement and matrix addition. Measurement can be expressed in terms of matrix multiplication, tensor products, and scalar division by a normalization factor (a case in which QuIDD property 2 is useful), and so its QuIDD implementation is merely a combination of the operations described above. Matrix addition is easily implemented by calling *Apply* with an argument directing it to add the terminals of the operands. Unlike the tensor product, no special variable ordering shifting is required for matrix addition.

QuIDD properties imply additional implementation details such as the use of complex arithmetic. Abstract ADDs can support terminals of any numerical type, but CUDD's implementation of ADDs does not. For efficiency reasons, CUDD stores node information in C *union*s which are interpreted numerically for terminals and as child pointers for internal nodes. However, it is well-known that unions are incompatible with the use of C++ classes because their multiple interpretations hinder the binding of correct destructors. In particular, complex numbers in C++ are implemented as a templated class and are incompatible with CUDD. This was the original motivation for storing terminal values in an external array, but it also provides the added benefit of efficient scalar operations (QuIDD property 2).

Another important issue is the precision of complex numeric types. Over the course of repeated multiplications, the values of some terminals may become very small and induce round-off errors if the standard IEEE double precision floating-point types are used. This effect worsens for larger circuits. Unfortunately, such round-off errors can significantly affect the structure of a QuIDD by merging terminals that are only slightly different or not merging terminals whose values should be equal, but differ by a small computational error. The use of approximate comparisons with an epsilon works in certain cases but does not scale well, particularly for creating an equal superposition of states (a standard operation in quantum circuits). In an equal superposition, a circuit with n qubits will contain the terminal value $\frac{1}{2^{n/2}}$ in the state vector. With the IEEE double precision floating-point type, this value will be rounded to 0 at $n = 2048$ regardless of any epsilon. Furthermore, a static value for epsilon will not work well for different sized circuits (e.g. an epsilon of $1e-6$ may work well for $n = 35$, but not for $n = 40$). Therefore, to address the problem of precision, QuIDDPro uses an arbitrary precision floating-point type from the GMP library [11] with the C++ complex template. Precision is then limited to the available amount of memory in the system.

QuIDD property 3 is directly supported by CUDD and only requires some extra implementation overhead to enforce the variable ordering on the resultant QuIDD of any operation [10]. Lastly, QuIDD property 4 simplifies our QuIDDPro implementation since we do not need to handle matrices and vectors of arbitrary size.

IV. Simulating Grover's Algorithm

We used QuIDDPro in a few types of quantum computing simulations to verify that QuIDDs provide a useful amount of compression in practical applications. One such simulation involves running Grover's algorithm [14] for identification of keys in an unstructured database. Given an unstructured database of N elements, search for a single key matching a given search criterion, using classical computation resources, takes $O(N)$ time owing to the fact that at least half the elements must be examined on average. Grover's algorithm is a quantum algorithm that can locate the key using only $O(\sqrt{N})$ queries to an oracle that judges a match with the search criterion. A circuit implementing Grover's algorithm is shown in Fig. 4. The algorithm can be summarized as follows:

Let N denote the number of elements in the database.

1. Initialize $n = \lceil \log_2 N \rceil$ qubits to $|0\rangle$ and the *oracle qubit* to $|1\rangle$.

2. Apply the Hadamard transform (H gate) to all qubits to put them into a uniform superposition of basis states.

3. Apply the oracle. This sub-circuit performs a special form of the controlled-NOT (CNOT) gate. If the input to this cir-

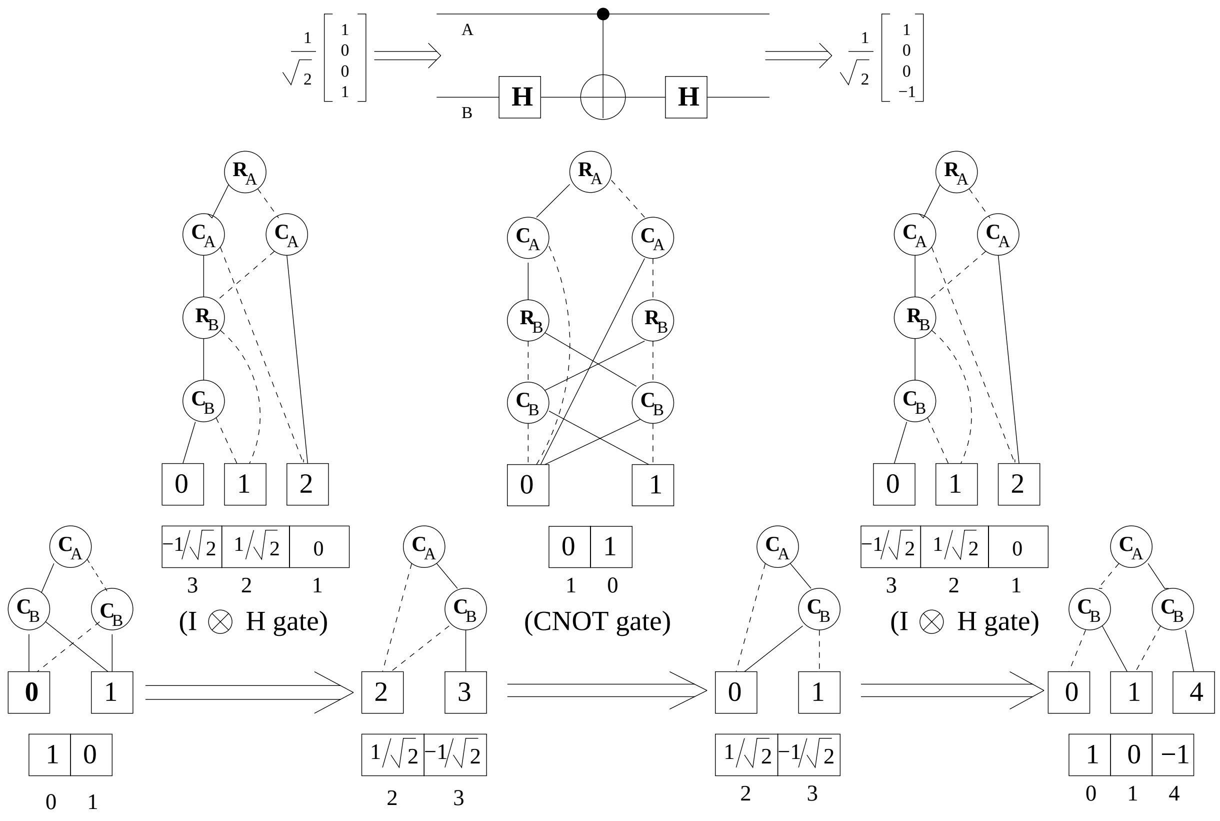
Fig. 3. Example quantum circuit with its QuIDD representation. The top row of QuIDDs represents the quantum operators $I \otimes H$ (a Hadamard acting on qubit B alone), $CNOT$, and another $I \otimes H$, respectively. The bottom row of QuIDDs represents the state vector form of qubits A and B before and after each operator acts on them. The terminal array is broken up with portions of it displayed under each QuIDD for visual clarity.

TABLE I
SIZE OF QUIDDS (# OF NODES) FOR GROVER'S ALGORITHM.

Circuit	Hadamards		Conditional	Oracles	
Size n	Initial	Repeated	Phase Shift	1	2
20	80	83	21	99	108
40	160	163	41	199	228
60	240	243	61	299	348
80	320	323	81	399	468
100	400	403	101	499	588

cuit satisfies the search criterion, the state of the oracle bit is flipped. For a superposition of inputs, those input basis states that satisfy the search criterion will induce a bit flip (X gate) on the oracle qubit in the composite state space. The oracle uses ancillary qubits as its workspace, reversibly returning them to their original states (shown as $|0\rangle$ in Fig 4). These ancillary qubits will not be operated on by any other step in the algorithm.

4. Apply the H gate to all qubits except the oracle qubit.

5. Apply the *Conditional Phase-Shift* gate on all qubits except the oracle qubit. This gate negates the probability amplitude of the $|000\ldots0\rangle$ basis state, leaving that of the others unaffected. It can be realized using a combination of X, H and C^{n-1}-NOT gates as shown. A decomposition of the C^{n-1}-NOT into elementary gates is given in [17].

6. Apply the H gate to all gates except the oracle qubit.

7. Repeat steps 3-6 R times, where $R = \lfloor \frac{\pi}{4}\sqrt{\frac{N}{M}} \rfloor$ and M is the number of keys matching the search criterion [16].

8. Apply the H gate to the oracle qubit in the last iteration. Measure the first n qubits to obtain the index of the matching key with high probability.

Before starting simulation, we construct the QuIDD representations of Hadamard operators by incrementally tensoring together one-qubit versions of their matrices $n-1$ times to get n-qubit versions. Similarly, we construct all other necessary operators. Table I shows sizes (in nodes) of respective QuIDDs at n-qubits, where $n = 20..100$. We observe that memory usage grows linearly in n and QuIDD-based simulations of Grover's algorithm are not memory-limited even at 100 qubits [10].

With the operators constructed, simulation can proceed. Tables IIa and IIb show performance measurements for simulating Grover's algorithm with an oracle that searches for one item out of 2^n. QuIDDPro achieves asymptotic memory savings compared to known interpreted and compiled numerical analysis packages. The overall runtimes are still exponential in n because Grover's algorithm entails an exponential number of iterations, even on an actual quantum computer [16]. We also studied a "mod-1024" oracle [10] that searches for elements whose ten least significant bits are 1. Memory us-

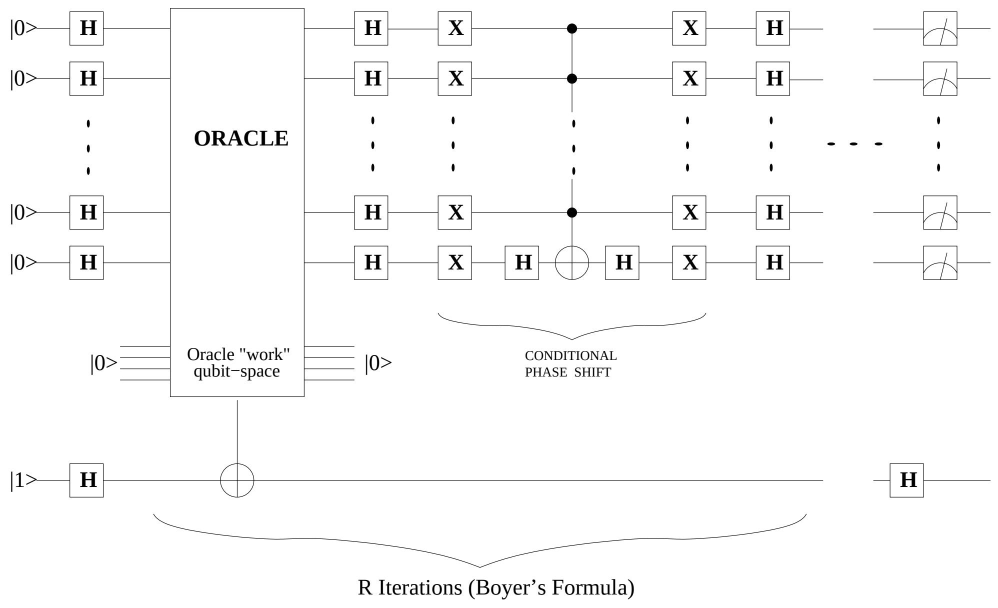

Fig. 4. Circuit-level implementation of Grover's algorithm

age for the mod-1024 oracle for up to $n = 25$ qubits grew as $(7.592 + 0.041n)$ (see Tables IIc and IId). Results were produced on a 1.2GHz AMD Athlon with 1GB RAM running Linux. Memory usage for MAT and Oct is lower-bounded by the size of the state vector and conditional phase shift operator; B++ and QP memory usage is measured as the size of the entire program. Simulations using MAT and Oct past 15 qubits timed out at 20 hours. See our technical report [10] for additional details.

V. Conclusions and Future Work

We proposed and tested a new technique for simulating quantum circuits using a data structure called a QuIDD. We have shown that QuIDDs enable practical, generic and reasonably efficient simulation of quantum computation. Their key advantages are faster execution and lower memory usage. In our experiments, QuIDDPro achieves exponential memory savings compared to other known techniques. We are currently studying the growth of required precision and simulation of other quantum algorithms, such as Shor's [15], where we plan to introduce the effects of errors and decoherence. Error simulation is important if one plans to model actual quantum computational devices, and we suspect it may prove to be a difficult problem since errors may cause some of the symmetries exploited by QuIDDs to be lost.

Additionally, our on-going work addresses the *limitations* of quantum computing by means of a "competition" between classical and quantum computers. Classical computers have the advantage that they are not subject to quantum measurement and errors. Thus, when competing with quantum computers, classical computers can simply run ideal error-free quantum algorithms (as we did in Section IV), allowing techniques such as QuIDDs to exploit the symmetries found in ideal quantum computation. To this end we are also attempting to characterize quantum gates that can be represented by polynomial-sized QuIDDs and thus facilitate fast error-free simulation.

Acknowledgments. This work was partially supported by the DARPA QuIST program. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing official policies or endorsements of funding agencies.

References

[1] C.Y. Lee, "Representation of switching circuits by binary decision diagrams," *Bell System Tech. J.*, Vol. 38, pp. 985-999, 1959.

[2] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. on Computers*, Vol. C35, pp. 677-691, Aug 1986.

[3] A. J. G. Hey, ed., *Feynman and Computation: Exploring the Limits of Computers*, Perseus Books, 1999, 438 pp.

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

[5] E. Clarke et al., "Multi-terminal binary decision diagrams and hybrid decision diagrams," in T. Sasao and M. Fujita, eds, *Representations of Discrete Functions*, pp. 93-108, Kluwer, 1996.

[6] R. I. Bahar et al., "Algebraic decision diagrams and their applications," *In Proc. IEEE/ACM ICCAD*, 188-191, 1993.

[7] D. Greve, "QDD: a quantum computer emulation library," 1999 `http://home.plutonium.net/~dagreve/qdd.html`

[8] A. N. Al-Rabadi et al., "Multiple-Valued Quantum Logic," *11th International Workshop on Post Binary ULSI*, Boston, MA., May 2002.

TABLE II
SIMULATING GROVER'S ALGORITHM WITH n QUBITS USING OCTAVE (OCT), MATLAB (MAT), BLITZ++ (B++) AND OUR SIMULATOR QuIDDPRO (QP).

Oracle 1: Runtime (s)				
n	Oct	MAT	B++	QP
10	89.4	14.0	0.22	0.20
11	2.94e2	45.9	0.72	0.39
12	9.26e2	1.53e2	2.22	0.88
13	3.09e3	5.80e2	6.92	1.94
14	1.36e4	5.90e3	23.09	4.79
15	7.10e4	5.92e4	70.4	9.32
16	TIME-OUT	TIME-OUT	2.13e2	22.2
17	TIME-OUT	TIME-OUT	6.34e2	50.7
18	TIME-OUT	TIME-OUT	1.92e3	1.13e2
19	TIME-OUT	TIME-OUT	5.74e3	2.00e2
20	TIME-OUT	TIME-OUT	1.74e4	3.25e2

(a)

Oracle 1: Peak Memory Usage (MB)				
n	Oct	MAT	B++	QP
10	3.60e-2	2.00e-2	1.95e-2	0.211
11	6.80e-2	4.40e-2	7.03e-2	0.207
12	0.132	9.20e-2	7.42e-2	0.281
13	0.260	0.188	0.129	0.426
14	0.268	0.264	0.250	0.444
15	0.524	0.520	0.500	0.605
16	1.04	1.03	1.00	0.840
17	2.06	2.06	2.00	0.965
18	4.11	4.10	4.00	1.59
19	8.20	8.20	8.00	1.77
20	16.4	16.4	16.0	2.04

(b)

Oracle 2: Runtime (s)				
n	Oct	MAT	B++	QP
12	5.45e2	79.3	1.40	0.450
13	1.28e3	1.95e2	3.10	0.500
14	3.85e3	8.27e2	6.90	0.570
15	1.23e4	4.67e3	15.0	0.570
16	3.62e4	3.78e4	32.3	0.680

(c)

Oracle 2: Peak Memory Usage (MB)				
n	Oct	MAT	B++	QP
12	6.56e-2	4.12e-2	0.203	0.230
13	0.131	8.22e-2	0.324	0.313
14	0.148	0.164	0.574	0.336
15	0.295	0.328	1.07	0.371
16	0.590	0.656	2.07	0.402

(d)

[9] D. Gottesman, "The Heisenberg representation of quantum computers," *Plenary speech at the 1998 International Conference on Group Theoretic Methods in Physics*, **quant-ph/9807006**.

[10] G. Viamontes, M. Rajagopalan, I. Markov, J. Hayes, "Gate-level simulation of quantum circuits," *Los Alamos Quantum Physics Archive*, Aug. 2002 `http://xxx.lanl.gov/abs/quant-ph/0208003`

[11] "GNU MP (GMP): Arithmetic Without Limitations," `http://www.swox.com/gmp/`

[12] G. Hachtel et al., "Markovian analysis of large finite state machines," *IEEE Trans. on Computer-Aided Design*, Vol. 15, pp. 1479-1493, Dec. 1996.

[13] F. Somenzi, "CUDD: CU Decision Diagram Package," release 2.3.0, Univ. of Colorado at Boulder, 1998.

[14] L. Grover, "Quantum Mechanics Helps In Searching For A Needle In A Haystack," *Phys. Rev. Lett.* (79), pp. 325-8, 1997.

[15] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. of Computing*, Vol. 26, p. 1484, 1997.

[16] M. Boyer, G. Brassard, P. Hoyer and A. Tapp, "Tight bounds on quantum searching," *4th Workshop on Physics and Computation*, Nov. 1996.

[17] A. Barenco et al., "Elementary gates for quantum computation", *Los Alamos Quantum Physics Archive*, **quant-ph/9503016**, Mar. 1995.