Stochastic Optimization of Quantum Programs

Peng Liu, Shaohan Hu, Marco Pistoia, Chun-Fu (Richard) Chen, and Jay M. Gambetta,
IBM T.J. Watson Research Center

We introduce Markov chain Monte Carlo quantum (MCMCQ), a novel compiler-level optimization of quantum programs that accounts for the emerging quantum programming mode. MCMCQ is the first systematic approach for stochastic quantum-program optimization, targeting program performance, correctness, and noise tolerance. An evaluation of MCMCQ over 500 quantum programs confirms its effectiveness.

Tech giants[1–4] are investing heavily in quantum computing. To interact with the quantum hardware, users write quantum programs in a high-level language. Such programs are translated to a sequence of hardware instructions. Quantum programming has a unique programming model underpinned by the quantum hardware: Unlike a classical bit, which exclusively represents either 0 or 1, a *qubit* may probabilistically represent both 0 and 1 simultaneously. A *quantum instruction* modifies the probability information carried by the qubit.

Digital Object Identifier 10.1109/MC.2019.2909711
Date of publication: 4 June 2019

THE PROBLEM

In this article, we propose a compiler technique applicable to quantum programs that improves performance through program simplification while guaranteeing correctness. More importantly, the near-term quantum hardware suffers from noise and has a very short decoherence time, thereby leading to unreliable results if the execution of a

 0018-9162/19©2019IEEE

circuit takes too long. The technique we propose mitigates this problem by reducing the number of gates and shortening the execution time.

Existing work in quantum-program optimization

We classify existing work for quantum-program optimization into two categories.

1. *Rule-based techniques* rely on heuristics-based specifications, or rules, defined by human experts. Numerous existing approaches,[5–8] including recent ones,[9] are rule based. Although these techniques can be quite effective, they demand great manual efforts. Furthermore, human experts may miss optimization opportunities. More importantly, these techniques are required to preserve correctness at every step. As such, they are forced to exclude aggressive optimization sequences,[10–12] which ensure correctness at the final step but also allow correctness to be temporarily sacrificed at intermediate steps. For this reason, important opportunities for optimization may be missed.
2. *Systematic techniques* leverage systematic algorithms, such as genetic algorithms and discrete/continuous optimization, to achieve circuit optimization without getting human experts involved.[13,14] Systematic solutions complement the rule-based ones because they may find optimization opportunities missed by the experts. While inspiring, systematic search procedures are not guided by the quality of the circuits, where by *quality* we mean performance, correctness, and noise tolerance. For instance, the genetic algorithm in Williams and Gray[13] is directed toward maximizing population diversity, as opposed to quality. For this reason, the existing systematic approaches are not very effective at quantum-circuit optimization.

Novel contributions of this work

In this article, we propose MCMCQ, the first quality-driven systematic optimizer of quantum programs. It is built upon stochastic optimization,[10,11] which has achieved success in classical optimization. An important contribution of MCMCQ is that it makes stochastic optimization applicable to quantum programs by accounting for the unique quantum programming model. For example, in contrast to classical computing, a quantum program state represents a probability distribution. Therefore, the execution semantics of quantum programs are drastically distinct from the semantics of classical programs. Consequently, compiler analysis must account for such differences.

Stochastic optimization[10,11] is a search-based technique that, from the original program, randomly generates a mutant and stochastically accepts or rejects that mutant with probability determined by the Markov chain Monte Carlo (MCMC) theory (see the "Theoretical Stochastic Optimization Framework" section). If a mutant is accepted, the search continues from it instead of the original program. This process is iteratively applied. MCMC guarantees that a mutant is visited with probability exponentially proportional to its quality—a mutant with higher quality is visited with higher probability (see the "Applying MCMC to Optimize Quantum Programs" section).

We observe that prior work[10] confirms the correctness of a mutant by checking whether it produces the same outputs as the original program for a set of inputs. However, the notion of *output* in quantum computing is drastically different from its classical counterpart and cannot be used for checking correctness. In particular, the program state of a quantum program represents a probability distribution of all possible outcomes, and the observed output is a sample from the distribution, which varies from run to run nondeterministically and hence cannot be used for correctness checking. To address these challenges, we designed MCMCQ fully accounting for all the aspects of the emerging quantum programming model and execution semantics and carefully examining the full stack, ranging from the hardware abstractions consisting of qubits, gates, and circuits, to the high-level notions of correctness, performance, and noise tolerance. Thus, our approach is not affected by the discrepancy caused by the probabilistic nature of a quantum program's output. Theorem 2 formally proves that MCMCQ guarantees correctness for any input as long as correctness can be established for a finite number of specific inputs.

We applied the technique to optimize an implementation of Grover's search algorithm thoroughly engineered by experts, for which MCMCQ found 1,989 optimized versions. In one of them, as shown in Figure 1(b), MCMCQ was able to optimize the original program by reducing it by five gates and three steps, evidence that the approach can discover implementations more efficient than those provided manually by experts.

Furthermore, a large-scale evaluation over 500 quantum programs confirms that MCMCQ effectively improves performance while preserving correctness.

This article makes the following contributions:

1. the first quality-driven systematic optimizer for quantum programs
2. a novel correctness-checking and performance-assessment technique based on the quantum programming model
3. a full implementation of MCMCQ on top of Qiskit
4. a thorough evaluation of MCMCQ.

STOCHASTIC OPTIMIZATION OF QUANTUM PROGRAMS

In this section, we illustrate in detail how MCMCQ works. We first present a summary of the MCMC theoretical framework and then explain how we have applied MCMC to optimize quantum programs. Under the hood, this optimization requires correctness checking and performance modeling; both techniques must be specific to the quantum programming model. In particular, by leveraging the unique characteristic of quantum computing, we show that correctness can be verified efficiently by checking the outputs for only a finite set of inputs.

Theoretical stochastic optimization framework

In general, the MCMC algorithm constructs a Markov chain, in which each state represents an item in the search space. Each state x is also associated with a score v_x that indicates how good state x is. We explain how the score is assigned in the "Applying MCMC to Optimize Quantum Programs" section. For now, let us assume the score is known.

The goal of MCMC is to ensure that the constructed Markov chain exhibits the stationary distribution expressed in (1), which can be summarized as follows: The probability $\pi(x)$ of visiting state x is exponentially proportional to the score v_x associated with x. Intuitively, states with a higher score are more likely to be visited:

$$\pi(x) = \alpha e^{\beta * v_x}, \quad (1)$$

where $\alpha, \beta > 0$.

MCMC achieves the stationary distribution by controlling the transition probability. Theorem 1 lays the theoretical foundation.[11]

Theorem 1. The stationary distribution $\boldsymbol{\pi}$ over all states satisfies $\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{P}$, where $\mathbf{P}$ is the transition probability matrix. A Markov chain has a stationary distribution if the following conditions are met:

- *Existence*: A sufficient but not necessary condition is the *detailed balance* condition: $\pi(x)P(x' \mid x) = \pi(x')P(x \mid x')$, where $P(x' \mid x)$ denotes the probability of the transition from state x to state x'.
- *Uniqueness*: The uniqueness is guaranteed by the ergodicity: every state is aperiodic and positive recurrent.

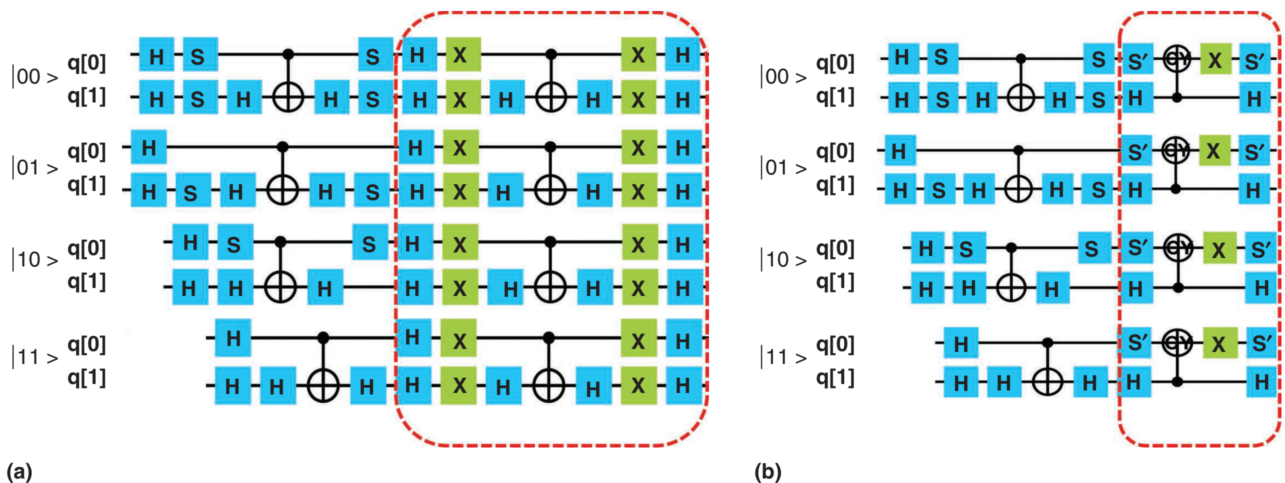


FIGURE 1. (a) The original amplification component used in Grover's search algorithm. (b) The optimized amplification.

A simple yet common way[11] to satisfy the conditions in Theorem 1 is to define $P(x' \mid x)$ as

$$P(x' \mid x) = \min\{1, e^{\beta*(v_{x'} - v_x)}\}. \quad (2)$$

Intuitively, MCMC always allows the transition from a state x to another state x' if it leads to an increase in the score, i.e., $v_{x'} - v_x > 0$. If the transition leads to a decrease in the score, e.g., $v_{x'} = 1$ and $v_x = 5$, MCMC accepts the transition in the probability $e^{\beta*(1-5)} = e^{-4\beta}$.

It is important to specify the parameter β properly. We adopt the popular *annealing* strategy to achieve a good tradeoff between exploration and exploitation. Initially, we adopt a very low β value (0.05 in our experiments) to encourage the acceptance of a mutant, even if it leads to a decrease in the score. Favoring exploration helps avoid getting stuck at the local optimum. Next, we gradually increase β (by 0.02 every 200 iterations) to restrict acceptance in favor of exploitation.

Applying MCMC to optimize quantum programs

Algorithm 1 shows our MCMC-based optimization strategy. The algorithm maintains the current mutant f, which corresponds to the state that is currently visited in the Markov chain. The value f is initialized as the original program f_o (line 1). In each iteration of the loop (lines 3–11), the algorithm creates a mutant f' from the current mutant f (line 3), following the mutation strategy in the "Quantum Program Mutation" section, and computes the difference of the scores (line 4), which determines the acceptance probability p_{accept} of f' (line 5).

With the help of the uniform distribution, the check at line 6 ensures line 7 is executed with probability p_{accept}, i.e., the mutant f' is accepted with p_{accept}. If f' is accepted, we update the current mutant f as f'. Additionally, if f' has a higher score than f and is correct with respect to the original program f_o, then f' is a target mutant and we save it to disk.

Score and cost. The score is an overall assessment of the correctness and performance of a mutant quantum program. Given a mutant f, the score is defined as follows:

$$\text{score}(f) = -\text{cost}(f)$$
$$\text{cost}(f) = \text{ratio} * \text{cost}_{\text{correct}}(f) + \text{cost}_{\text{perf}}(f).$$

Intuitively, the correctness cost $\text{cost}_{\text{correct}}$ measures how much the mutant behaves differently from the original program, whereas the performance cost $\text{cost}_{\text{perf}}$ measures the performance slowdown compared to the original program. We explain these concepts in subsequent sections.

Algorithm 1 uses the score to generally guide the search without distinguishing correct and incorrect mutants. This is because the incorrect mutants are often necessary intermediate states during the transition from one correct mutant to another correct mutant. However, the algorithm saves the correct mutants only (lines 8 and 9).

ALGORITHM 1. Program Optimization.

```
Input   : original program f_o
Input   : number of iterations N
1  f = f_o;
2  for i in range(N) do
3  |  f' = mutate(f);
4  |  d = score(f') - score(f);
5  |  p_accept = min(1, exp^(β*d));
6  |  if random.uniform(0,1) ≤ p_accept then
7  |  |  f = f';
8  |  |  if d > 0 and correct(f', f_o) then
9  |  |  |  save(f');
10 |  |  end
11 |  end
12 end
```

Proving correctness. As per line 8, a mutant is reported only if it is proven correct. Existing stochastic optimization techniques for classical programs check for correctness by verifying that the mutant produces the same outputs as the original program for a set of inputs. However, in quantum computing, the outputs observed during measurement are nondeterministic and cannot be used for correctness verification. We observe that a correct mutant should produce the same final system state as the original quantum program for all possible inputs. Accordingly, we customized our design to check for correctness by comparing the final system state of a mutant with that of the original program.

Another challenge is that it is infeasible to check for all possible input states because the input-state space is infinite. Fortunately, a unique characteristic of quantum states allows us to reduce the checking for all input states to the checking for a finite set of basis input states without loss of generality. Specifically, given that an arbitrary superposition input state is a linear combination of the basis states, the correctness checking for a superposition input state can be reduced to the checking for the basis input states. Theorem 2 formalizes this insight.

Theorem 2. Let M and M' be the matrices representing the original program (or subprogram) and its optimized version, respectively. Let $\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{N-1}$ be basis states, so that any superposition input state $\mathbf{q}$ can be uniquely written as a linear combination of them, as follows: $\mathbf{q} = \sum_{0 \le i < N} c_i \boldsymbol{b}_i$. Then $M\mathbf{q} = M'\mathbf{q} \Leftrightarrow M\boldsymbol{b}_i = M'\boldsymbol{b}_i, \forall i = 0, \ldots, N-1$.

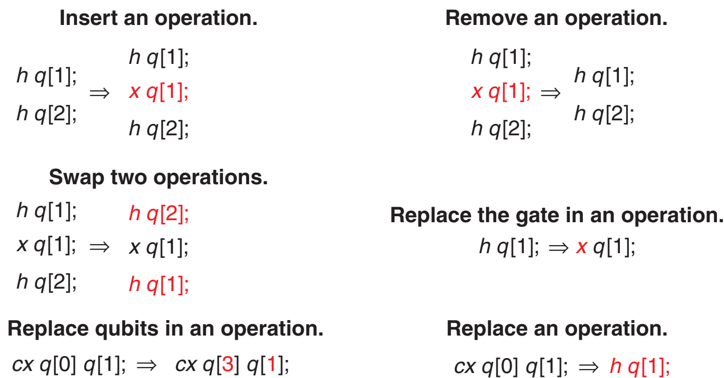


FIGURE 2. The six basic mutation strategies (with changes highlighted in red) for generating a mutant from an existing program.

Computing the correctness cost. Similarly, we compute the correctness cost by focusing on the final system states derived for the basis input states

$$\text{cost}_{\text{correct}}(f) = \frac{1}{N}\sum_{0 \le i < N} \text{diff}(fss(f, \mathbf{b}_i), fss(f_o, \mathbf{b}_i)).$$

Specifically, we first compute the difference of the final system states between the optimized version f and the original version f_o for each basis input state. We then average the difference over all the basis input states.

Let $\boldsymbol{o}_1$ and $\boldsymbol{o}_2$ stand for the final system states $fss(f, \mathbf{b}_i)$ and $fss(f_o, \mathbf{b}_i)$ prior to the measurement. Note that $\boldsymbol{o}_1$ and $\boldsymbol{o}_2$ are in amplitude-vector form. They may differ if they 1) contain distinct entry values or 2) place the same set of entry values in different orders. The *difference function* diff combines both types of differences. To account for the first type of difference, we ignore the orders of the entry values. We try to match each entry value in a vector to an entry with the same value in the other vector. Under the constraint that each value can be matched at most once, we count how many entry values remain unmatched. The second type of difference occurs if the two vectors arrange the same set of entry values in different orders. For this, we simply impose a constant penalty cost if two vectors are not equal to each other.

Computing the performance cost. Performance is a complex issue that depends on many factors, including both program features (such as the size of the program) and hardware conditions. However, because ours is a software-based approach, we aim at abstracting away any complexity caused by the hardware. Therefore, we adopt a simplified performance modeling based on the program features only.

One way of measuring the performance cost is to simply count the number of gates used under the assumption that every type of gate operation takes the same amount of time. In practice, multiple gates may be applied in parallel at the same step following the schedule algorithm.[15,16] Therefore, in this article, we use the number of steps to approximate the execution time more realistically. Accordingly, we simply leverage the quantum circuit scheduler[16] to compute the number of steps. In future work, we plan to apply different weights to different types of gates when the corresponding operations take different amounts of time.

Quantum program mutation

The mutate function in Algorithm 1 randomly picks a mutation strategy and applies it. Figure 2 shows six basic mutation strategies (with changes highlighted in red) for generating a mutant from an existing program.

1. Insert, at a random program point, an operation synthesized by randomly choosing gate and qubits.
2. Remove a randomly chosen operation.
3. Swap two randomly chosen operations.
4. Randomly choose an existing operation and replace the gate with a different one.
5. Randomly choose an existing operation and replace the qubits with other qubits.
6. Randomly choose an existing operation and replace it with an operation randomly synthesized.

EVALUATION

In this section, we evaluate MCMCQ via a set of experiments. In particular, we focus on MCMCQ's effectiveness and performance as well as the correctness of the optimized programs.

Benchmarks, methodology, and environment

We built MCMCQ upon the open-sourced Qiskit framework. We measured MCMCQ's effectiveness and performance by conducting experiments using 500 quantum programs randomly

sampled from the 3-SAT benchmark suite accompanying Qiskit.[17] On average, each program has 100 lines of code (100 operations). We did not choose the RevLib benchmark suite because the code could not be compiled by Qiskit.

Also, as an in-depth case study, we focused on a popular 2-qubit Grover algorithm implementation well engineered by experts. Any improvement induced by MCMCQ is, therefore, highly promising as it indicates MCMCQ's ability to discover correct variants more optimal than what experts could provide. All experiments were carried out on 25 machines in parallel, each equipped with four Intel Xeon E5-2683 CPUs and 16 GB of random-access memory.

Effectiveness

To demonstrate MCMCQ's effectiveness, we show 1) the dynamics of the cost throughout the optimization process and 2) the distribution of the cost at specific iteration timestamps. As explained in the "Applying MCMC to Optimize Quantum Programs," section, the cost combines costs for performance and correctness, which are not distinguished during optimization. The absolute value of the cost is not very important here. We are more concerned about the trend of the value change.

TO DEMONSTRATE MCMCQ'S EFFECTIVENESS, WE SHOW 1) THE DYNAMICS OF THE COST THROUGHOUT THE OPTIMIZATION PROCESS AND 2) THE DISTRIBUTION OF THE COST AT SPECIFIC ITERATION TIMESTAMPS.

Overall cost dynamics. The graphs in Figure 3 summarize the experiments on the 500 programs. Programs in the same group have similar curves. We were able to categorize the majority (90%) of the programs into eight

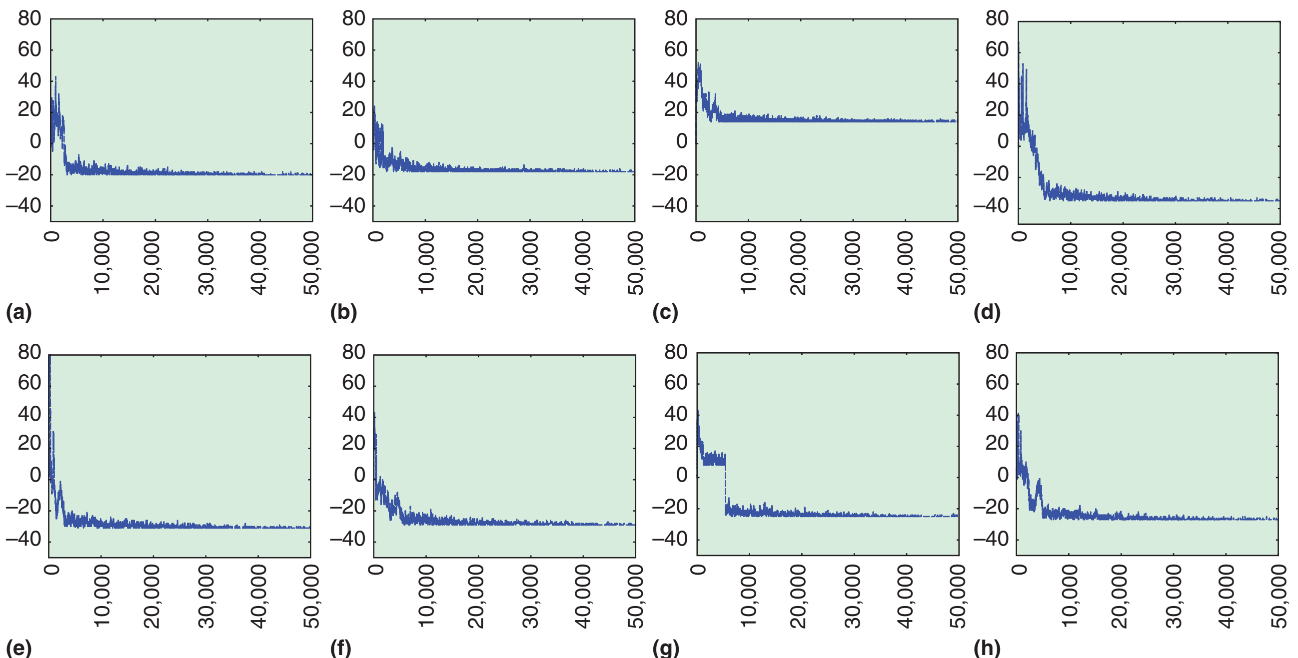


FIGURE 3. Eight representative graphs showing change in cost as the number of iterations increases. The cost is on the *y*-axis and the MCMC iterations on the *x*-axis. (a) Group 1. (b) Group 2. (c) Group 3. (d) Group 4. (e) Group 5. (f) Group 6. (g) Group 7. (h) Group 8.

groups. A representative graph is selected from each group. Each graph shows the dynamics of the cost (y-axis, ranging from −50 to 80) with respect to the MCMC iterations (x-axis, capped at 50,000 iterations).

Despite the differences in the curve shapes, we observe a common trend: cost decreases as the number of iterations increases, indicating that MCMCQ is effective in lowering the overall cost. Furthermore, the cost converges quickly, typically after only 5,000 iterations.

We also see that early iterations display a much higher cost variance than later ones. This is due to having applied simulated annealing to parameter β, as mentioned in the section "Applying MCMC to Optimize Quantum Programs." In particular, an initially small β makes MCMC more tolerant to the mutants with higher cost, thereby favoring exploration. In contrast, a larger β later on shifts the focus from exploration to exploitation.

Finally, we observe that the cost does not always converge to a negative value, as shown, for example, in Figure 3(c). This demonstrates a limitation of MCMC: getting stuck at local optima. Using a small β at the beginning mitigates this problem but does not eliminate it completely.

Cost dynamics: Correct mutants only. We now focus on the correct mutants only, whose correctness cost is zero, which implies that the overall cost is equal to the performance cost.

Figure 4 shows the cost dynamics of the correct mutants for the same programs shown in Figure 3. In Figure 4, parts (c), (e), and (g) are blank because no correct mutants are found throughout the 50,000 iterations. Overall, MCMCQ failed to find correct mutants in 35% of the programs.

Interestingly, almost every part in Figure 4 [except (g)] has a curve shaped similarly to its counterpart in Figure 3. This suggests that correct mutants are scattered, rather than clustered together. In other words, there are intermediate incorrect mutants during the transition between two correct mutants.

Performance and scalability

In our experiments, the number of qubits ranges from six to nine. The average running time taken by each iteration increases with the number of qubits, as can be seen in Figure 5(a). Overall, each iteration takes fewer

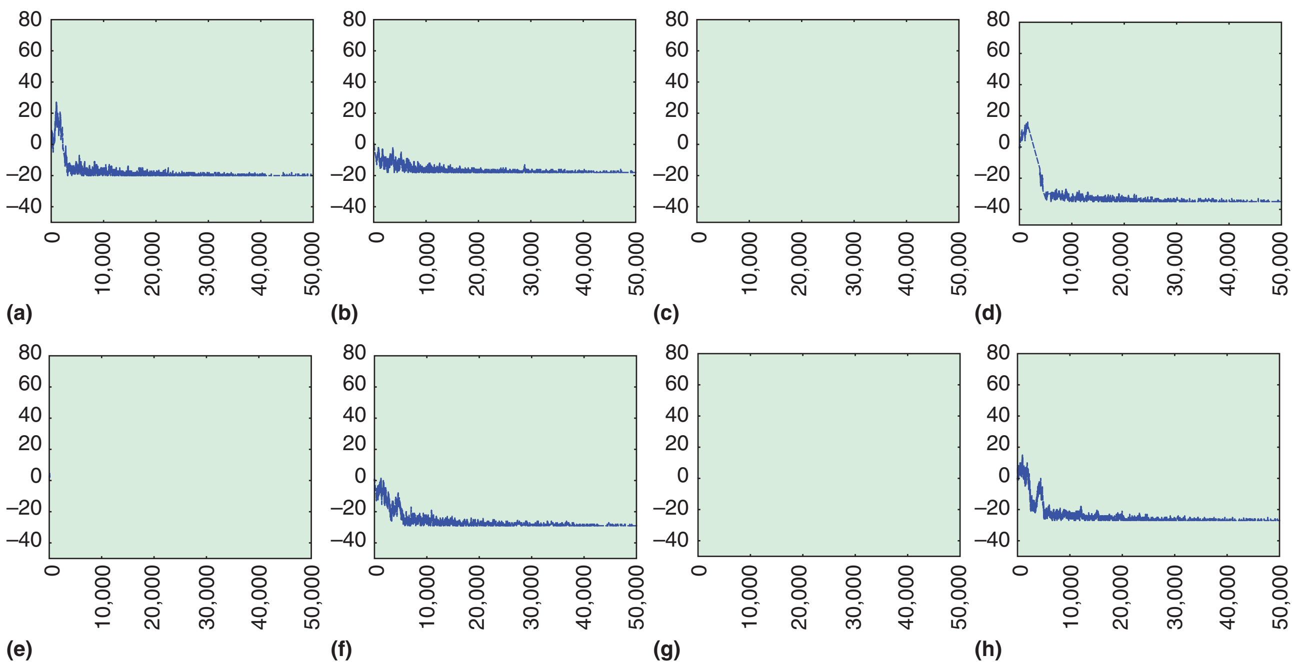


FIGURE 4. The change in the cost of correct mutants when the number of iterations increases for the same eight programs shown in Figure 3. The cost is on the y-axis and the MCMC iterations on the x-axis. (a) Group 1. (b) Group 2. (c) Group 3. (d) Group 4. (e) Group 5. (f) Group 6. (g) Group 7. (h) Group 8.

than 3 s. Figure 5(b) shows the time elapsed after every 5,000 iterations during the optimization of a 6-qubit program. The figure suggests that the running time is linearly proportional to the number of iterations. Also, the optimization of each program takes roughly 13 h. Therefore, optimizing all 500 programs takes 260 h (around 11 days) on the 25 machines.

MCMCQ does not scale well when the number of qubits involved in the quantum system increases. According to our experiments, when the system has more than 13 qubits, MCMCQ incurs out-of-memory errors within 1 h.

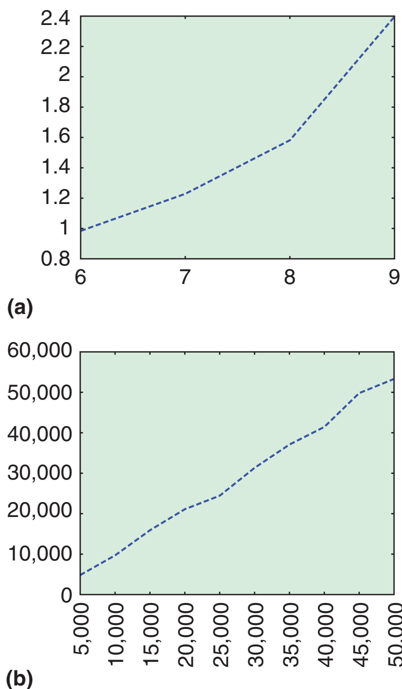


FIGURE 5. (a) Time taken by each iteration with different numbers of qubits. Time (s) is on the *y*-axis, and the number of qubits is on the *x*-axis. (b) Time taken by different numbers of iterations with 6 qubits. Time (s) is on the *y*-axis; MCMC iterations are on the *x*-axis.

This limited scalability is due to the size of the system state growing exponentially with the number of qubits. Despite this scalability issue, we believe MCMCQ is still very useful, provided that quantum programmers can select a subcircuit from the quantum circuit and apply MCMCQ to the subcircuit. Note that the selected subcircuit should not be entangled with the rest of the circuit, i.e., no CNOT gate should connect it to the rest of the circuit.

In our experiments, the programs have 100 operations on average. As shown in the previous section, we achieved good exploration of the mutant space with merely 5,000 iterations. In practice, programs are much longer, thereby demanding more iterations. Similar to what we observed above, we believe MCMCQ is still very useful, given that programmers can select a subcircuit from the quantum circuit and apply MCMCQ to optimize the subcircuit.

Correctness verification

In this section, we verify whether the optimized programs are strictly equivalent to the original versions. We adopt the 2-qubit Grover algorithm for brevity and clarity. In Figure 1(a), we show four Grover implementations for searching for $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$, respectively. As seen, all four implementations share the same amplification structure, which has seven steps and 11 gates in total.

By applying MCMCQ to optimize the amplification component, we saved 1,989 target mutants to disk throughout the entire optimization process. We show one of them in Figure 1(b), where the optimized amplification component is highlighted. Note that it uses the S dagger gate S' and the controlled-Y gate. In our work, the domain of the gates consists of a set of standard gates.

We verified that the optimized version can be used as a drop-in replacement for the original amplification component. In particular, the optimized amplification structure leads to the same final system state as the original amplification structure in all four cases, thereby correctly amplifying the marked states. Importantly, the optimized amplification component has only four steps and six gates, three steps and five gates fewer than the original.

Furthermore, we applied MCMCQ to other quantum-circuit components and achieved similar success. In particular, we adapted a circuit from Nam et al.[9] and applied MCMCQ to it. The original and optimized versions of the circuit are shown in Figure 6(a) and (b), respectively.

In this article, we presented MCMCQ, a stochastic compiler optimization technique applicable to quantum programs. A large-scale evaluation confirms the effectiveness of MCMCQ.

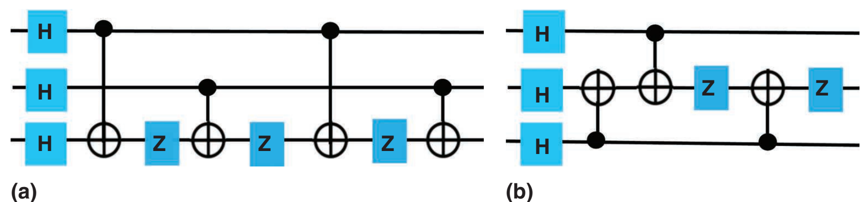


FIGURE 6. The (a) original and (b) optimized versions of a circuit to which MCMCQ was applied.

ABOUT THE AUTHORS

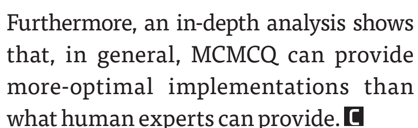
PENG LIU is a research staff member at the IBM T.J. Watson Research Center, Yorktown Heights, New York. He has published extensively in several areas of computer science, including compiler theory, artificial intelligence, static and dynamic program analysis, security, and quantum computing. Liu received a Ph.D. in computer science from the Hong Kong University of Science and Technology. Contact him at liup@us.ibm.com.

SHAOHAN HU is a research staff member at the IBM T.J. Watson Research Center, Yorktown Heights, New York. His research interests include quantum computing, cyberphysical systems, mobile ubiquitous computing, crowd and social sensing, big data analytics, and cloud computing. Hu received a Ph.D. in computer science from the University of Illinois at Urbana–Champaign. He is a Member of the IEEE. Contact him at shaohan.hu@ibm.com.

MARCO PISTOIA is a distinguished research staff member and senior manager at the IBM T.J. Watson Research Center in Yorktown Heights, New York. His research interests include linear algebra, invariant theory, and quantum computing. Pistoia received a Ph.D. in mathematics from New York University. He has authored or coauthored 10 books and more than 50 scholarly papers. For his publications, he has received four ACM distinguished paper awards and one IEEE honorable mention. Contact him at pistoia@us.ibm.com.

CHUN-FU (RICHARD) CHEN is a senior software engineer at the IBM T.J. Watson Research Center, New York. His research interests include quantum computing for chemistry, machine learning, and optimization as well as computer vision and graph computing. Chen received a M.S. from the Department of Electrical Engineering of the National Cheng Kung University, Taiwan. He is a Member of the IEEE and ACM. Contact him at chenrich@us.ibm.com.

JAY M. GAMBETTA is an IBM Fellow and the IBM global lead of quantum theory, applications, theory, and software at the IBM T.J. Watson Research Center, Cambridge, Massachusetts. Gambetta received a Ph.D. in physics from Griffith University, Brisbane, Australia. He has authored or coauthored more than 100 peer-reviewed articles in the field of quantum computing. Gambetta is a Senior Member of the IEEE and was a fellow of the American Physical Society. Contact him at jay.gambetta@us.ibm.com.



Furthermore, an in-depth analysis shows that, in general, MCMCQ can provide more-optimal implementations than what human experts can provide. ■

REFERENCES

1. Microsoft, "Quantum." Accessed on: Nov. 2017. [Online]. Available: https://www.microsoft.com/en-us/quantum
2. Google, "Quantum A.I." Accessed on: Nov. 2017. [Online]. Available: https://research.google.com/pubs/QuantumAI.html
3. IBM, "IBM Q." Accessed on: Nov. 2017. [Online]. Available: https://www.research.ibm.com/ibm-q/
4. Intel, "Quantum computing." Accessed on: Nov. 2017. [Online]. Available: https://newsroom.intel.com/press-kits/quantum-computing/
5. M. Curry, "Symbolic quantum circuit simplification in SymPy," 2011. Accessed on: Mar. 16, 2019. [Online]. Available: https://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1039&context=physsp
6. D. Maslov, G. W. Dueck, D. M. Miller, and C. Negrevergne, "Quantum circuit simplification and level compaction," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 27, no. 3, pp. 436–444, 2008. doi: 10.1109/TCAD.2007.911334.
7. D. Maslov, C. Young, D. M. Miller, and G. W. Dueck, "Quantum circuit simplification using templates," in *Proc. Design, Automation and Test in Europe Conf.*, 2005, vol. 2, pp. 1208–1213.
8. B. P. Lanyon et al., "Simplifying quantum logic using higher-dimensional Hilbert spaces," *Nature Physics*, vol. 5, no. 2, pp. 134–140, 2008.
9. Y. Nam, N. J. Ross, Y. Su, A. M. Childs, and D. Maslov, "Automated optimization of large quantum circuits with continuous parameters," *npj Quantum Inform.*, vol. 4, no. 1, p. 23, 2018.
10. E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *Proc. 18th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, 2013, pp. 305–316.
11. V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided

stochastic program mutation," in *Proc. 2015 ACM SIGPLAN Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*, pp. 386–399. doi: 10.1145/2814270.2814319.

12. Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of JVM implementations," in *Proc. 37th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '16)*, 2016, pp. 85–99.
13. C. P. Williams and A. G. Gray, "Automated design of quantum circuits," in *Quantum Computing and Quantum Communications*, C. P. Williams, Ed. Berlin: Springer-Verlag, 1999, pp. 113–125.
14. S. Khatri, R. LaRose, A. Poremba, L. Cincio, A. T. Sornborger, and P. J. Coles, Quantum-assisted quantum compiling. 2018. [Online]. Available: https://arxiv.org/abs/1807.00800
15. J. Heckey et al., "Compiler management of communication and parallelism for quantum computation," in *Proc. 20th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, 2015, pp. 445–456. doi: 10.1145/2694344.2694357.
16. G. Aleksandrowicz et al., "Qiskit: An open-source framework for quantum computing," 2019. doi: 10.5281/zenodo.2562110.
17. IBMQ, "Benchmarks," Accessed on: Nov. 2017. [Online]. Available: https://sites.google.com/site/qbenchmarks/

IT Professional

TECHNOLOGY SOLUTIONS FOR THE ENTERPRISE

CALL FOR ARTICLES

IT Professional seeks original submissions on technology solutions for the enterprise. Topics include

- emerging technologies,
- cloud computing,
- Web 2.0 and services,
- cybersecurity,
- mobile computing,
- green IT,
- RFID,
- social software,
- data management and mining,
- systems integration,
- communication networks,
- datacenter operations,
- IT asset management, and
- health information technology.

We welcome articles accompanied by web-based demos. For more information, see our author guidelines at www.computer.org/itpro/author.htm.

WWW.COMPUTER.ORG/ITPRO

Digital Object Identifier 10.1109/MC.2019.2915435