Toward Design and Implementation of a Quantum-Classical Hybrid Computing System

Shota Arakaki[1], Masao Hirokawa[2][a] and Hiroki Watanabe[2]
[1]*Joint Graduate School of Mathematics for Innovation, Kyushu University, 744 Motooka, Nishi-ku, Fukuoka, Japan*
[2]*Faculty of Information Science and Electrical Engineering, Kyushu University, 744 Motooka, Nishi-ku, Fukuoka, Japan*

Keywords: Quantum-Classical Hybrid Computation, High-Level Quantum Programming Language, Transpiler.

Abstract: We propose the paradigm of our quantum-classical hybrid (QCH) computing system. The combination of the quantum and classical parts is realized by a kind of distributed computation with the help of the classical channel such as used in the trusted node of quantum network. In the programming at the front-end of the QCH computing system, the description of quantum circuits is hidden as possible, and then, the intrinsic functions corresponding to the individual quantum circuits is used instead. We show the example of a QCH computation and the results of some of the experiments. We generalize the notion of the QCH computation based on those results, and aim to establish the concept of our paradigm.

1 INTRODUCTION

Since the quantum supremacy was set as a quantum computing milestone, quantum computer has been the focus of attention in the last decade (Harrow and Montanaro, 2017; Preskill, 2018; Arute et al., 2019). Its social implementation is based on nothing but the establishment of a computing system with quantum computation. According to the current state of quantum technology, it is the arithmetic unit of the processor that reaps benefits from quantum mechanics. Today's quantum computer is among computing systems with the quantum arithmetic unit; the system needs the help of classical technologies for other units in the computer architecture as well as the language processor. Thus, researches on the problem of the construction of the quantum-classical hybrid (QCH) computing systems have increased significantly (Kim et al., 2023; Tran et al., 2023). It, however, is hard to reconcile classical and quantum worlds and make both coexist since the physical theory combining classical mechanics and quantum mechanics has not been completed yet, and the classical logic and the quantum logic are different from each other. The barriers between the two worlds compound the problem of exploitation.

Another problem arises. A programmer prepares a computer program with a high-level programming language and feeds it into a computing system. This programmer's job is the very beginning process at the front-end of the computing system. In this sense, we refer to the programmer as a front-end programmer throughout this paper. For conventional computing systems, the isolation between the language system and the circuit system is almost established: the language processor and the computer architecture. Usually, therefore, the front-end programmers do not have to write any logical circuit in their programs for the conventional (hereinafter "classical") computation. On the other hand, for current quantum computing systems, such the isolation has not been established yet; therefore, the front-end programmers have to write some quantum circuits in their programs. This means that they have chances to operate some quantum circuits. Regardless of whether it is intentional or not, they can write a circuit producing troublesome noises and break a quantum computation.

We propose a paradigm of the QCH computing system so that the troublesome and irksome tasks for the front-end programmers can be pushed away to the low layers consisting of the language processor and computer architecture. In our QCH computing system, the front-end programmers have two options: writing quantum circuits or not. In the latter case, the front-end programmers use intrinsic functions for quantum computation, just like as they use built-in functions of the four arithmetic operations. Thus, the quantum computation is invoked as a subroutine in

[a] https://orcid.org/0000-0001-9020-3992

Arakaki, S., Hirokawa, M. and Watanabe, H.
Toward Design and Implementation of a Quantum-Classical Hybrid Computing System.
DOI: 10.5220/0013542300004525
In *Proceedings of the 1st International Conference on Quantum Software (IQSOFT 2025)*, pages 112-119
ISBN: 978-989-758-761-0
Copyright © 2025 by Paper published under CC license (CC BY-NC-ND 4.0)

the low-layer processes. To the best of authors knowledge, such a paradigm for the QCH computing system has not been proposed.

In our paradigm of the QCH computing system, we consider the limit of the ability against noises for individual quantum processors. As the depth of a quantum computation makes the processor go beyond the limit of its ability, the QCH transpiler warns the front-end programmers, and enables them to use a kind of distributed computation for the original quantum computation. For this computation, we employ the classical channel such as used in the trusted node of quantum network (Salvail et al., 2010; Huttner et al., 2022; Lemons et al., 2023). In order to obtain the ability limit, we use a noise generation quantum circuit (NGQC) (Hirokawa, 2021) which is among quantum circuits generating noises in the zero-noise extrapolation (He et al., 2020; Majumdar et al., 2023). Our goal is that the QCH computing system makes the front-end programmers avoid writing quantum circuits in their programming if at all possible. In this paper, we report some results of the example of a QCH computation. Based on these results, we establish the notion of the QCH computing system including the high-level programming language.

Throughout this paper, the gate-level netlist means the description of the design of a quantum circuit as well as of a classical circuit after the logical synthesis in classical computer architecture. Thus, the gate-level netlist is physically executable by a quantum processor. The term, transpiler, is used for the transformation from a source program written by a front-end programmer to its gate-level netlist, while the term, compiler, is used for the transformation from the source program to its object program in the language processing.

2 PROPOSAL OF NEW CONCEPT OF QCH COMPUTATION SYSTEM

In the first part of this section, we propose the concept of an architecture design for the QCH computation, and formalize some of them in brief. A class of problems which the QCH computation can apply is restricted so that the specification of the problem in the class can be written in the classical logic, and moreover, a proper quantum computation can be used for the problem.

Our idea of the QCH computation is motivated from the solution method for the following problem. Give any sequence, $a_1, a_2, \cdots, a_N$, of N bits. Assume that all the elements a_i are 1 other than $a_{i_*} = 0$ for an index i_*. Then, consider the binary search for the sequence to seek the solution a_{i_*}. For simplicity, we assume that there is just one solution in the sequence.

One of advantages of the classical computation is that the call by value method can be repeated in an iteration statement over and over again. This cannot be performed by quantum computation in general due to the reduction of wave packet and no-cloning theorem. The binary search is based on this advantage, and its time complexity is $\mathrm{O}(\ln_2 N)$. It basically consists of the two operations.

O1. Determine a division point $a_\star$ (i.e. $i = \star$) and split the sequence at $a_\star$ into the two exploration intervals so that one interval includes a_i for $1 \leq i \leq \star$, while another includes a_i for $\star < i \leq N$.

O2. Choose the interval with the solution, and screen out the other interval.

If each element a_i were an integer and the sequence were sorted, the standard method of the classical computation would work for O2 since it is easy to make a judgment function for O2 by comparing the values of the division point $a_\star$ and the solution a_{i_*}. Unfortunately, however, our sequence is of N bits, and therefore, it is not sorted. A quantum computation can play the role of the judgment for O2; the judgment function is defined by using the $(m+1)$-qubit Toffoli gate, $(x_1 \,\mathbf{AND}\, \cdots \,\mathbf{AND}\, x_m)\,\mathbf{XOR}\, x_{m+1}$, where $x_1, \cdots, x_m$ are for the control qubits and x_{m+1} is for a target qubit. The judgment function $\mathbf{judgment}(x_1, \cdots, x_m)$ is concretely defined by $(x_1 \,\mathbf{AND}\, \cdots \,\mathbf{AND}\, x_m)\,\mathbf{XOR}\, 1$. Therefore, if the sequence a_1, $\cdots$, a_m includes the solution, then $\mathbf{judgment}(a_1, \cdots, a_m) = 1$. On the other hand, if it does not include the solution, then $\mathbf{judgment}(a_1, \cdots, a_m) = 0$. The front-end programmers can use this judgment function as if it were an intrinsic function. Once the front-end programmers use the judgment function in their source programs, the QCH transpiler makes the subroutine call for **judgment** and executes the quantum computation.

If necessary, the QCH transpiler or the front-end programmer divide the sequence of N bits into some subsequences so that the number of bits of each subsequence can be less than $2M$ for a proper integer M given below. For each quantum computer processor (QP), a proper value $N_* = N_*^{\mathrm{QP}}$ (hereinafter "critical value") of **judgment** is determined by the total number of CNOT gates used in the gate-level netlist of **judgment**. We here assume that we use one QP for one judgment function. The above integer M is defined by $M := \min_{\mathrm{QP}} N_*^{\mathrm{QP}}$, where $\min_{\mathrm{QP}}$ means the minimum running over all the QPs used for **judgment**.

For an arbitrary subsequence, we denote by m_* the number of bits making the subsequence. The division points $a_\star$ are set as $\star = \lfloor m_*/2 \rfloor + 1 = \lceil m_*/2 \rceil$, where $\lfloor\ \rfloor$ and $\lceil\ \rceil$ are respectively the floor function and ceiling function. We make the $(m+1)$-qubits Toffoli gate by using some standard Toffoli gates for $m \leq m_*$. The QCH transpiler reads out the result of **judgment**$(a_1, \cdots, a_\star)$. According to the result, it executes the judgment function for the survivor, that is, surviving subsequence. These procedures need a help of a classical channel between the QP for **judgment**$(a_1, \cdots, a_\star)$ and the QP for **judgment**(the first half sequence of survivor). Here, we have two remarks. One is that we need an I/O transformation between bits and qubits in the channel. The other is that the result of judgment function is obtained through quantum-state estimation.

We have implemented the judgment function, and determined the critical value N_* for each processor. Then, some results of the benchmark for N_* are given (Watanabe, 2022). For one judgment function, we use just one IBM Q processor: ibmqx2, ibmq_quito, ibmq_belem, ibmq_lima, ibmq_manila, ibmq_santiago, ibmq_bogota, ibmq_athens (which are 5-qubits processors), and ibmq_16_melbourne (which is 15-qubits processors). Our experimental results say the following. In their individual netlists of the judgment function for the subsequence with $m_* = 9$, ibmqx2 with the cruciform topology uses about 20 CNOT gates. Meanwhile, ibmq_quito, ibmq_belem, and ibmq_lima with the T-shaped topology use about 40 CNOT gates, and ibmq_manila, ibmq_santiago, ibmq_bogota, ibmq_athens with the linear-type topology use about 50 CNOT gates. In the netlists of the judgment function for the subsequence with $m_* = 29$, ibmq_16_melbourne uses about 230 CNOT gates. Here, we note that the IBM Q processor requires the so-called nearest-neighbor interaction, and therefore, extra SWAP gates are used to achieve this interaction. We recall that one SWAP gate consists of three CNOT gates. The accuracy rate here is given by a statistical quantum-state estimation, that is, the number of success divided by the total number of trials, in the case for $m_* = 9$. The accuracy rates of ibmqx2 and ibmq_bogota are almost 100%. Meanwhile, it is between 55% and 100% for ibmq_belem, and between 77% and 100% for ibmq_lima. Therefore, defining the critical value of **judgment** by the maximum of the number of variables so that **judgment** can work well, we can set it as 9 for these QP. However, our experiments reveal that ibmq_16_melbourne works for the subsequences only with $m_* < 6$.

We refer to the binary search described above as "QCH binary search" from now on. Generalizing this, we consider the QCH computation whose computation procedures are in P1-P3 below. In order to handle the QCH computation, we make the concept of our QCH computing system. The concept comprise C1-C3.

C1. The QCH transpiler hides the description of some quantum circuits in the front-end programming as possible. The hidden description should be processed as a subroutine call in the QCH transpiling process.

C2. The QCH transpiler can handle a multi-thread processing. Concretely, it divides a quantum computation written in a source program into several parts so that the divided quantum computation of each part is not broken. Therefore, the quantum transpiler uses QP like a multi-core processor, and assigns the task of each part to a proper core.

C3. The QCH transpiler works as a multi-pass compiler. Thus, at least 2 languages for the intermediate codes should be prepared. These codes are respectively for the language and the quantum circuit, and then, the optimizations of language and quantum circuits are separated.

C4. In the error-recovery process (Ullman, 1976) of the QCH transpiler, the error warnings are alerted if it meets some problems concerning quantum circuits as well as problems concerning the lexical, syntactic, and semantic analysis in the language processing.

The flow of the QCH computing system in Fig.1 is in the following. The QCH computing system receives a source program input from the front-end. The QCH transpiler classifies the source program into the classical and quantum parts. The QCH transpiler makes the object program of the classical part, and transfers it to the classical computer processor through the optimization phase of the classical language processor. In the source program, the quantum part is written with some intrinsic functions or quantum circuits by a front-end programmer. If QCH transpiler gets the intrinsic function, it makes a subroutine

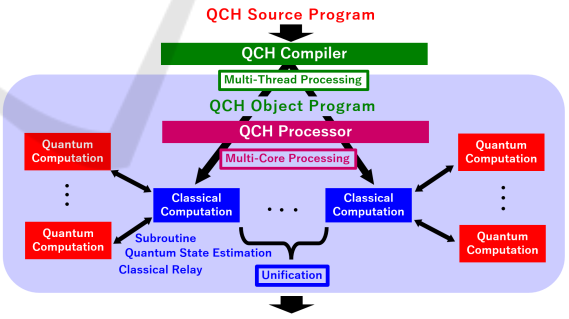

Figure 1: The schematic picture of the QCH computing system.

call for the corresponding quantum computation, and asks a QP to give its answer. If the quantum computation is not so deep, the QCH transpiler optimizes the quantum circuit and executes it with a proper QP. Otherwise, the QCH transpiler executes the multi-thread processing with good use of a multi-core processor. In this multi-thread processing, since the quantum memory for movement of the halfway computation from an exhausted core to another fresh core has not been invented yet, we adopt a notion of classical channel, as is often used in the trusted node of quantum network (Salvail et al., 2010; Huttner et al., 2022; Lemons et al., 2023). The result of the halfway quantum computation by the exhausted core is measured and converted to the classical data. The converted data are converted to quantum data, and handled in the next fresh core. Since measurements destroy the quantum states such as superposition, the QCH transpiler uses the quantum-state estimation and the help of classical relays. The QCH computing system unifies the results of the divided quantum computation if necessary, and outputs the answer to the front-end.

One of the examples for C1 is the QCH binary search. As explained in Sec.3, the front-end programmers can write the circuit destroying a quantum computation in their programs whether they willingly or unwillingly do it. Thus, we think it is important to hide some vital operations in the low-layer from the front-end. We can learn such importance from the history of the exploitation of the classical computing system, for instance, the history of the disappearance of the go-to statement in high-level programming languages. The recursiveness is vital for the classical computability in accordance with the recursive function theory (Cutland, 1980), and therefore, its notion is indispensable for classical computation (Roberts, 1986). Nowdays, we use the iteration statement (i.e., loop statement) to describe the recursion without using the go-to statement. The recursion is handled by the GOTO function in the syntactic analysis (i.e., parsing) of compiler (Aho and Ullman, 1977; Aho et al., 2006). The recursiveness is realized on the circuits with the help of the jump instruction in the instruction set of the computer architecture (Dandamudi, 1998). Thus, the go-to operation is important for the compiler and instruction-set architecture. Nevertheless, the go-to statement is hidden in the structured programming (Dijkstra, 1970; Dahl et al., 1972; Ullman, 1976) based on the structured program theorem (Böhm and Jacopini, 1966). It is because the go-to statement causes the so-called spaghetti code (Dijkstra, 1968; Knuth, 1974) and the software crisis (Naur and Randell, 1969; Dijkstra, 1978).

We prepare an alternative for the front-end programmers so that the QCH computing system can hide some quantum circuits from them. In other words, in the case where the front-end programmers need only the specification of a quantm computation, the only thing they have to do is to write the intrinsic function (e.g., **judgment**) satisfying the specification, not to write the quantum circuit. Then, the quantum computation is handled as a subroutine inside the QCH transpiler.

We can come up with an example of the scheme for C2 in the QCH binary search. A quantum circuit destroying quantum computation with noises is given by, for instance, NGQC (Hirokawa, 2021) as in Sec.3. The quantum computation by QP is broken if its depth exceeds the critical value. Referring to the critical value, the QCH transpiler divides the original quantum computation into some small quantum computations so that the depth of each divided quantum computation can be less than critical value. The QCH transpiler has to refer to all the results of the divided quantum computations, and then, it can obtain a solution of the original quantum computation by gathering and processing them with the help of the classical computation. For the result of each divided quantum computation, it estimates the individual correct answer. Thus, the notion of classical channel explained above is adopted in the QCH computing system. Describing repeatedly, the QCH transpiler employs the quantum-state estimation and the classical relays to connect the small quantum computations. Therefore, the procedures of our QCH computation are as follows:

P1. Divide the target quantum computation into some quantum computation so that the depths of all the divided quantum computations can be less than the critical value.

P2. Label divided quantum computations with numbers. The labeled quantum computations are handled in the order of labels. Read out each result and store it in the individual classical register.

P3. Output the solution if all the computations are completed and the data make the solution of the original quantum computation. If another quantum computation is needed using the data in the classical register, input them and execute a new QCH computation.

P4. Use the iteration process as a classical computation in the case where some repetitions are required for P1–P3.

The high-level programming language to write programs for the QCH computation consists of a standard programming language and a quantum-circuit description language. The two types of languages are mathematically different. We need to make the both coexist in a high-level programming language. We

design the high-level programming language for the QCH computation so that it can be processed by the QCH transpiler.

We have been studying its design and implementation of the prototype of the high-level QCH programming language according to C3 and C4 (Arakaki, 2023). We have tried to write a program for the QCH binary search with the language, and to execute it on a quantum-computation simulator. The design directions for our high-level QCH programming language is as follows.

D1. We make the high-level QCH programming language free from the properties of quantum hardware.

D2. We consider and add some special rewriting rules (i.e., production rules) for the error-recovery process (Ullman, 1976) in the language processing so that the front-end programmers can be alarmed to the possibility that the quantum computation has some noise-induced errors.

D3. We adopt the lambda-calculus for quantum computation (van Tonder, 2004; Selinger and Varilon, 2005).

D4. We introduce the type theory (Fu et al., 2020) and multi-stage calculus (or multi-stage programming) (Taha and Sheard, 2000) into our QCH computing system.

D1 means the ideal separation of software and hardware. More precisely, we consider the completely executable description of quantum circuit so that the QCH program can be free from the characteristic of physics of the device for QP. In particular, the QCH program should be freed from how to use the classical and quantum registers. Recall the recursive program in the C programming language. In the case the computation is so deep, we often meet 'Segmentation Fault' due to the stack overflow. Actually, NGQC in Sec.3 is made by taking advantage of how to use of quantum registers. In order to gain its freedom, the roles of the compiler part and the instruction-set architecture part of the QCH transpiler are important.

For D2, the QCH transpiler must learn the error rates of the quantum gates and the critical values of intrinsic functions for quantum computation. The QCH transpiler should activate the special rewriting rules if it detects the possibility of errors as well as other linguistic errors in the language processing.

We set D3 and D4 because several preceding studies show that quantum computation can be represented with the typed lambda calculus (Selinger and Varilon, 2005; Varilon, 2011; Kawata and Igarashi, 2019). In particular, it enables us to manipulate classical and quantum both data. In the multi-stage calculus, the process of the target program is divided into several stages, and it is evaluated at the individual stages. Therefore, we believe that the method of the typed lambda calculus meets our QCH computation, and moreover, that it is useful for the multi-pass compiler. We think we can prepare the individual codes for language and quantum circuit following the types. For our QCH programming language, therefore, we adopt the ML-like syntax with the extended let-binding (such as let val bindings of ML, or let/let* bindings of Common Lisp) for stage progression in order to facilitate the handling of both classical and quantum data.

3 NOISE CREEPING IN QUANTUM CIRCUIT

The pass manager carries the quantum-circuit optimization in IBM Q transpiler. We, in fact, can write several quantum circuits producing noises and cheating the pass manager's ability. We are interested in the noises caused chiefly by CNOT gates (He et al., 2020; Hirokawa, 2021; Majumdar et al., 2023). How to cheat is to use the fact that the technology of optimization in the language processing of the compiler and that in the circuit processing of the logical synthesis are different. The individual optimizations meet so tough problems. The combination of the language processing and the microarchitecture compounds the difficulties of the problem. For instance, the optimization problem of the instruction scheduling combined with the register allocation is NP-hard even for classical computer processors. In particular, the optimization of the quantum circuits means that of the individually corresponding quantum Hamiltonian (Kandala et al., 2019).

We consider NGQC which generates some noises in a QP during the computation. The simplest NGQC is the quantum circuit for the **skip** statement. We denote this program specification merely by **skip**. In mathematical terms, **skip** plays the role of the identity in quantum computation. Therefore, the best optimization of **skip** is to do nothing. NGQC can deceive the IBM Q's transpiler into believing NGQC is for a meaningful operation, not just an identity operation, and it can slip through the individual optimization processes in the transpiler.

Let A and B be unitary operators in a Hilbert space. We now assume $AB = I$, where I is the identity operator. Then, we realize that B is uniquely determined; we obtain $B = A^*$, the adjoint operator of A, in mathematics. In the individual optimization processes, therefore, the compiler should employ the NOP instruction, and then, the logical synthesis only has to avoid the executions of A and A^*. Thus, as the

transpiler finds AA^*, it should not output any quantum circuit in the gate-level netlist. The computation in the low layers are sometimes beyond a mathematical grasp. By thinking out how to use some quantum registers, we can make a circuit of the quantum operation for B so that it is different from that of the quantum operation for A^* though $AB = I$ holds. The most fundamental operation with a use of register in classical computation is for the assignment statement, $x := a$, restoring the value of a in the register of the variable x. The classical SWAP statement is written by the three assignment statements, and then, **skip** is obtain by executing the SWAP statements twice in succession. In the individual optimization processes even of classical computers, it is difficult remove **skip**$\cdots$**skip**, the serial executions of **skip**. We can realize this difficulty, for instance, by using the optimization options of GCC, the GNU Compiler Collection. We can set up a similar trick using the quantum register of the CNOT gate, and make a SWAP gate with the three CNOT gates. The **skip** gate is obtained by repeating the two SWAP gates continuously. Then, the trick enables us to make various NGQCs in Qiskit of IBM Q so that the transpiler seldom grasps all of them and cannot reduce them to the best optimization. Because our NGQC is made by using the CNOT gates, the noise of NGQC originates from that of the CNOT gate. The total number of the CNOT gates in the gate-level netlist is at least 6 per one fundamental module of NGQC. We denote by U_{NG} the unitary operator for our NGQC. Thus, U_{NG}^n is $U_{\mathrm{NG}}\cdots U_{\mathrm{NG}}$, the n times products of U_{NG}.

We consider quantum circuits for the two-qubits **skip** operation given by

$$U_{\mathrm{NG}}^n|11\rangle = |11\rangle. \tag{1}$$

This unitary operator is implemented by using SWAP gates as described above. We use ibmq_santiago for the examinations in Fig.2. The left graph of Fig.2 is one of the results for Eq.(1) with $n = 0$. Following the majority decision, we can realize that $|11\rangle$ is the solution of the quantum computation, $U_{\mathrm{NG}}^n|11\rangle$, for $n = 0$. Once turning NGQC on, the aspects of the results change completely. The right graph of Fig.2 is one of the results for $n = 30$. In the case $n = 30$, the majority decision cannot tell us that $|11\rangle$ is the solution of the quantum computation, $U_{\mathrm{NG}}^n|11\rangle$. The total number N_{CN} of the CNOT gates in the netlist is equal to or more than 180 for $n = 30$. In this way, we can statistically obtain an approximate upper limit of N_{CN}. We note that the value of N_{CN} is dependent on the topology of QP and the ability of the IBM Q transpiler.

Actually, the value of N_{CN} also depends on a kind of quantum state. For the unitary operator U of any quantum computation, we can leave U_{NG}^n in the quantum computation by, for instance, $U_{\mathrm{NG}}^n U$ and UU_{NG}^n. The results of U, $U_{\mathrm{NG}}^n U$, and UU_{NG}^n are same. However, the noise induced by NGQC creeps in the latter two unitary operations. The larger n grows, the more the noise effect increases. Let us take U_{QE}, the unitary operator of making an entangled state, for the unitary operation U. For example, it is given by

$$U_{\mathrm{QE}}|00\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle). \tag{2}$$

to produce a Bell state now. Then, we realize that the noise produced by NGQC easily destroys the quantum entanglement. We use ibmq_athens for the tests in Fig.3. The left graph of Fig.3 is for $n = 0$, and the right graph of Fig.3 is for $n = 10$.

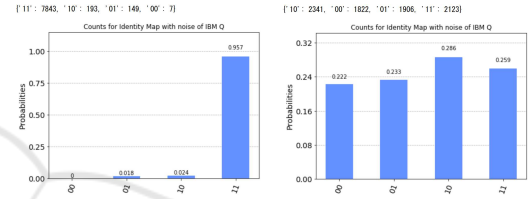

Figure 2: $U_{\mathrm{NG}}^n|11\rangle = |11\rangle$. Test results for $n = 0$ (left) and $n = 30$ (right).

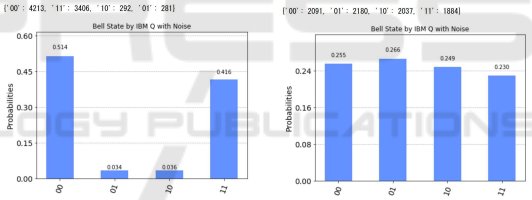

Figure 3: $U_{\mathrm{QE}}U_{\mathrm{NG}}^n|00\rangle = (|00\rangle + |11\rangle)/\sqrt{2}$. Test results for $n = 0$ (left) and $n = 10$ (right).

4 CONCLUSION

We have described the conception of our quantum-classical hybrid (QCH) computing system as well as we have explained the background that led up to it. We have introduced the broad outline of our concept of the QCH transpiler for the system and that of our scheme for the design of the high-level QCH programming language. We have reported implementation of QCH computing system in progress.

In the light of the security for the computing system, it should be restricted that the front-end programmers write quantum circuits in their program because there is a possibility that NGQC makes a quantum computer virus.

In order to achieve our paradigm, we must clarify what kinds of quantum computations can be hidden

from the front-end programmers, and establish how we can divide quantum computation into some parts which are computable in the QCH computing system. We have been studying these problems.

ACKNOWLEDGMENTS

The authors wish to thank the referees for their useful comments. They acknowledge the support from MEXT Quantum Leap Flagship Program (MEXT Q-LEAP) Grant Number JPMXS0120351339.

REFERENCES

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, &Tools*. Addison-Wesley, Boston, 2nd edition.

Aho, A. V. and Ullman, J. D. (1977). *Principles of Compiler Design*. Addison-Wesley, Boston, 1st edition.

Arakaki, S. (2023). *A Research on Classical-Quantum Hybrid Programming Language And Its Implementation (in Japanese)*. Master thesis, Kyushu University, Fukuoka.

Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J. C., Barends, R., Biswas, R., Boixo, S., Brandao, F. G. S. L., Buell, D. A., Burkett, B., Chen, Y., Chen, Z., Chiaro, B., Collins, R., Courtney, W., Dunsworth, A., Farhi, E., Foxen, B., Fowler, A., Gidney, C., Giustina, M., Graff, R., Guerin, K., Habegger, S., Harrigan, M. P., Hartmann, M. J., Ho, A., Hoffmann, M., Huang, T., Humble, T. S., Isakov, S. V., Jeffrey, E., Jiang, Z., Kafri, D., Kechedzhi, K., Kelly, J., Klimov, P. V., Knysh, S., Korotkov, A., Kostritsa, F., Landhuis, D., Lindmark, M., Lucero, E., Lyakh, D., Mandrà, S., McClean, J. R., McEwen, M., Megrant, A., Mi, X., Michielsen, K., Mohseni, M., Mutus, J., Naaman, O., Neeley, M., Neill, C., Niu, M. Y., Ostby, E., Petukhov, A., Platt, J. C., Quintana, C., Rieffel, E. G., Roushan, P., Rubin, N. C., Sank, D., Satzinger, K. J., Smelyanskiy, V., Sung, K. J., Trevithick, M. D., Vainsencher, A., Villalonga, B., White, T., Yao, Z. J., Yeh, P., Zalcman, A., Neven1, H., and Martinis, J. M. (2019). Quantum supremacy using a programmable superconducting processor. *Nature*, 574:505–510.

Böhm, C. and Jacopini, G. (1966). Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9:366–371.

Cutland, N. (1980). *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, Cambridge, 1st edition.

Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R. (1972). *Structured Programming*. Academic Press, London.

Dandamudi, S. P. (1998). *Introduction to Assembly Language Programming*. Springer, Berlin, 1st edition.

Dijkstra, E. W. (1968). Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148.

Dijkstra, E. W. (1970). Structured programming. In Randell, B. and Buxton, J., editors, *Software Engineering Techniques*, pages 84–88. NATO Scientific Affairs Division.

Dijkstra, E. W. (1978). The humble programmer. In Gries, D., editor, *Programming Methodology. A Collection of Articles by Members of IFIP WG2.3*, pages 84–88. Springer.

Fu, P., Kishida, K., and Selinger, P. (2020). Linear dependent type theory for quantum programming languages. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 440–453. ACM.

Harrow, A. W. and Montanaro, A. (2017). Quantum computational supremacy. *Nature*, 135:203–209.

He, A., Nachman, B., de Jong, W. A., and Bauer, C. W. (2020). Zero-noise extrapolation for quantum-gate error mitigation with identity insertions. *Phys. Rev. A*, 102:012426.

Hirokawa, M. (2021). Can we make the noise filtering theory for nisq computer? In *Proceedings of the Joint Workshop of 14th Superconducting SFQ VLSI Workshop and 3rd Workshop on Quantum and Classical Cryogenic Devices, Circuits, and Systems*. Nagoya University.

Huttner, B., Alléaume, R., Diamanti, E., Fröwis, F., Grangier, P., Hübel, H., Martin, V., Poppe, A., Slater, J. A., Spiller, T., Tittel, W., Tranier, B., Wonfor, A., and Zbinden, H. (2022). Long-range qkd without trusted nodes is not possible with current technology. *npj Quantum Information*, 8:108.

Kandala, A., Temme, K., Córcoles, A. D., Mezzacapo, A., Chow, J. M., and Gambetta, J. M. (2019). Error mitigation extends the computational reaqch of a nboisy quantum processor. *Nature*, 108:491–495.

Kawata, A. and Igarashi, A. (2019). A dependently typed multi-stage calculus. In *Programming Languages and Systems*. Springer.

Kim, D., Noh, P., Lee, H.-Y., and Moon, E.-G. (2023). Advancing hybrid quantum-classical algorithms via mean operators. *Phys. Rev. A*, 108:L010401.

Knuth, D. E. (1974). Structured programming with go to statements. *ACM Computing Surveys*, 6(4):261–301.

Lemons, N., Gelfand, B., Lawrence, N., Thresher, A., Tripp, J. L., Gammel, W. P., Nadiga, A., Meier, K., and Newell, R. (2023). Extending quantum key distribution through proxy re-encryption. *J. Opt. Commun. Netw.*, 15:457–465.

Majumdar, R., Rivero, P., Metz, F., Hasan, A., and Wang, D. S. (2023). Mitigation with digital zero-noise extrapolation. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 1, pages 881–887. IEEE.

Naur, P. and Randell, B. (1969). *Software Engineering*. NATO Scientific Affairs Division, Brussels,.

Preskill, J. (2018). Quantum computing in the NISQ era and beyond. *Quantum*, 2:79.

Roberts, E. (1986). *Thinking Recursively*. John Wiley & Sons, New York, 1st edition.

Salvail, L., Peev, M., Diamanti, E., Alléaume, R., Lütkenhaus, N., and Länger, T. (2010). Security of trusted repeater quantum key distribution networks. *J. Comput. Secur.*, 18:61–87.

Selinger, P. and Varilon, B. (2005). A lambda calculus for quantum computation with classical control. In Urzyczyn, P., editor, *Typed Lambda Calculi and Applications*, pages 354–368. Springer Berlin Heidelberg.

Taha, W. and Sheard, T. (2000). Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248:211–242.

Tran, Q. H., Ghosh, S., and Nakajima, K. (2023). Quantum-classical hybrid information processing via a single quantum system. *Phys. Rev. Research*, 5:043127.

Ullman, J. D. (1976). *Fundamental Concepts of Programming Systems*. Addison-Wesley, Boston, 1st edition.

van Tonder, A. (2004). A lambda calculus for quantum computation. *SIAM Journal on Computing*, 33:1109–1135.

Varilon, B. (2011). On quantum and probabilistic linear lambda-calculi (extended abstract). *Electronic Notes in Theor. Comp. Sci.*, 270(1):121–128.

Watanabe, H. (2022). *A Research Toward Classical-Quantum Hybrid Distributed Computation (in Japanese)*. Master thesis, Hiroshima University, Higashi-Hiroshima.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS