Program Equivalence Checking for the Facilitation of Quantum Offloading

Jon Speer
Faculty of Science
University of Helsinki
Helsinki, Finland
jongspeer@gmail.com

Jukka K. Nurminen
Faculty of Science
University of Helsinki
Helsinki, Finland
jukka.k.nurminen@helsinki.fi

***Abstract*—Computational offloading involves the transfer of computational tasks to a separate device. We apply this concept to quantum computing, whereby particular algorithms (i.e. "quantum algorithms") are automatically recognized and executed on a quantum computer. We propose a method that utilizes program equivalence checking to discern between code suited for execution on a conventional computer and a quantum computer. This process involves comparing a quantum algorithm's implementation with code written by a programmer, with semantic equivalence between the two implying that the programmer's code should be executed on a quantum computer instead of a conventional computer. Using a novel compiler optimization verification tool named CORK, we test for semantic equivalence between a portion of Shor's algorithm (the "prototype") and various modified versions of this code (representing the arbitrary code written by a programmer). Some of the modified versions are intended to be semantically equivalent to the prototype while others semantically inequivalent. Our approach is able to correctly determine semantic equivalence or semantic inequivalence in a majority of cases.**

***Index Terms*—Program Equivalence Checking, Quantum Computing, Quantum Offloading, Shor's Algorithm**

I. Introduction

In this paper, we analyze the use of program equivalence checking for the facilitation of quantum offloading. The term "quantum offloading" is derived from "computational offloading", which involves the transfer of computational tasks to a separate device, typically because this separate device is more suited to performing a particular computation. Quantum offloading involves automatically recognizing particular algorithms (i.e. "quantum algorithms") that are then executed on a quantum computer. (By "quantum algorithms", we mean algorithms that are ideal for execution on a quantum computer, though these algorithms may also be executed on a conventional computer as well.) The automatic recognition of quantum algorithms is the focus of this paper. Our goal is to propose a method that can be applied at some point in the future where general-purpose quantum computers are no longer a theoretical construct, but rather an integral part of the computing landscape.

This paper envisions a future in which quantum computers have achieved non-trivial capabilities that greatly surpass the limitations faced by conventional computers. Realization of this capability means that *certain* types of tasks will be completed much more quickly on a quantum computer than on a conventional computer, while other types of tasks will see *no* speedup on a quantum computer. Thus, the latter may be best left to conventional computers. It is reasonable to assume that both computing technologies will exist in parallel, with quantum computers "in the cloud" and conventional computers found locally, e.g. in laptops, cell phones, etc. This stems not only from the aforementioned limitations of quantum computers, but also from the cost and complexity of quantum computers. While it is likely that production costs will fall as technology progresses, conventional computers will likely remain cheaper to produce - possibly significantly cheaper - meaning that keeping the more expensive quantum computers in the cloud will be an economically preferable approach. The much larger physical size of quantum computers (as of the writing of this paper) also makes the cloud-based approach preferable.

Assuming this cloud-based model, and a computing landscape where programmers have a need for quantum hardware on a somewhat regular basis (we will expand on this point later), code will be separated into conventional and quantum categories, meaning that programmers will need to write code for both types of architectures. While dedicated languages already exist for the programming of quantum computers, the gradual incorporation of the quantum realm into mainstream programming languages would make sense from a practicality standpoint. One approach is the use of APIs: programmers would have libraries at their disposal to perform tasks on quantum hardware. However, this would require a programmer to have a requisite understanding of these APIs; this is by no means insurmountable, considering that APIs are commonplace today. However, it's worth considering if another approach is feasible, one where the programmer need not even think about the underlying hardware.

This alternative approach involves abstracting away the conventional and quantum realms altogether from the programmer's perspective. Instead of the programmer knowing, or even caring, about what should be executed on a conventional computer and what should be executed on a quantum computer, a compiler or interpreter (i.e. translator) would make that decision. This concept is the focus of this paper.

The key points of this paper:

978-0-7381-4394-1/21/$31.00

1) We propose the concept of quantum offloading to automatically execute suitable code on quantum hardware (Section I).
2) We discuss the use of program equivalence checking for the automatic detection of code suitable for quantum offloading (Section III).
3) We implement a trivial example of program equivalence checking that demonstrates how the code detection phase of quantum offloading would work in the case of Shor's algorithm (Section IV and Section V).
4) We discuss our test results and the future potential of quantum offloading (Section VI).

II. Related Work

The idea of utilizing program equivalence checking in order to identify quantum algorithms is an unexplored topic. However, program equivalence itself is a well-known and researched topic with numerous applications, including algorithm recognition, program verification, program optimization, and compiler optimization [1] [2] [3] [4] [5]. The term "semantic equivalence" may be used to describe programs that are equivalent: even if written differently, they have the same meaning. For example, a particular sorting algorithm may be implemented in various ways, but (assuming they are implemented correctly) still implement the same algorithm and thus would be considered semantically equivalent. Program equivalence concerns semantic equivalence as it relates to programs.

The "equivalence problem" asks whether it is possible to determine if two programs are equivalent. The answer, as it relates to "decidability", is that it varies, based on the representation of the programs being compared (e.g. context-free grammar, finite-state automata, etc.). A problem is called *decidable* if there exists a "yes" or "no" answer to a given problem [6].

The equivalence problem is said to be undecidable "as soon as the considered program class is rich enough to be interesting" [5]. However, it has been formally shown that the equivalence problem is decidable in simple cases [7]. Other research into program equivalence has shown promising results [1] [2] [3] [4]. Simple cases of algorithm recognition can be solved using pattern matching [8].

III. Quantum Algorithm Recognition

Our algorithm recognition process is as follows: We start with an algorithm that we want executed on quantum hardware and its corresponding quantum implementation via a combination of quantum logic gates. With this in hand, we need to determine the algorithm's implementation using a programming language. This implementation serves as the prototype we can use to compare against an arbitrary block of code and determine if they are semantically equivalent. Thus, we have a method that links a quantum algorithm and its implementation on a quantum computer to code written by a programmer. Our translator would be tasked with identifying this algorithm during the process of translation, resulting in this particular block of code being executed on a quantum computer, while "non-quantum" code would be executed on a conventional computer.

Now we will elaborate on this process. First, we address the correlation between the quantum algorithm and the prototype. Consider a quantum algorithm, such as Shor's algorithm for integer factorization. The implementation of Shor's algorithm on a quantum computer must be determined (in terms of quantum logic gates). The next step is to determine the implementation of Shor's algorithm via a classical programming language such as C or Java. With this in hand, we now have a link between the algorithm's quantum executable and its programming language equivalent (the prototype). The more difficult part is the process of recognizing equivalent code with respect to the prototype. It is up to our translator to perform program equivalence checking between the code it translates and the various prototype implementations of quantum algorithms. See Figure 1.

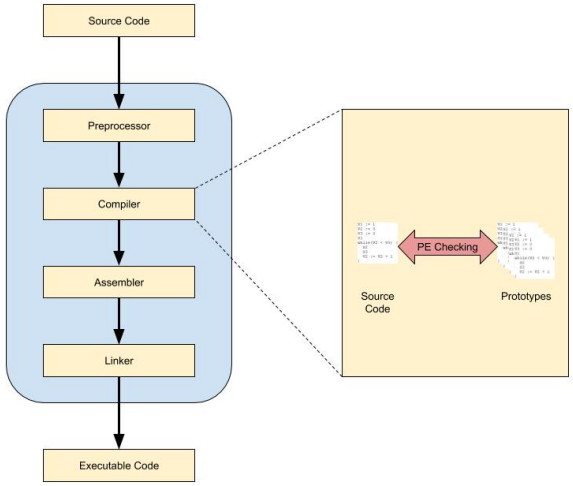

Fig. 1. Depiction of the proposed program equivalence checking process as implemented in a C compiler. The right-hand side depicts source code being checked against the collection of prototypes, corresponding to various quantum algorithms. Not shown are necessary compilation steps, such as lexical analysis, syntax analysis, semantic analysis, etc.

Figure 2 illustrates our proposed program equivalence checking method, with a quantum algorithm implemented via some programming language (the C programming language in this example). This prototype is used to compare against the code it translates. The translation process would thus entail the translator analyzing source code against the prototype. As might be expected, there would likely be more than one prototype to check against, considering that each prototype corresponds to a single quantum algorithm.

Referring back to our Shor's algorithm example, if our translator discovers the presence of the algorithm during compilation, this algorithm will be compiled for a quantum target instead of a conventional target. In this sense, what happens "under the hood" is similar to the API model discussed earlier

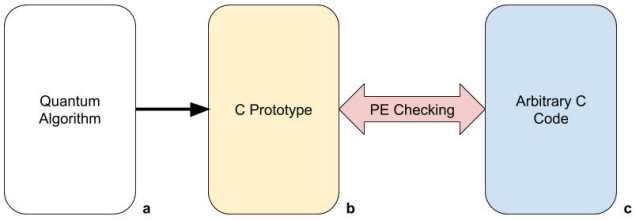

Fig. 2. The program equivalence checking procedure proposed by this paper. A quantum algorithm (**a**) is implemented via some programming language (the C programming language in this example) and serves as the prototype (**b**). Program equivalence checking is performed by our translator (Figure 1) between the prototype and the code it compiles (**c**). Semantic equivalence implies that this code is an implementation of a quantum algorithm and should thus be executed on a quantum computer.

in that execution of this particular code occurs on a quantum computer instead of a conventional computer. The difference is that the API model relies on the programmer to explicitly make this determination, whereas our proposal entails an automated approach.

IV. Program Equivalence Checking with CORK

While there exist various program verification tools, many of these focus on a single source file, whereby verification according to a certain specification is performed on this code. Our proposal centers on the ability to compare *two* pieces of code, meaning that we needed a software tool that performs program verification checking on both pieces of code and determines if there exists semantic equivalence between them. This process would thus simulate our comparison between a prototype and arbitrary code.

A paper on a novel approach to program equivalence checking [9] along with an implementation of the technique has direct applications to our own research. The authors' program equivalence checking algorithm compares a block of code with another block of code and determines if they are semantically equivalent. The ability to perform program equivalence checking directly relates to our own work: the identification of code that has been determined to be a "conventional" equivalent of a quantum algorithm.

The implementation of the compiler verification technique proposed in [9] is CORK (Compiler Optimization coRrectness checKer), created by Nuno P. Lopes and Jose Monteiro. CORK performs analysis on code written in a simplistic language created by the authors (what they call "WHILE"). Consequently, we cannot apply CORK for code written in a programming language such as C or Java. Instead, we will translate a particular block of code that we want to perform analysis on into the WHILE language. The "arbitrary" code that we want to compare to this prototype will undergo the same translation procedure. The goal will be to work with simple chunks of code in order to eliminate problems associated with our code translation.

Our translation process is illustrated in Figure 3. We start with a block of code that we designate to be the prototype. Considering the theme of this paper, this block of code should

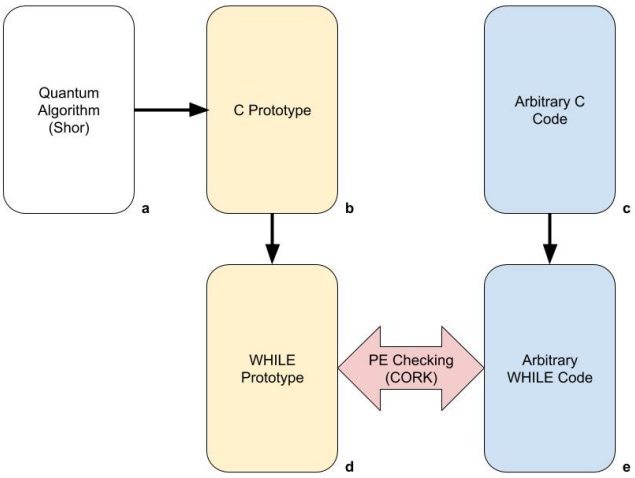

Fig. 3. The program equivalence checking procedure performed in this paper. The repeat period-finding procedure of Shor's algorithm (**a**) is implemented via the C programming language and serves as the prototype (**b**). The prototype is manually translated into the language used by CORK, known as "WHILE" (**d**). A modified version of the prototype is created (**c**), intended to be either semantically equivalent or semantically inequivalent to the prototype. This code is also manually translated into WHILE (**e**). CORK is thus used to perform program equivalence checking between the two WHILE scripts, with a determination made regarding semantic equivalence.

have high relevance to the realm of quantum computing, such as a quantum algorithm. We chose a partial implementation of Shor's algorithm as found in [10], using the C programming language. This particular block of code is the conventional implementation (i.e. code written for a modern-day computer) of an algorithm that finds a function's repeat period [10]. While the details are beyond the scope of this paper, it shall suffice to say that determining a function's repeat period is part of Shor's algorithm. The book [10] provides a useful description of Shor's algorithm, including the repeat period-finding procedure used here.

With the prototype determined, our next step is to write programs that will test the program equivalence checking technique performed by CORK. We begin with a block of code (the repeat period-finding procedure of Shor's algorithm) to serve as the original script (i.e. prototype) and write various modified versions of this script via compiler optimizations to serve as the arbitrary code. CORK is used for performing program equivalence checking between the prototype and arbitrary code.

Some versions of code we write will be intentionally semantically inequivalent to the prototype in order to test if CORK recognizes this difference. Others will be various compiler optimizations that are intended to be equivalent. We verify the functionality of the various implementations of C code by printing the repeat period as calculated by the algorithm.

The "Quantum Algorithm" of Figure 3 is the repeat period-finding procedure of Shor's algorithm in our case. This im-

plementation (using the C programming language) serves as the "C Prototype" shown as **b**. For the purposes of testing with CORK, we translate this C code into CORK's WHILE language (box **d**). We use CORK to compare this WHILE prototype against other WHILE code ("Arbitrary WHILE Code", box **e**), with a determination regarding semantic equivalence provided by CORK. Thus, for our testing purposes, program equivalence checking takes place between boxes **d** and **e** of Figure 3.

A real-world implementation of this proposal would exclude boxes **d** and **e** of Figure 3, and would instead follow the process described in Figure 2. The program equivalence checking process would be performed by our translator as depicted in Figure 1.

V. Results

Figure 4 shows our repeat period-finding prototype implemented as C code and its equivalent WHILE implementation. Again, it is worth emphasizing that the need to manually translate the C prototype to an equivalent WHILE version is due to our test environment. A real-world implementation would not include this procedure.

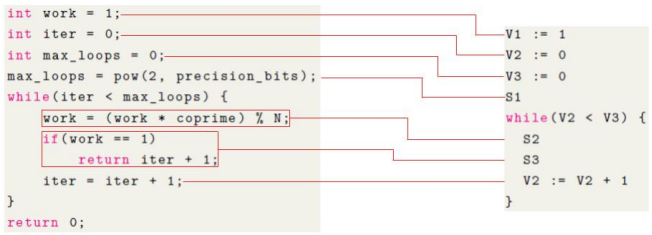

Fig. 4. The repeat period-finding prototype used in our testing. Shown here is the C code version and its equivalent WHILE implementation. The red lines link corresponding C code and WHILE code. For example, variable `work` in C code is implemented as variable `V1` in WHILE code. Due to limitations associated with WHILE, certain lines of code are substituted with template statements (`S1`, `S2`, and `S3`).

Various types of compiler optimizations were tested along with more simplistic code modifications. Figure 7 summarizes our test results. This table lists the values calculated by the repeat period-finding C code along with CORK's equivalence determination of the two WHILE scripts (the WHILE prototype and the equivalent WHILE code for each test). In total, seven tests were performed, with five producing expected results as reported by CORK.

These tests are meant to simulate the process of program equivalence checking for the repeat period-finding portion of Shor's algorithm. Thus, the same prototype is used for all seven tests.

Two tests ("Loop Modification I" and "Loop Modification II") were written such that they were intended to be semantically inequivalent to the prototype. CORK correctly identified these as such. As seen in Figure 7, these two tests produced a repeat period of 13 instead of the expected value of 12, as calculated by the prototype.

Figures 5 and 6 display the code associated with the Loop Modification I test. Figure 5 displays the C prototype and "arbitrary" C code, while Figure 6 displays the corresponding WHILE implementations (these four blocks of code correspond to boxes **b** through **e** of Figure 3). The red text in each figure indicates the line of code that has been modified in the arbitrary version with respect to the prototype. By moving this line of code from below the if-statement to above it, the calculation of `iter` results in different values. Thus, the two programs are semantically inequivalent.

```c
int ShorNoQPU(int N, int precision_bits, int coprime) {
    int work = 1;
    int iter = 0;
    int max_loops = 0;
    max_loops = pow(2, precision_bits);
    while(iter < max_loops) {
        work = (work * coprime) % N;
        if(work == 1)
            return iter + 1;
        iter = iter + 1;
    }
    return 0;
}
```

```c
int ShorNoQPU(int N, int precision_bits, int coprime) {
    int work = 1;
    int iter = 0;
    int max_loops = 0;
    max_loops = pow(2, precision_bits);
    while(iter < max_loops) {
        work = (work * coprime) % N;
        iter = iter + 1;
        if(work == 1)
            return iter + 1;
    }
    return 0;
}
```

Fig. 5. The C implementation of the Loop Modification I test. The prototype is listed at the top of the figure while the code being checked against the prototype is listed at the bottom.

```
V1 := 1
V2 := 0
V3 := 0
S1
while(V2 < V3) {
  S2
  S3
  V2 := V2 + 1
}
```

```
V1 := 1
V2 := 0
V3 := 0
S1
while(V2 < V3) {
  S2
  V2 := V2 + 1
  S3
}
```

Fig. 6. The WHILE implementation of the Loop Modification I test. The prototype is listed at the top of the figure while the code being checked against the prototype is listed at the bottom.

VI. Discussion

A. CORK Efficacy

CORK is a software tool that verifies the correctness of compiler optimizations. Given that the verification of compiler

Test	Repeat Period	Expected Result?	CORK	Expected Result?
No Modification	12	Yes	Equivalent	Yes
Loop Modification I	13	Yes	Not Equivalent	Yes
Loop Modification II	13	Yes	Not Equivalent	Yes
Loop Peeling	12	Yes	Equivalent	Yes
Loop Tilting	12	Yes	Not Equivalent	No
Loop Unrolling	12	Yes	Equivalent	Yes
Software Pipelining	12	Yes	Not Equivalent	No

Fig. 7. The results of the seven tests are listed. The "Repeat Period" column lists the values calculated by the C code implementation of the repeat period-finding portion of Shor's algorithm. The "CORK" column indicates the result of CORK's program equivalence analysis of the two WHILE scripts (i.e. the prototype and the script corresponding to that particular test). A "Yes" or "No" accompanies each field, indicating whether or not the result was expected.

optimizations relates to our program equivalence checking proposal, we chose to use CORK as a means of performing program equivalence checking during our testing. Our findings indicate that the compiler verification capability presented by CORK is a step toward our envisioned ability to automatically identify quantum algorithms from an arbitrary block of code. Given a prototype, CORK was able to identify semantically equivalent code in a highly-controlled environment. We thus believe that a basis exists for further development.

We showed in our testing that it is possible to conclude that two programs are semantically equivalent in certain, albeit trivial, tests. The need to translate C code to WHILE code, the limitations of the WHILE language, and other issues all imply that this is hardly a real-world example. Our testing should be viewed more as a proof of concept instead of an example of a real-world program equivalence checking method.

Our tests produced two unexpected results, where CORK did not report semantic equivalence in tests that were expected to yield semantic equivalence. However, the ability to correctly determine semantic equivalence or semantic inequivalence was nonetheless established for multiple tests. Our discussion with the authors of CORK leads us to believe that the two unexpected results were anomalous with regard to CORK's theoretical capabilities. The authors point out that our test environment may have contributed to the unexpected results. In theory, the ability to perform program equivalence checking on these two pairs of scripts (and report semantic equivalence) is not beyond the capabilities of CORK.

B. Implementing Quantum Algorithm Recognition

Our translator must be able to compare the prototype program to all portions of code in the files being compiled. Unlike the highly controlled environment used in our testing, the arbitrary code being compared against the prototype in a real-life environment would be the entire compilation unit. Determining program equivalence in simple cases is manageable. More complex cases, such as those likely to be encountered in real-world scenarios, would potentially be much more difficult, even *undecidable* [11]. However, as previously discussed, the field of program equivalence is advancing.

The process of performing program equivalence checking would likely require the use of a compiler. It is difficult to envision an interpreted language providing the requisite analysis of code for the recognition of quantum algorithms. One possible exception, however, would be the use of a Just-In-Time compiler.

As seen in Figure 8, compilation is a multi-step process. Determining if code is suitable for quantum hardware requires an understanding of the semantics of said code. It is during the syntax analysis phase that a compiler constructs an internal representation of the code structure, typically via a parse tree [12]. The semantic analysis phase utilizes this parse tree in order to check the source program for semantic consistency with the language definition [12]. In addition to the parse tree, the symbol table generated and utilized by the semantic analyzer makes the semantic analysis phase a good place to perform program equivalence checking of quantum algorithms, as the semantic analyzer has the ability to analyze and understand the semantics of the program.

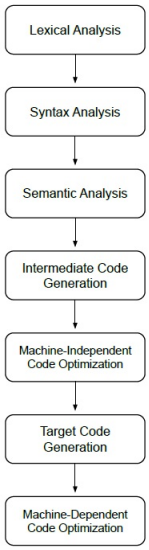

Fig. 8. Compilation Steps

Determining that a block of code should be executed on a quantum computer is not enough; the code then needs to actually be executed on a quantum computer. Using Shor's algorithm as an example, upon recognition of the algorithm by the compiler, the code would be removed and replaced with a request to a quantum computer instructing it to run Shor's algorithm with a given set of parameters, dependent upon the programmer's code. At runtime, Shor's algorithm would be executed on quantum hardware.

The location of this quantum computer might be in the same device as the conventional computer or in the cloud. The latter is more probable in the initial stages of quantum computing. Perhaps some day we will advance to a point where quantum processing units are as ubiquitous as their conventional, modern-day counterparts, and are compact enough to be placed in handheld devices.

We should also consider the possibility that a cloud-based quantum environment could have issues at runtime. Consider the following: several sections of code suitable for execution on a quantum computer are identified during compilation. These sections are replaced with some yet-to-be-determined method of contacting a cloud-based quantum computer with a request to execute a particular quantum algorithm. Of course, this action occurs during compile-time. The compiled program (and its associated quantum portion) won't be executed until later. Suppose that during runtime, the device the code is executing on is unable to contact a quantum computer. This would be a problem if a quantum target was the only option for the code to run on. Perhaps it would be worthwhile to compile this code for a conventional target as well, to account for this potential issue.

C. Dynamic Offloading

The concept of quantum offloading can be taken one step further by using runtime parameters to determine if a section of code should be offloaded to a quantum computer. The overhead associated with offloading may outweigh the performance gain, depending on the algorithm's input parameters. Sufficiently short execution time resulting from particular input parameters may imply that algorithm execution is best left to the local conventional processor, instead of offloading algorithm execution to a quantum processor in the cloud. In other words, the time associated with contacting the quantum computer, execution of the algorithm, and obtaining the results may be greater than performing the computation on the local conventional processor.

This centers around the assumption that the quantum processor is located in the cloud, instead of in the same device as the conventional processor. We make the assumption that the early stages of quantum computing will utilize the cloud-based approach, meaning that a characterization of offloading overhead as well as algorithm execution times with respect to various inputs would likely be useful.

D. Complexity Class BQP

The realm of quantum computing has its own complexity classes, including **BQP** (Bounded-error Quantum Polynomial time), which is essentially the quantum equivalent of **P** (using a more technical definition: **BQP** is the class of all computational problems that can be solved efficiently on a quantum computer, where a bounded probability of error is allowed [13]).

Figure 9 illustrates several complexity classes as many computer scientists believe them to exist. Note the relationship between the various classes. **BQP** does not fully encompass **NP**, yet it also protrudes outside of **NP**. This implies that for some problems, a quantum computer provides no useful speedup in comparison to a conventional computer, while in other cases, a quantum computer provides a potentially massive speedup.

The range of problems in **BQP** has an impact on the usefulness of our automatic program equivalence checking proposal.

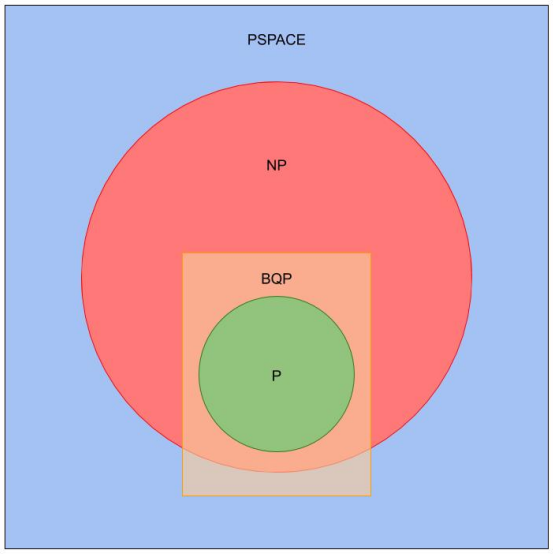

Fig. 9. Depiction of relationship between **BQP** and other complexity classes as currently theorized.

If there are only a handful of problems that are worthy of execution on quantum computers, it would reduce the need for automatic program equivalence checking and make an API-based approach more practical. Even the current approach whereby dedicated languages are used for the programming of quantum computers might be the most practical option in this case.

The need for a programmer to have familiarity with only a handful of algorithms, that are used in very specific cases, is not a lot to ask for. Should the number of problems in **BQP** grow, the automation and abstraction presented in this paper begins to increase in usefulness.

E. Quantum API

A potential argument against the use of program equivalence checking is the fact that APIs are in existence today, and are commonplace among software development. A quick check of the Java Platform API Specification shows just how large this collection of APIs is. Many, if not most, programming languages utilize APIs. Their usage allows for programmers to make a call to a specific library instead of writing the code themselves, eliminating redundancy by providing a more efficient way of writing code. Of course, properly utilizing an API requires knowledge of said API. As an API increases in size, more is required of the programmer in order to understand the potentially large number of libraries to which the API provides access to. A programmer may not realize a library call already exists and write the code himself/herself, resulting in wasted time and a potentially less efficient implementation.

Considering the current number of known problems in **BQP**, the use of a quantum API makes sense. The potential for a programmer to be unaware of a particular algorithm's presence in this API is lessened. Additionally, certain algorithms may be specific to the quantum-world, meaning that an

implementation via conventional programming techniques is impossible. However, as shown by our testing with Shor's algorithm, certain scenarios may allow for an implementation on either conventional or quantum hardware. And while forgoing the use of an API and instead utilizing a programmer's own implementation may result in slightly reduced performance on a conventional computer, the same situation in a quantum environment could result in a drastic reduction in performance, relative to what would be achieved with quantum hardware. In other words, if a programmer does not realize that a quantum API exists and writes an implementation for a conventional computer instead of calling the equivalent quantum API, the loss in performance could be enormous. This is possibly the main attraction to using the automated approach suggested in this paper.

Even if the quantum API model wins out, the use of program equivalence checking techniques could still be of use by means of suggestions to the programmer if the compiler (or some other software application such as an Integrated Development Environment) believes that code written by the programmer could be replaced by a quantum API call.

VII. Conclusion

The concept presented in this paper deals with the use of program equivalence checking for the facilitation of quantum offloading. Instead of relying on the programmer to decide whether code should be executed on a quantum computer or a conventional computer, the goal here is to remove the decision altogether from the programmer and automate the process. Our testing presents a step in this direction. Using CORK, a software tool that implements a novel program equivalence checking algorithm, we were able to perform program equivalence checking on two blocks of code and determine if there exists semantic equivalence between the two. This was repeated for multiple successful tests.

Our idea is dependent upon several key assumptions: the field of quantum computing advancing far beyond present-day capabilities, a programming environment where quantum code and conventional code are interspersed with each other, and (expanding upon the previous assumption) a preference for the quantum and conventional realms to be abstracted away from the programmer, as opposed to an API model where the programmer explicitly decides when use of a quantum computer is necessary. An additional assumption is a sufficiently large number of problems in $\mathbf{BQP}$ that would make an automatic approach to quantum code recognition useful.

Program equivalence checking has important applications in computer science, and thus has attracted interest by means of research within the computer science community. The results of this research can be seen in various research papers and software tools, such as CORK, utilized in our own testing. A basis for further progress thus exists, and with the likely progression of quantum computing, automatic recognition of quantum algorithms could be of use in the future.

Further progress must be made regarding quantum hardware. Problems with noise need to be overcome in order for quantum computers to perform increasingly non-trivial tasks. With regard to computer science, the number of problems that present a significant speedup relative to conventional computers needs to increase. This entails progress made within the computer science community regarding quantum algorithm research. Finally, the field of program equivalence plays an essential role with regard to the idea proposed in this paper. The ability to analyze code via program equivalence checking in a real-world environment and make a determination regarding semantic equivalence is quite a major step compared to the testing performed in this paper. In order for our idea to progress, research into program equivalence is vital.

References

[1] B. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic Program Alignment for Equivalence Checking", *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, Phoenix, AZ, USA*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery.

[2] K. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens, "Verification of Source Code Transformations by Program Equivalence Checking", *Lecture Notes in Computer Science*, vol. 3443, pp. 221-236, Apr. 2005.

[3] S. Kundu, Z. Tatlock, and S. Lerner, "Proving Optimizations Correct using Parameterized Program Equivalence", *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, vol. 44, pp. 327-337, May 2009.

[4] B. Churchill, "Blackbox Equivalence Checking of Program Optimizations", Ph.D. dissertation, Stanford University, Stanford, CA, USA, June 2019.

[5] G. Iooss, C. Alias, and S. Rajopadhye, "On Program Equivalence with Reductions", *Lecture Notes in Computer Science*, vol. 8723, Sept. 2014.

[6] M. Margenstern, "Frontier Between Decidability and Undecidability: A Survey", *Theoretical Computer Science*, vol. 231, no. 2, pp. 217-231, 2000.

[7] T. Harju, and J. Karhumäki, "The Equivalence Problem of Multitape Finite Automata", *Theoretical Computer Science*, vol. 78, no. 2, pp. 347-355, 1991.

[8] C. Alias, and D. Barthou, "On the Recognition of Algorithm Templates", *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 2, 2003.

[9] Nuno P. Lopes, and J. Monteiro, *Automatic Equivalence Checking of UF+IA Programs*, INESC-ID / IST - TU Lisbon 2013

[10] Eric R. Johnston, N. Harrigan, and M. Gimeno-Segovia, *Programming Quantum Computers: Essential Algorithms and Code Samples*, 1 ed. O'Reilly Media, Incorporated, 2019.

[11] Vladimir A. Zakharov, "The Equivalence Problem for Computational Models: Decidable and Undecidable Cases", in *Machines, Computations, and Universality*, M. Margenstern, Y. Rogozhin, Eds., Third International Conference, MCU 2001. Berlin, Heidelberg: Springer, May, 2001, pp. 133-152. vol. 2055, 2001.

[12] Alfred V. Aho, Monica S. Lam, R. Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2 ed. Pearson Education, Inc., 2007.

[13] Michael A. Nielsen, and Isaac L. Chuang, *Quantum Computation and Quantum Information*, 7 ed. Cambridge University Press, 2010.