EFFICACY OF HARDWARE SCHEDULING ON CURRENT GENERATION QUANTUM COMPUTERS

By

Colin Raymond Halseth

B.Sc., University of Northern British Columbia, 2017

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER SCIENCE

UNIVERSITY OF NORTHERN BRITISH COLUMBIA

July 2022

Approval Page

Prepared by Office of Graduate Programs and provided to the Chair of the student's oral examination

Abstract

Quantum computing is a rapidly advancing field of computer science that is increasingly becoming more practical. With these devices becoming more realistic, frameworks are needed by which the hardware resources, both quantum and classical, of quantum computers can be utilized more efficiently. This research aims to fill gaps in the research examining the effectiveness of hardware scheduling on the current generation of quantum computers. A hardware scheduling strategy is implemented using the A* search algorithm for routing qubits to conform with hardware limitations, and this algorithm is tested against a wide variety of quantum programs and devices. The effectiveness of the scheduler is determined through analysis of metrics obtained from the scheduling process. This particular scheduler proved to be effective for most of the tested algorithms and efficient for some, making it useful for general purposes, though some potential sources of improvement could increase the number of algorithms it is efficient for.

Table of Contents

List of Figures

List of Tables

List of Code Samples

Acknowledgements

I would like to take this moment to thank both of my co-supervisors, Dr. David Casperson and Dr. George Jones. Dr. Casperson was kind enough to take me on as his graduate student after my previous supervisor was unable to continue, and Dr. Jones taught himself about quantum computing so that he could teach me. Without the support of both of my co-supervisors, this research would not have been possible.

Additionally, I would like to thank all the members of my defense committee including my co-supervisors, Dr. Alia Hamieh, Dr. William Bowman, and the committee chair Dr. Bill Owen for their interest and their feedback.

Furthermore, I would like to thank Gian Guerreschi. I reached out to him, and he took time out of his schedule to clarify some of the details of his paper "Gate scheduling for quantum algorithms" which served as inspiration for this thesis.

Lastly, I would like to express my endless gratitude to my family who supported me through even the hardest times. To my parents, Greg and Regine, who taught me how to write an academic paper of this scale and sat through hours of practice presentations; to my sisters, Nicole and Cara for their words of encouragement; and to my dog Jessie who sadly passed away during my research but provided me with much emotional support. My family is my biggest driving force and I cannot thank them enough for the hardships I put them though in my pursuit of a Master's degree.

Thank you all

Chapter 1: Introduction

Quantum computing is a rapidly evolving branch of computer science, mathematics, and physics. In order for quantum computers to be practical, several hurdles need to be overcome. One of these hurdles is the development of a portable quantum compiler that can take human friendly quantum algorithms and compile them to a wide variety of existing or near-future quantum computing hardware. Portable compilers require methods to convert from the abstracted computing model used by the programmers to the specifics of a given hardware configuration. Hardware scheduling, which rearranges instructions to exploit the parallelism of the hardware, is one of these methods. This thesis involves research into the development of hardware scheduling strategies in order to create effective quantum compilers.

1.1 Context

Before addressing the purpose of this research, one should briefly understand the basics of quantum computing and how it is both similar to and different from that of classical computing. This background context is important for understanding some of the terminology and uses of concepts that are addressed throughout this research. Quantum mechanics is a theoretical framework, dating back to 1926, that describes how particles behave and interact at microscopic levels [1]. It involves strange phenomena with properties that are not easy to understand [2]. Quantum computing is a new computing paradigm that exploits the strange behaviours of small physical systems that are governed by the laws of quantum mechanics [2]. Because it involves performing computations on quantum computers instead of the traditional classical computers that are so widely used today, there is the potential to develop much faster algorithms than any classical counterpart. For this reason, many companies have been investing heavily towards the development of these machines, including Microsoft, IBM, and D-Wave Systems.

Unlike classical computers, quantum computers operate on quantum bits rather than classical bits. These are more colloquially referred to as 'qubits.' While classical bits are an abstraction for electrons flowing through a wire (1 if electricity is flowing, 0 otherwise), quantum bits are abstractions of any physical two state system [3]. Common examples of two state systems that are used when discussing quantum computers are photon polarity (where the two states are represented as horizontal or vertical polarization) and electron spin (where the two states are spin up and spin down). At any given instance, a qubit can be in a superposition of either of its two states. This means that quantum bits have some probability of being 0 and some probability of being 1, with the probabilities summing to 1, which is representative of 100% probability. It is this idea of linear superposition that makes some quantum algorithms more efficient than currently known classical counterparts because computations can be performed in a way where it appears as if each of the possible inputs to a computation are evaluated at exactly the same time.

Single qubits are not very useful for any real quantum computation. For practical computational usage, many more qubits are needed. Similar to how collections of bits are called registers in a classical processor, collections of qubits in a quantum computer are called quantum registers. Each qubit exists as a linear superposition of two possible states and, as such, can be represented as a vector in two-dimensional space. Because of this representation, the combination of multiple qubits can be represented by a vector in a space that is the Kronecker tensor product[1] of all the qubits' vector spaces [3]. Current generation

[1] The Kronecker tensor product is a generalization of the outer product from linear algebra that results in a block matrix.

quantum computers, like those offered through the IBM Quantum Experience, have fewer than 100 qubits. This is still not enough qubits for many practical computations [4].

As with classical computers, quantum computers need to have operations applied to them to be able to perform work. Basic operations used in quantum computations must be "reversible" [5]. In order to be reversible, one must be able to uniquely identify the inputs of each operation given only the outputs. These reversible operations, which are referred to as quantum gates, are usually represented by unitary matrices on an *n*-dimensional Hilbert space[2] whose size is compatible with the size of the vector for all qubits in the system [3]. By the nature of being unitary, each operation is its own inverse and is, therefore, reversible by definition. Quantum algorithms are created by applying many quantum gates to a qubit in sequence, in order to manipulate the probabilities of each outcome so that it favours certain values. In the quantum computational model, once a computation is completed, a method must be employed to translate and relay the quantum information to either a human or classical machine for review or further usage. This process is performed by collapsing the possible states to one distinct value so that it can then be used outside of the quantum device. The method of obtaining classical data from a quantum computational device is known as "measurement," and it is an irreversible operation that selects a particular state probabilistically from each of the possible outcomes [6].

Similar to classical algorithms, quantum algorithms are sequences of instructions that work together to accomplish a particular task. However, there are some major differences between quantum algorithms and classical ones in terms of input, instructions, and output. In terms of input, classical algorithms assume that the input is given to the algorithm in a logical

[2] An *n*-dimensional Hilbert space is a generalized notion of Euclidean geometric space with '*n*-' dimensions where '*n*' is a positive integer.

form [7]. For instance, a factoring algorithm could assume that its input is a number which it will factor. For quantum algorithms, the input is a little more complicated. First, all quantum algorithms begin in a specific state within the state space created by the qubits used. This state is usually referred to as the ‘0’ state, as all qubits are said to be 0. Input is then encoded onto the state of the qubits using a sequence of unitary gates. This encoding differs for each input. As a result, one could say that input is not being passed to an algorithm as it would classically; instead, the algorithm is actually changing for each different input [7].

In terms of instructions, quantum algorithms are composed of quantum gates and measurements as described previously, which can direct the computer to perform specific tasks. Output is probably the place where quantum algorithms are most different from classical ones. For classical algorithms, the output is always definite. Even if parts of the algorithm are “random”, each random input determines a single computational path. In quantum algorithms, the result of a computation is more equivalent to a slot machine [7]. There is some probability of getting each different possible outcome (winning nothing, winning a small amount, or winning the jackpot, for instance). Hopefully the instructions in the algorithm are designed in such a way as to “hedge one’s bets” to have a higher chance of producing the desired output (ie. the jackpot). To obtain output from a quantum algorithm, one measures the qubits involved, which collapse to definite states based on their individual probabilities, and the values of each qubit are returned. This is much more akin to sampling a probability distribution than simply retrieving an output value. Quantum compiler software would use and manipulate these quantum algorithms to produce executable programs for specific quantum hardware.

At the present time, quantum computing is relegated mainly to the realm of theory, mathematics, and simulation. This is because there are currently few physical quantum computational devices. The devices that do exist are not yet powerful enough for anything but the simplest of computations, since more practical algorithms require more qubits than are currently available [4]. When quantum computers were originally envisioned, they were independent machines completely separate from classical computers that would be used to simulate quantum mechanical phenomena. However, in recent years, this concept has shifted due, in part, to the research that has gone into this field noting that a fully universal quantum computing machine is not something that can be realized in the near-future. In the new quantum computing model, the quantum computing device is considered as a hybrid machine that will act as a coprocessor or add-on hardware component controlled by a classical machine. In a similar way to graphical processing units (GPU), the quantum device will be asked by a classical processor to invoke specific actions or run quantum sub-programs [1, 5, 8, 9, 10]. Before quantum computers can become a viable tool for use outside of laboratory experimentation, many issues need to be resolved, ranging from hardware issues to the software toolchain used by perspective quantum software developers.

At the present time, there is no standard quantum computing hardware and due to the challenges facing quantum computing hardware, there is unlikely to be any standardization for a while. Each of the current publicly accessible quantum computers has its own programming frameworks and compilers, and no portable solution exists for easily porting software from one machine to another.

1.2 Motivation

Research into many aspects of quantum computing is ongoing and rapidly evolving. Much of the body of research in this field focuses on the design of quantum programming languages, assembly, and hardware specifics. As noted in the previous section, near-future quantum computers will most likely not have any hardware standardization and these machines will be limited in computing power due to the very limited numbers of qubits and low degrees of connectivity between each qubit.

To date, many quantum compilers exploit existing classical compiler frameworks, such as ScaffCC, which uses the LLVM[3] framework. It is assumed that optimizations that work for classical programs should work just as well for quantum programs. While this may be true for the most part, some studies have shown that certain types of optimizations do not work as well in a quantum or functional context and other types perform even better [11, 12]. This means that more research is needed in analyzing optimization strategies within the context of quantum computing. One type of optimization is hardware constrained resource instruction schedulers. These particular optimizations would not rely on underlying hardware specific implementations but rather on abstract machines and would take into account the qubits and connectivity in order to maximize resource usage. By creating good optimizations for quantum computers, the limited computing power can be maximized by creating well optimized programs that are capable of running on devices of differing underlying hardware implementations. This area of research has only recently become more prevalent as quantum computers are becoming less theoretical and more practical.

[3] LLVM originally stood for (Low-Level Virtual Machine) but now is just the name of the compiler framework, as the original acronym is no longer totally representative of the project's scope.

1.3 Research Question

This research aims to examine a hardware scheduling algorithm in the context of quantum computing and determine its effectiveness. Specifically, the question this research addresses is whether utilizing the A* algorithm for hardware routing using the gate model of quantum computing is effective for near-future quantum computers. To determine this, the effect of a hardware scheduling strategy is examined against various quantum algorithms when scheduled for several current generation quantum computing hardware. To the author's knowledge, no other publicly available research has examined the effectiveness of a particular hardware scheduling algorithm across a wide variety of quantum algorithms and hardware.

1.4 Methodology

To address the research question, several steps are taken. First, a simple quantum compiler is developed to compile OpenQASM assembly to executable code within a quantum computer simulator or the IBM Quantum Experience. This compiler is designed to be modular so that a hardware scheduling algorithm can be utilized or swapped out if necessary. The second step involves the development of a hardware scheduling algorithm which can be tested. In this research, the hardware scheduling algorithm is inspired by the works of Gian Guerreschi and Jongsoo Park [13]. This hardware scheduling algorithm takes Guerreschi and Park's [2017] broad steps but uses my own interpretation of the effects of those steps. A key difference from their work is that this research uses the A* search algorithm to support arbitrary qubit connectivity rather than only the five-qubit linear arrangement used by Guerreschi and Park. The research involves developing an experiment in which various quantum algorithms are scheduled, with multiple trials, for several different real-world hardware configurations using the previously developed algorithm. The hardware configurations range from a single qubit to 5-qubit configurations with differing constraints, and to a 32-qubit device with no additional constraints. For each of the trials, data is recorded for various metrics and stored in tabular form. These metrics are analyzed to determine if the hardware scheduling for a given algorithm on a given hardware is effective or not.

1.5 Contributions

This research contributes to the discourse on hardware scheduling and quantum computing focused compiler optimizations. Currently, research on the effectiveness of compiler optimizations within the context of quantum computing is sparse, particularly research on near-future quantum computing. The existing research focuses primarily on mathematical analyses of optimization strategies or applies one type of optimization technique to only one specific hardware configuration or quantum algorithm.

This research contributes to this field of research by offering another approach to hardware scheduling compiler optimization for quantum algorithms. The hardware scheduling algorithm implemented in this research combines readability and clarity of implementation with the routing capabilities granted by the A* search algorithm. This allows the hardware scheduler to support numerous devices with arbitrary qubit connectivity configurations. This approach to hardware scheduling is then examined across a wide variety of quantum algorithms and hardware configurations. The benefit is that the overall effectiveness of the hardware scheduling strategy can be determined rather than its specific effectiveness under certain circumstances. This research could be a foundation for further research into quantum algorithm design that would allow for larger and more complex programs to be realized on near-future quantum computers, even considering the limited resources they will possess.

Additionally, there are few comprehensive literature reviews surrounding the topic of quantum computing. This research found only one literature review that covered a wide range of quantum computing concepts. The literature review provided in this thesis is intended to provide an up-to-date comprehensive review of many different aspects of quantum computing from hardware to software, including optimization strategies.

1.6 Outline

The thesis is organized into six chapters. Chapter 2 provides a review of existing literature covering topics from classical computing and their analogues in the context of quantum computing. These topics include “Hardware,” “Instruction Set Architectures,” “Assembly Languages,” and “Compilers.” The “Hardware” section provides a brief overview of how quantum computing works, some of the core concepts of quantum computing, and the current limitations of the quantum computing hardware. “Instruction Set Architectures” discusses what an instruction set architecture is and what research has been going on towards the development of instruction set architectures for quantum computers. Issues that have occurred with designing instruction set architectures for quantum computers are also addressed in this section. “Assembly Languages” covers the research on quantum assembly languages which are used by programmers to interface with the quantum hardware. This section covers the researcher’s thoughts on what is required to make an effective quantum assembly language, as well as the advantages and disadvantages of various currently existing quantum assembly languages. The last section of the literature review focuses on compilers. Specifically, this section discusses how compilers turn programmer friendly high-level languages into executable assembly or machine code. This includes the basic steps that are required, as well as many of the auxiliary compiler tasks that a compiler needs to be considered a good compiler. These auxiliary tasks mainly take the form of optimization strategies, as these are extremely important for producing highest quality programs. Lastly, the "Compiler" section compares several existing quantum compilers.

Chapter 3 of this thesis covers the methodology behind the experimentation performed in this research. This includes the equipment used in the experiment, the process used to create the quantum computing framework and compiler used in this research, the implementation of the scheduling algorithm being tested, how the experimentation process actually works, what kind of data is collected, how the scheduler is validated, how the compiler is run, what each of the experimental trials are, how the data from the trials is analyzed, and lastly, how the experiment ensures it is rigorous, reliable, and valid.

Chapter 4 covers the key findings from the experimental work, as well as the analysis of the data. The first section of this chapter discusses the compatibility of each of the quantum algorithms being tested with each of the hardware being compiled for. Subsequent sections cover findings and analysis for each of the key metrics gathered from the experimental data. These metrics include qubit count, instruction count, estimated run time before scheduling, estimated run time after scheduling, change in the number of instruction stages, the number of added SWAP gates as a result of hardware routing, and lastly, the amount of time it took to schedule the algorithm. The final section of this chapter discusses the effectiveness of the tested hardware scheduling algorithm, which is derived from analyzing each of the key metrics.

Chapter 5 provides a discussion of key findings and analysis from Chapter 4 within the context of the literature from Chapter 2. It highlights the advantages and disadvantages of the tested hardware scheduling algorithm including its clarity, performance, and limitations. Finally, Chapter 6 provides a summary of key findings, as well as concluding thoughts and avenues for future research.

Chapter 2: Literature Survey

2.1 Introduction

The literature supporting this proposal comes from several research areas and was formatively developed through a literature survey report for CPSC 706: Topics in Computer Science Research and Methodology. Since this research involves the construction of a quantum assembly language compiler, literature in the areas of quantum computer hardware, quantum computer instruction set architectures, quantum assembly languages, and quantum compilers is included. The area of quantum compilers includes the many components involved with the compilation process, including optimization strategies such as hardware scheduling. The hardware scheduling section is particularly relevant to this research.

The literature that informs this review was identified through multiple sources. First, several science journal databases were identified through the University of Northern British Columbia's Computer Science databases, including Access Science, ACM portal: the ACM digital library, IEEE Xplore, and Science Direct. These were searched using the terms, "quantum" as well as "programming," "compiling," and "algorithm verification." This brought up a limited number of relevant publications, many of which were outdated. To supplement the search, the websites for IBM,[4] Microsoft,[5] and D-Wave Systems[6] were consulted, as these companies are doing substantial research in the field of quantum computing and provide free online access to research undertaken by their own researchers and affiliated research partners. The search for relevant literature from these websites was limited to items published from 2010 onwards so as to acquire the most recent research on

[4] https://quantumexperience.ng.bluemix.net/qx/community?channel=papers&category=ibm
[5] https://www.microsoft.com/en-us/research/lab/quantum/#!publications
[6] https://www.dwavesys.com/resources/publications

this topic. Since the majority of articles obtained from these sites linked back to papers located at arXiv.org,[7] the next stage of the literature search was directed to articles found specifically on the arXiv database, using the search terms “quantum” and “programming.”

After reviewing approximately 40 articles, the search for relevant articles focused primarily on filling specific knowledge gaps that were identified in previous articles. Additionally, some literature was identified on classical compilers through the same databases used for the quantum mechanical research. However, there do not appear to be many recent publications on concepts surrounding classical compilers, with the exception of a few textbooks from which the relevant areas were included in this review.

2.2 Quantum Computing Hardware

While some quantum computing/processing devices exist and are even available for public use, at the moment, these quantum devices do not yet satisfy the definition of a general purpose computing device [5]. This is because practical quantum computing devices require potentially tens of thousands of physical qubits in order to be of any use in solving anything but simple problems [4]. The hardware used in quantum computing technologies is also much larger than the transistors that make up the basic components of modern classical computing devices [10]. Given this situation, the current state of quantum computing can be compared to that of classical computers in the 1950s, where the devices are large and not yet functional enough for practical uses [11].

At the most basic level, quantum computing hardware requires a device that is capable of performing all the actions described by the quantum computational model. It must

[7] arXiv.org is an online archive of research articles administered by Cornell University Library. The database provides a free open access e-print repository of scientific research topics, including computer science; however, articles must be approved for publication after moderation.

be able to prepare physical quantum systems using superposition and entanglement of individual qubits, as well as store, process, and manipulate registers of many physical qubits [8]. Additionally, since these quantum processing devices will act as coprocessors to classical computational devices, they must also be able to exchange messages between themselves and a host classical processing unit. This coprocessor model is often described as being similar to the modern graphical processing unit where the quantum computer would be able to execute entire sub-programs independently [9]. Other versions of this model rely on a classical processor to control what gates are applied at the right times. The challenge with designing quantum processing hardware that meets all of these requirements is that current methods used for quantum processing devices are difficult to make large and accurate enough [10].

Research is currently going into developing new methods for creating and manipulating qubits in order to overcome these difficulties. One area of research interest is on how qubits themselves can be physically implemented. There are many different research pathways being explored to create physical qubits; for example, some researchers are looking at the use of superconductors, trapped-ions, solid state spin, nuclear spin, non-linear photonics, and neutral atoms [14]. Each of the current methods for physically realising qubits results in inherently fragile qubits that are subject to being disturbed or changed by external influences. Better techniques are needed to guard or protect these qubits from external environmental influences [10, 11, 15].

These phenomena, where qubits are disturbed by external forces, lead to an effect called quantum decoherence; that is, the loss of computational information from the qubits as they become changed and influenced by their environment [16]. The time it takes for a qubit to suffer from too much decoherence is referred to as its decoherence time. Dealing with the decoherence of qubits is a task that quantum compilers may need to perform in order to minimize the accumulation of computational errors. In current machines, qubits typically remain coherent for times that are in the order of 100 microseconds [14].

Current quantum computers experience a non-trivial amount of noise relative to the underlying decoherence rates and this noise severely limits the depth of quantum programs [17]. In many qubit implementations, this external noise is dampened somewhat by keeping the quantum computing devices at cryogenic temperatures [8]. While this method of dampening noise works for devices in the lab, it is not feasible for public computational devices.

Error is also introduced into the qubits whenever quantum operations (gates) are applied to them. According to Gambetta et al. [2017], this is because the physical realization of any quantum gate is not perfect and, therefore, not completely equal to the mathematical definition of the gate [14]. They determined the time it takes to execute a quantum gate, based on experiments, is in the range of 10 to 100 nanoseconds. Given the current decoherence times of qubits, it is possible to perform several thousand quantum gates before a qubit completely decoheres. Errors can also arise when preparing quantum states or during the measurement process, and as noted by McCaskey and colleagues [2018], these sources of error are currently major causes of performance loss [17]. Given this information, several important hardware challenges to quantum computing devices require further research and development, including finding ways to increase the coherence times of individual qubits, reducing errors from state preparation and measurement, and producing quantum gates that are fast and offer high-fidelity computations with optimal control routines. This research is of particular importance for quantum gates that require two or more input qubits, as these quantum gates are often more difficult to accurately control and apply than single qubit gates and, as such, have the potential for more sources of error [14]. An example of these errors can be observed with IBM's quantum computer, using IBM's Q-Experience, which has been shown to suffer from a high degree of error for anything but the simplest of computations [15].

Van Meter and Horsman [2013] indicate that the application of quantum gates is imperfect and that methods need to be identified to minimize the negative effects of these gates [10]. They point to the use of quantum error correction codes as one such method that can be used to minimize error buildup from these imperfect quantum gates. Quantum error correction codes are a series of techniques that allow for calculations to be performed in a more error resistant way. However, error correction codes often require the application of many quantum gates and use of multiple qubits rather than just a single gate and single qubit. In many quantum error correction algorithms, qubits are often organized into groups of more than one qubit. These groups are often used to represent a single error resistant "logical" qubit from a higher-level of abstraction [10]. Since these error correction codes do add some computational overhead in terms of the number of gates and physical qubits used, new quantum error correction codes that have less of an impact need to be developed [18]. Due to the number of additional qubits that are used in quantum error correction codes, current codes are beyond the ability of the existing quantum computing hardware [17]. It should be noted that error correction codes would most likely not be directly used by programmers, but instead be applied by a compiler to efficiently map the physical qubits to logical qubits that programmers would use [11]. In this way, the programmer would not have to deal with adding error correction codes that would result in less readable code.

Aside from the fragility of the individual qubits themselves, considerable research is also going into determining ways in which quantum computers can be scaled up to contain even more qubits, by companies like D-Wave, IBM, and Google. Having a large number of qubits is a requirement for many quantum algorithms, but current generation machines contain few physical qubits. For an ever-increasing number of qubits, the size of the device substrate would need to be increased [14]. While this can be minimized through a creative design, this issue will need to be addressed in order to support the number of qubits that are needed to perform the types of calculations currently being envisioned by algorithm designers.

Another issue that needs to be addressed with respect to device scalability is the degree of connectivity between qubits [8]. Only neighbouring or connected qubits can be used in multi-qubit operations. However, in the majority of devices, not all qubits will be connected together due to hardware fabrication constraints. This makes the inter-qubit connectivity a limiting factor in terms of scalability [8].

Besides general issues that need to be overcome like those discussed earlier, each type of quantum computer implementation has its own specific issues that need to be addressed in order to create scalable devices. In quantum computers based on Ion-Trap technology, the spectral overlap of normal modes can also contribute to the difficulties in scaling up quantum computing devices [18]. In superconductor-based quantum computing devices, the number of wires required for biasing qubits affects the degree to which the devices can be up scaled [18].

2.3 Instruction Set Architectures

The discussion will now move away from the physical quantum computing hardware to instruction set architectures and instruction sets for quantum computers. Instruction set architectures are descriptions of supported instruction sets and they provide the first layer of abstraction from the hardware. They describe the actions the hardware can perform without exposing the hardware specific implementation of each action. The instruction set architecture is the main concept that a compiler needs to be concerned with as it abstracts some, if not all, of the hardware away from the compiler, giving the compiler a much more stable compilation target that can be used on many different machines as long as they all implement the same instruction set architecture. In classical computing, there are two main categories of instruction set architectures. These categories are the “Classical Instruction Set Computer (CISC)” architecture and the “Reduced Instruction Set Computer (RISC)” architecture. The main difference between these two is that RISC describes instructions that use a fixed bit width and these architectures often contain fewer instructions than a CISC device [8]. Both of these architectures come with their own strengths and weaknesses. Determining which architecture is chosen for any particular device is primarily influenced by what the device will end up being used for.

Britt and Humble [2017] discuss some of the strengths and weaknesses of both CISC and RISC architectures when applied to the context of quantum computers [8]. They argue that CISC-based architectures are a good fit for quantum computers from an efficiency standpoint because the non-standardized instruction widths are useful when the instructions exist as predefined functions implemented as hardware. They also suggest that RISC architectures have one feature that gives them an advantage when dealing with nonanticipated problem classes — the ability to utilize their standardized widths to execute more instructions quickly. Each of these classes of instruction set architectures has a potential advantage for use on quantum computers, but neither is completely satisfactory.

One of the biggest issues with instruction set architectures is the addressable number of qubits [8]. One reason why this is an issue is that instructions for both RISC and CISC architectures typically consist of an opcode (instruction operation code) and an address as to where the data being operated on is stored. For quantum computers, each qubit is able to be individually addressed. Consider a two-qubit operation – the format of the instructions would be opcode followed by the first qubit's address, then the second qubit's address. If we consider a 64-bit classical computer controlling a quantum coprocessor, then the width of each instruction would be 64 bits. If there are only 16 possible opcodes in the architecture's theoretical instruction set, and if the addresses of the two qubits are stored in equal numbers of bits in the instruction, then each address would range from 0 to approximately 1 billion (30 bits per address, 4 bits for opcode) [8]. While this number sounds like it would allow for a large number of qubits, due to their fragility and the use of error tolerant 'logical' qubits, this number becomes greatly reduced. The number is also small when compared to the number of bits in a classical computer, where one billion bits is actually less than a single gigabyte and the majority of computers nowadays have many gigabytes. When taking all of these factors into account, Britt and Humble [2017] determine that neither RISC nor CISC are perfectly suited for use by a quantum instruction set architecture, as neither fits neatly into the hybrid classical-quantum device model, though both types do have important features that would be of great benefit for a quantum processing device [8]. They suggest that further research into quantum instruction set architectures should focus less on which category the architecture should fall into and more on developing a new architecture that addresses long-term issues such as the addressable number of qubits. As a result, no current instruction set architecture can be applied to the next generation of machines.

2.4 Assembly Languages

A concept that often goes in tandem with the instruction set architecture and its instruction set is that of assembly languages. Assembly languages are at the bottom of the software development tool chain but are one of its most important parts. This means that assembly level programming is the lowest level of programming that software developers will typically need to know. Assembly languages act as an interface between the instruction set of the hardware and the programmer, and are most often just text-based, human-readable, versions of the instructions provided by the architecture as well as its primitive types (integers, floats). This is true in both classical computers as well as quantum computers. Quantum assembly languages are similar to their classical counterparts in terms of their purpose; however, they use registers of qubits instead of bits [18]. What the exact form a quantum assembly language will take when practical quantum computing devices become prevalent is currently unknown due to the rapidly changing nature of quantum computer hardware. However, a standardized assembly language, as well as a set of software and compiler tools, needs to be developed for when quantum computers do become more commonplace [14]. Currently, researchers have found designing quantum assembly languages and instruction set architectures that support classical feedback and are scalable to large numbers of qubits to be difficult [20].

One of the most popular forms of quantum assembly language today, developed in 2005 by researchers from the Massachusetts Institute of Technology, is known as Quantum

Assembly (QASM) [21]. QASM was originally developed for drawing quantum circuit diagrams in LaTeX[8] documents. However, modern QASM aims to provide a set of basic instructions for use with quantum algorithms that is both compact and expressive at describing quantum circuits [21]. Since its inception, researchers have been developing many different variations of QASM for different purposes. As many of these variations of QASM, as well as other quantum assembly languages, are based on the quantum circuit model, they tend to be hardware independent and are thus ideally suited for use as an intermediate language during compilation [22]. In these intermediate forms, algorithms can be represented by QASM in a way that is close to how they would appear in a quantum circuit diagram. This model of computation closely matches the circuit model used in classical computing where the input qubits are modelled as wires and a distinct set of elementary operations can be applied to the qubits as 'gates' that act on the wires. However, the quantum circuit model has additional restrictions imposed by the laws of quantum mechanics that the classical circuit does not [8]. Some of the most well-known restrictions in quantum computing include the no-cloning theorem, which stipulates that qubit states cannot be copied, and the fact that all the gates must be reversible. Also, algorithms written in an intermediate form, such as the quantum circuit model, typically cannot be executed on real hardware without additional processing steps [20].

Each of the different variations of QASM is designed to expand upon the original QASM and overcome its limitations. The variation used in the publicly available IBM quantum computers is referred to as OpenQASM. It is arguably the most common variation of QASM at the present time due to the public availability of the IBM machines. Other variants of QASM include F-QASM, which was developed to address an emerging need for

[8] LaTeX is a domain specific language for easily writing technical or scientific documents.

quantum assembly languages to support measurement-based classical feedback [23], and Common QASM (CQASM), which was developed specifically to be hardware independent and used as an intermediate representation [21]. One of the goals of CQASM is to be usable by many different types of software, including compilers, simulators, and diagram generation tools. In this way, different programming languages can compile down to CQASM and then be executed on simulators or on real hardware without translating the CQASM code into another format [21]. Some quantum programming languages already implement compilers to CQASM, including ScaffCC, ProjectQ, LIQUi|>, and OpenQL, among others [21].

Rigetti Labs developed another QASM variant, Quil, which is currently being used for their own quantum computers. At the time of writing, they had not yet exposed a public API to it but intend to in the near future. Unlike other QASM variants, Quil enables classical control of quantum programs, a feature that was absent from the original QASM specification [6]. Quil's classical control concepts extend to conditional and unconditional jumps, program counters, halting instructions, and classical interrupts. QASM language dialects such as Quil are based on manipulations of three main atomic types: qubits, classical memory addresses, and classical memory segments [6]. Other QASM versions include QASM-H and QASM-HL [20].

Having so many different dialects of QASM can make translation between each of them time consuming and the code written in each QASM variation not be easily ported to devices that use different QASM variants. As such, some researchers have argued for a standardized QASM variant such as CQASM [21]. However, it must also be noted that none of these quantum assembly languages are without limitations, and until we begin experimenting with them on real hardware, there will likely be even more dialects developed.

For example, Liu and colleagues highlighted several limitations with CQASM, including the lack of classical control flow, classical and quantum instruction interleaving, and quantum circuit reuse [23]. These types of limitations prevent the implementation of some quantum algorithms. Due to these issues and others mentioned previously, current variations of QASM are unlikely to be scalable to large numbers of qubits [8]. Some researchers have developed variations of QASM that minimize these issues through the use of "single-instruction multiple-data" systems, as well as techniques developed to address large numbers of bits in classical computers [20]. Some of the different variations of QASM are listed in Table 2.1 below.

Variant Name	Description
OpenQASM	Most common variant used on IBM platforms
F-QASM	QASM with classical feedback
QASM-H	Module based QASM
QASM-HL	Module based QASM with loop support
Common QASM	QASM as a compiler intermediate language
Quil	QASM with classical control of quantum programs

Table 2.1 Variations of QASM

During the development of any quantum assembly language, regardless of whether it is being used as an intermediate representation or as an executable, it is important to consider the functionalities of the device the language is designed for. Rigetti Labs came up with the concept of an Abstract Quantum Machine while developing their quantum language, Quil. This idealized quantum machine has no hardware specifics and is based on the idea of a quantum computer acting as a coprocessor to a classical processing unit. Most assembly languages are instruction-based, and in an instruction-based language, each instruction will be a single action that will cause a transition on the state of the abstract machine [6].

According to Smith et al. [2016], any quantum assembly language should be able to perform all possible actions on the Abstract Quantum Machine [6]. This includes allowing gates or sub-circuits to be applied to qubits, measuring a qubit, and performing classical control flow such as branching. Any action in a quantum assembly language, based on a particular Quantum Abstract Machine, should have well-defined semantics. Additionally, quantum assembly languages should allow for the definition of new quantum gates that can be decomposed by a compiler into primitive gates, promoting code reuse through the use of sub-circuits (functions / methods) [6].

An important aspect of any quantum assembly language is to provide explicit methods that allow programmers to exploit computational properties resulting from the quantum mechanical nature of the device. One of the most important of these is parallelization, which allows for a large number of gates to be executed at the same time. It is well known that quantum computing provides the potential for extremely high degrees of parallelization. Some languages, like CQASM, allow parallelization to be controlled by the programmer, while others, such as Quil, use implicit parallelization. In Quil's implicit parallelization scheme, the only control the programmer has with respect to how parallelization is run is to create parallelization barriers [6]. By using non-parallelizable instructions, like the "No Operation" (NOP) operation, instructions *before* the barrier cannot be scheduled to run in parallel with instructions *after* the barrier [6].

2.5 Quantum Compilers

An area of research interest related to quantum computers and that relies on the research that has been done in hardware, instruction set architectures, and assembly languages, is the development of quantum compilers. A compiler is a program that translates code written in one language, often a high-level and programmer-centric language, into another language that is able to be directly executed on a physical machine [24].

A compiler follows many different steps as it compiles a program from its source language to the target language. The book, "Modern Compiler Implementation in Java," written by Andrew Appel and Jens Palsberg [2002], has, in my opinion, one of the best breakdowns of the various compilation steps that are employed by modern compilers [25]. Broadly speaking, these steps are broken down into two categories: the front-end and the back-end of the compiler. The front-end typically involves all the steps that deal with the input source code language, whereas the back-end involves all steps dealing with output language.

In the front-end, a compiler performs lexicographic analysis, parse tree generation, abstract syntax tree generation, syntactic analysis, semantic analysis, intermediate representation (otherwise known as IR) generation, and machine independent code optimization. Lexicographic analysis is the first stage in the compilation procedure. Sometimes simply referred to as lexing or scanning, lexicographic analysis is responsible for breaking the input source code into a sequence of meaningful symbols known as lexemes [26]. The output of this stage will be a list of tuples which contain the value from the source file, a type, and any additional meta-data or attributes which will aid in further compilation stages. Additionally, this stage will confirm that all the symbols that exist in the source code are, in fact, valid for the given language. Many programming languages are only valid for sequences of ASCII characters; typically only the letters `a-z`, the numbers `0-9`, and a few special symbols such as `'+', '-', '*'`, and `'/'` . If a symbol is found during this stage that is not valid for the input language, the compiler has the choice of gracefully handling that symbol via some rule set or stopping the compilation procedure entirely. Aho and colleagues [2007] recommend constructing a symbol table at the same time as performing lexicographic analysis, as this information is useful for the subsequent semantic analysis and IR generation stages [26].

Parse tree generation, abstract syntax tree generation, and syntactic analysis are three stages that are often performed at the same time in a single "parsing" stage. This compilation stage takes the lexemes generated in the lexicographic analysis stage and outputs a tree shaped data-structure representing the compilation unit's (source file) structure [26]. This tree is formed from the grammatical structure of the language. A parse tree is a literal representation of the language's concrete grammar, whereas an abstract syntax tree is one that compresses the parse tree and discards information that is not relevant to subsequent stages [26]. While the compiler is forming these trees, it follows the format grammar of the language. Syntactic analysis is performed as the tree is constructed to identify incorrect grammar. Just as with the lexicographic analysis, the compiler can choose to either cancel the remaining compilation stages if incorrect grammar is found, or it can attempt to perform some operations in an attempt to repair the grammar.

Semantic analysis can be performed on the tree that was created in the parsing stage. Semantic analysis is the process of analysing instructions to determine their properties and verify if the instructions indicated by the grammar actually make logical sense [26]. To make sense of this, consider the English language. Just because one can make a syntactically valid sentence does not mean that the resultant sentence actually has any meaning. Examples of semantic checks could include type checking, null reference checks, and ensuring that variables are not being re-defined.

With the program now entirely verified, thanks to lexicographic analysis, syntactic analysis, and semantic analysis, IR code generation can now take effect. With IR code generation, the abstract syntax tree is used to generate new code in what is referred to as an intermediate representation [26]. The IR is chosen by the compiler designer. Three-address-code format intermediate representations are often used in educational materials, including in Aho et al.'s [2007] textbook [26]. When represented in the form of three-address-code, instructions appear visually similar to those seen in assembly languages, with the restriction that each instruction can only contain a maximum of three operands. Many mainstream compilers are built off of the LLVM compiler infrastructure and, as such, they use an IR representation called LLVM intermediate representation (LLVM-IR), which is an intermediate representation that allows a compiler front-end to interface with the LLVM compiler back-end. Many intermediate representations are based on the "basic block" representation of a program. In this representation, the program is considered as if it were a flow-graph, where the edges represent the flow of control from one basic block to another. Each basic block is composed of zero or more instructions that are always executed in sequential order.

Finally, the last operation performed by the compiler front-end is the machine independent code optimization. These are optimizations that operate entirely on the intermediate representation or basic blocks, with no regard for the specifics of the actual machine that will be executing the code. Many of the most common optimization strategies can be applied at a machine independent level. Technically these kinds of optimizations can be done on both the front-end and the back-end of the compiler. It is for this reason that these optimizations are sometimes placed as the first step of the back-end instead of the last step of the front-end. Since these optimizations are not directly related to the hardware on which an algorithm will run, which is what the back-end is mainly concerned with, for the purposes of this discussion, it is included here in the front-end section.

The back-end of the compiler follows the front-end analysis and is in charge of instruction selection, control flow analysis, data flow analysis, machine-dependent code optimization, code emission, code assembling, and linking [26]. The first of these stages is instruction selection, where the intermediate representation instructions are mapped to the specific hardware instructions. Control flow analysis creates graphs that indicate how the flow of control changes throughout a program's lifespan. Sometimes this can mean simply annotating the existing basic block graph with additional information. Data flow analysis is similar to control flow analysis but tracks how data is moved throughout a program. Machine-dependent code optimization attempts to optimize code with the information achieved from the aforementioned analysis passes. After some optimizations are applied, some analyses may need to be re-run as certain optimizations can expose opportunities for other optimizations. Code assembling produces machine code for the target platform, while linking takes the machine code produced by the assembly stage and groups the machine code together into an executable format for the target machine.

A quantum compiler will perform this translation from a quantum mechanically-oriented programming language and "compile" it into a quantum assembly language. These compilers must be able to perform many different jobs as they undertake this translation and do this while abstracting the low-level concerns and physical limitations of the hardware away from the programmer and performing optimizations to code wherever feasible [27].

A compiler that allows for input source code to be re-targeted to a number of physical devices with a reasonable amount of effort promotes portable code and can be classified as a portable compiler; its portability is further aided if the compiler itself can easily be ported to other devices [24]. These portable compilers can minimize the issues discussed in the "Assembly Languages" section because the compiler can handle translation of quantum algorithms to each of the different QASM variants automatically. Chong and colleagues [2017] identify some requirements for quantum compilers [11]. One of the most important jobs that a quantum compiler can perform is implementing quantum error correction codes that help to overcome issues due to the fragility of the qubits. Compilers need to satisfy any quantum mechanical restrictions, such as the no-cloning theorem. Quantum compilers also need to perform qubit memory management on these logical qubits by efficiently reclaiming previously used qubits, as long as they are not in use or entangled to any other qubit in use, as there are a limited number of available qubits on the device.

Quantum compilers require much more knowledge about the problem size of a particular program compared to classical compilers. This is because quantum compilers need to consider a larger number of ways to organize instructions to provide much more aggressive optimizations given the current hardware restrictions for quantum computing devices. Quantum compiling also allows for a greater degree of loop unrolling and function in-lining (discussed in the "Compiler Employed Code Optimization Strategies" section of this chapter) due in part to the higher level of parallelization in quantum computers compared to classical ones; however, the decision around the best time to perform loop unrolling is still up for debate [11].

As with classical compilers, quantum compilers are restricted in what they can do by the physical limitations of the targeted system. Often more quantum gates must be added to circuits so that qubits can be moved around the physical device until operational constraints, such as applying two-qubit gates to neighbouring qubits, can be satisfied [28]. This qubit swapping to satisfy the hardware constraints can impose a significant run time drawback and reduce circuit reliability, in part because each of these swaps will result in a little more error being introduced to the qubits [28]. IBM currently uses a randomized search in order to identify the appropriate mapping of logical qubits onto physical qubits, but other approaches such as employing shortest-path algorithms, such as A*, have been shown to be more feasible for use on larger programs [28].

Decoherence times for individual qubits should also be taken into account by compilers when deciding how to allocate qubits. Individual qubits often have different decoherence times; as such, qubits with shorter decoherence times should be passed over by the compiler for available qubits with greater deocherence times [22]. An example of why this is important can be seen when using the IBM Q5 device. On this device, the third qubit decoheres faster than the others. This means that one cannot perform as many operations on this qubit before losing quantum information due to decoherence. As such, compiler techniques should be developed to efficiently determine when to use short-decoherence qubits and when to avoid using them [22].

Several methods have been proposed to address the issues involved with efficiently mapping higher-level components to elementary operations; however, little work has been done to identify effective methods of satisfying any of the additional constraints of real quantum mechanical computing hardware [28]. A potential reason why there has been little work done in this area is that there is not enough access to device-level information available to researchers outside of the hardware vendor's direct staff [17]. Compilers for quantum computers based on the adiabatic[9] model have additional concerns when it comes to compiling. This includes considering the compilation process as a "competitive game," where different goals are competing to be the most successful, and finding Nash equilibriums[10] for a given program by considering the ideal compilation as one which has the lowest number of graph edges [29]. A future area of research for adiabatic quantum computer compiling includes considering alternative strategies that increase the likelihood of finding the Nash equilibrium.

Auxiliary Compiler Tasks

Besides optimizing and translating high-level languages to low-level error resistant code, compilers are also required to perform many other secondary tasks that aid in the software development process. One such task involves the decomposition of arbitrary quantum gates into gates from hardware specific universal gate sets. The Solovay-Kitaev theorem guarantees that if the hardware set is universal, then there exists an efficient sequence of gates from the universal set for approximating an arbitrary gate [5]. However, even though this theorem guarantees the existence of such an efficient decomposition, no best method has yet been found for actually determining the sequences for arbitrary quantum gates. For the decomposition of arbitrary quantum gates, the compiler should allow the programmer to specify the precision, accuracy, and tolerance levels to use in order to favor

[9] The adiabatic model is a quantum computation model where the algorithms are encoded as the lowest energy state of a quantum system and then allowed to evolve over time towards the lowest energy solution [8].
[10] A Nash equilibrium is a solution to a non-cooperative game in which each player is assumed to know the equilibrium strategies of other players and each player has nothing to gain by changing his/her strategy.

faster compilation times or more accurate results [11]. Soeken et al. [2018] state that quantum compilers should also perform a task called reversible logic synthesis [19]. With reversible logic synthesis, functions of quantum operations can have their inverses automatically determined at compile time. They mention that for larger functions, reversible logic synthesis methods require additional qubits and cannot be determined ahead of time. Synthesis methods that can find a reversible solution without exceeding a certain number of qubits must be developed. Soeken et al. [2018] indicate that the state of this research is still in its infancy [19].

Compilers also play an important role in determining the properties of quantum programs. They should be able to keep track of programmer-described annotations or assertions that can be used in the debugging process to test the correctness properties of quantum programs [11]. This is important because full simulation of quantum systems for debugging is unfeasible for larger than 40 or 50 qubits on most current machines [11]. As a result, the testing of programmer declared assertions about the program's data on real or small-scale virtualized hardware is one of the only ways to currently debug quantum programs. Compilers should also be able to provide resource estimates for a given program such as the number of qubits required, as well as the estimated run time, in order to aid developers in optimizing code fragments [1].

Compiler Employed Code Optimization Strategies

A compiler has to deal with all of the restrictions caused by the physical hardware while also outputting efficient code, often by minimizing the number of low-level operations and maximizing the usage of computational resources such as CPU or memory access. For a quantum compiler, this could mean generating code that minimizes the total number of quantum gates that need to be applied [19]. For classical compilers, many different algorithms have been developed over the years which take higher level constructs and attempt to generate efficient sets of elementary operations that are functionally equivalent. These algorithms, sometimes called optimization strategies, can dramatically affect the performance of compiled code. While they are called optimization strategies, the resulting code may not necessarily be optimal; rather, they are strategies for improving the performance of the code, with no guarantees on how much the code will be improved. For this reason, the term 'optimizing' in this context more accurately means 'improving.'

For a compiler to be able to apply an optimization strategy, it should be designed such that any transformations made do not alter the code's meaning from the original program. Several optimization strategies exist, some of which work in isolation while others work together. Early optimization strategies focused on arithmetic expressions and basic blocks (sequential non-branching code). Later strategies included branches, loops, and procedural optimizations [30]. While all optimization strategies are important for practical quantum compilers, for this research one needs only an understanding of how optimization strategies are generally employed, as the research focuses specifically on one type of optimization – instruction scheduling. As little is written on optimization strategies within the quantum domain and current quantum compilers rely mostly on classical strategies, it is important to this work to discuss some of the common classical compiler optimization strategies. Some of the strategies that can be applicable to both classical and quantum compilers include strength reduction, common subexpression elimination, constant propagation, dead code elimination, inlining, register allocation, loop optimization, and instruction scheduling.

Strength Reduction

Strength reduction is one of the simplest optimization strategies performed by many compilers at a machine independent level. In strength reduction, sequences of computationally expensive[11] operations are replaced by cheaper ones. Sometimes these replacements can result in the elimination of useless operations, such as is the case with algebraic identities like adding 0 or multiplying by 1 [26]. Operations involving exponentiation are more expensive in that they take more time than multiplication, which are more expensive than addition, which in turn are more expensive than bit-shift operations. Strength reduction can take advantage of hardware specific knowledge to determine cheaper operations based on specific hardware abilities. In some cases, operations cannot be reduced in strength. This depends on whether cheaper operations are available on the hardware and the context in which the operations are being performed. While strength reduction is not a loop-based optimization strategy, it is particularly effective when used within the bodies of loops, especially when the operations in question involve computations with the loop index, such as array access, which can sometimes be folded into the target computer's built-in addressing modes [31]. The main improvements that result from strength reduction optimizations come from using less costly operations, as well as potentially reducing the total number of operations required.

Some of the main reasons why strength reduction is so commonly used in modern compiler design are because it is easy to implement and one of the more effective optimization strategies for high-level languages, since code generated from high-level source code abstractions often involve non-optimal sequences of instructions which strength

[11] Computationally expensive refers to the time it takes to complete an operation, the longer an operation takes to complete the more expensive it is.

reduction can clean up [31]. When combined with other optimizations, strength reduction can reduce the number of loop induction variables, and thus the potential demand on registers, decreasing the likelihood of register spilling that can create costly memory access operations.

Common Subexpression Elimination

Common subexpression elimination is a well-known standard simple optimization strategy that has been used by imperative programming language compilers for many years [12]. This optimization strategy is sometimes grouped together with others under a very broad category called "redundant code elimination." In common subexpression elimination, expressions are examined to see if they are repeated multiple times throughout a program. Often, these repetitions are the result of compiler artifacts as the compiler translates high level abstracted structures to a lower level form [32]. If the results are used again, the subsequent redundant expressions are eliminated and replaced with references to the original computed value.

There are several conditions which must be met for common subexpression elimination to be applied. First, an expression can only be eliminated if, regardless of where it is in the program, the expression always produces the same value [26]. Second, the expression can only be eliminated if all constituent components of the expression have remained unchanged since the original assignment was performed [12]. This kind of analysis is often more difficult to do in lower-level programming languages which make use of memory pointers that can point to any location in memory whose value can be changed outside of the current executing context [12].

Common subexpression elimination can be performed as either a local or global optimization. Since local optimizations take place within the confines of each individual basic block, common subexpression elimination is straightforward when implemented as a local optimization. However, implementing common subexpression elimination at a global optimization level, across basic blocks, can be more challenging as expressions can only be eliminated if they have been computed on every possible path leading to the basic block where the expression exists [12].

While this optimization strategy has become standard, it can have different levels of effectiveness for different types of languages. For example, Chitil [1997] notes that implementing common subexpression elimination for functional languages does not result in much of a performance increase to the resulting program, and given the significant additional compilation time needed, its use may not be justified [12]. Since many existing quantum programming languages are functional in nature, the effectiveness of using this strategy in a quantum computing environment may require additional exploration.

Constant Propagation

Constant propagation is another standard optimization technique utilized by many compilers. In constant propagation, constant values are identified and then substituted into wherever they are referenced in the code body. The main purpose of constant propagation is to allow for other optimization strategies to be more effective. In order for constant propagation to be effective, constant folding should be performed, as folding can lead to opportunities for constant propagation to be employed [33]. In constant folding, expressions that are constant at compile time are evaluated into a single value. This folding can lead to the creation of new constants from the folded expressions, which constant propagation can then propagate through the remainder of the code [33].

One of the most commonly used constant propagation algorithms is the sparse simple constant propagation algorithm (SSCP). This algorithm operates on a basic block graph representation in which the variable assignments are in single static assignment form [32]. The SSCP algorithm consists of an initialization phase, where a value is assigned to each applicable single static assignment (SSA) name, as well as a propagation phase, where each constant is propagated into each expression where it is used. During this latter stage, constant folding can be reapplied to potentially create additional constant values that can be further propagated through the remainder of the code. Due to its operation on SSA form code, the sparse simple constant propagation algorithm can be efficient for both local and global optimizations [32].

Dead Code Elimination

Dead code elimination is a type of optimization strategy whereby during compilation, branches of code that are never able to be executed are identified and subsequently removed from the resultant program [34]. Dead code elimination is one of the optimization strategies referred to as local code optimizations. This means that the optimization is usually performed within a single basic block or a small connected group of blocks [26]. In a basic block directed graph representation, the strategy involves deleting blocks that do not have any live variables attached to them. As a result, this optimization requires a live variable analysis to be performed before the optimization can be run in order to identify which variables within each block are alive or not. Knoop et al. [1994] developed a slightly improved version of the basic dead code elimination algorithm that is more effective at optimizing code than older approaches which they called partial dead code elimination [34]. Their algorithm checks not only for dead expressions, but also partially dead ones, where the value of the expression is unused in at least one branch but used in another.

Since there are many different types of dead code elimination, ranging from simply eliminating dead code to performing code transformation like partial dead code elimination, the time complexity of this type of optimization is heavily dependent on the algorithm and can exist anywhere from O(n^2) to O(n^5), depending on the characteristics of the algorithm in question [34]. For Knoop and colleagues' [1994] partial dead code elimination, this time complexity is in the order of O(n^4) in the average case, but O(n^5) in the worst case [34]. However, it can be sped up by altering some of the parameters that control the algorithm behaviour, though this might result in less optimized code.

Inlining

Inlining, known also as function inlining or expression inlining, is a compiler optimization strategy whereby the entire body of a function is duplicated, modified for the current context, and then inserted into a calling location [35]. Manuel Serrano [1997] presents a detailed description of function inlining and believes it to be one of the most valuable optimizations a compiler can perform [36]. The direct benefit of performing inlining is that the overhead of function invocation is completely removed from the compiled executable. Function invocation overhead often involves context save operations, memory fetches, program jumps, and context restoration operations. He notes that operations added to a program by function invocation can sometimes be more computationally expensive than the function body that is intended to be executed. This is why inlining is important as it can remove the function invocation overhead. Additionally, function inlining can offer secondary benefits to further compilation stages. Inlining, in essence, creates specialized copies of functions for each of the contexts where it is used. This means that other optimization strategies can make better decisions because they have more information to base decisions on [37]. Instead of having a function call instruction to consider, they can now consider all the instructions of that function specialized for the current context. The body of the function benefits from knowing the properties of the actual function parameters, which can be used to optimize the duplicated function body after the formal parameters are replaced by the actual parameters. The invoking function can also be optimized by knowing more about the properties of the returned value from the function. Additionally, in the case of languages that support polymorphism,[12] the polymorphic functions may become monomorphic under the application of function inlining.

However, Serrano [1997] also cautions against overusing function inlining because there are several obvious drawbacks to this optimization procedure that need to be balanced with the potential gains [36]. The most obvious drawback is the potential for the resultant size of a program to increase dramatically. Since functions are being duplicated at each invocation point, the number of added operations will be proportional to the number of times that a function is inlined. This becomes particularly pronounced when inlining occurs at a nested level where an inlined function contains invocations to other functions. Because of the resultant larger function bodies, the likelihood of register spilling increases. Additionally, by increasing the number of operations, total compilation time will also increase as other optimization strategies that run after inlining has been completed now have to process more code than they would have before the inlining. Inlined code can also impair code production as high level information such as aliasing must be discarded [36].

[12] Polymorphism refers to the ability of an object to take many forms. In this context it refers to the ability of a subclass to refine the behavior of a parent class's method and have that refined method called instead of the base method whenever an object of the subclass type is used.

In many older languages such as C and C++, the choice as to which functions should be inlined is left to the programmer [36]. The programmer determines which functions should be inlined and then annotates them such that the compiler would know to always inline the annotated function. In C++, the “inline” keyword, which is a part of the function’s declaration syntax, is one example of such an annotation. Misplaced use of inlining annotations can lead to some of the issues noted above. To avoid the potential pitfalls of misusing function inlining, it has been suggested that modern languages should perform automatic inlining by having the compiler decide which functions are appropriate candidates rather than relying on the programmer [36].

Register Allocation

Register allocation applies to a number of program transformations aimed at reducing the overall pressure of programs on the limited CPU registers available. By reducing the pressure on the CPU registers, there is less risk of register spilling and therefore the addition of costly memory access operations, that are required to resolve the spill, can be avoided [38]. In many modern compiler implementations, register assignment occurs on single static assignment (SSA) versions of the control-flow graph representation of the program. Graph-colouring algorithms are used to perform parts of the register allocation process [26]. By making use of the SSA format of the graph, register allocation can be done in polynomial time [38]. Register allocation can occur locally, within the confines of a single basic block, or globally across several blocks as indicated by the control flow graph.

Quantum computers have an optimization strategy similar to register assignment but using quantum bits rather than CPU registers [39]. In quantum computing, this strategy has several names including ‘Qubit mapping’, ‘Qubit routing’, ‘Qubit allocation’, and ‘Qubit movement' [38, 40]. Unlike classical register allocation, there are no worries about register spilling for qubit allocation. In a quantum computer, all qubits are available to be operated on at any given time so there is no concept of CPU registers or memory. Instead, qubit allocation is mainly concerned with minimizing the number of swaps required to perform any given two-qubit operation [41]. Minimizing the number of swaps is important in all near-future quantum computers because of limited geometric connectivity between qubits, meaning that only certain qubits which are considered to be connected or neighbouring can have operations performed between them [38, 41, 42]. In order to perform generic operations between two qubits that are not neighbouring, swaps must be performed until they satisfy the hardware connectivity. However, swapping operations is a slow process and can introduce computational overhead leading to qubit decoherence [38]. As such, logical qubits must be mapped to hardware qubits so that the numbers of swaps are minimized. In many cases, tree-like data structures can assist in finding the shortest swap paths when swapping is required [42, 43].

Quantum compilers must take into account different qubit connectivities, decoherence times, and other compatibility issues between different quantum computer hardware architectures [43]. With the state of current quantum compiler research, most approaches to this kind of optimization make use of similar flow-graph representations as the classical versions, but additionally make use of heuristics in order to accommodate all of the hardware limitations, including the time that instructions take to execute and qubit connectivity [38, 44]. There are other algorithms that have unique approaches to solving the qubit mapping problem such as that of Liu et al. [2019], which uses Markov chains to stochastically perform this mapping when compiling quantum algorithms [44].

Loop Optimization

Since loops are where many programs spend much of their time, loop-based optimization strategies are some of the most effective optimization strategies for classical computers [11]. Some loops are defined within the structure of the programming language itself, such as in Java, C, and C++, with the inclusion of the ‘for’, ‘while’, and ‘do-while’ loop constructs. In other contexts, loops simply refer to areas of code that are repeated multiple times. These implicit loops can be determined by examining the control flow of the program and seeing if a basic block jumps back up to another previously visited block. Loop optimization strategies are designed to optimize code within loops by doing things such as reducing the overhead required to perform a loop or minimizing the number of instructions executed per loop iteration. There are numerous loop optimization strategies, such as invariant code motion, unrolling, fusion, fission, and un-switching.

Like all loop-based optimizations, loop invariant code motion (code motion for short) depends on a series of analyses which must be run against the program’s control flow graph. Loop invariant code motion also works best when the intermediate representation of the program is in single static assignment form, as it becomes easier to analyse and identify candidate expressions in this form [45]. This optimization strategy is one that programmers themselves can do without much loss of clarity to their original code. Loop invariant code motion identifies natural loops in the control flow of the graph that contain expressions whose results do not change between iterations of the loop [30, 45]. These expressions are referred to as loop invariant code because they do not vary as the loop changes. The invariant expressions are then moved out of the loop’s body through a process known as hoisting or code motion. The identification of loop invariant code that is suitable for the hoisting process requires the compiler to perform initial analyses such as using the reaching definition analysis algorithm [45]. The primary purpose of loop invariant code hoisting is to reduce the number of redundant computations in order to increase the program's speed. By hoisting code outside of the loop bodies, any re-calculating of the expressions can be removed, allowing them to be performed only once. The speed-up of the program after instruction hoisting depends completely on the nature of the hoisted instructions and how many instructions can be hoisted. Hoisting more expensive expressions will result in a greater speed-up than hoisting trivial expressions [45].

According to Cooper and Turczon [2012], "loop unrolling is, perhaps, the oldest and best known loop transformation" [32, p. 441]. In loop unrolling, the body of a loop is repeated multiple times with each copy being specialized for a particular range of input values. Each copy must have its boundary logic adjusted for the new ranges. An unrolling factor is used to determine how many times to unroll. A loop with a known number of repetitions can be completely unrolled such that it is completely eliminated and replaced with several copies of the loop body or each of the possible iterations. When applied to nested loops, application of loop unrolling can be performed on either the outer loop or inner loop first. The order in which the loops are unrolled can result in different generated code. Loop unrolling by itself has minimal impacts on the performance of code as it only removes the loop overhead if the loop is completely unrolled. This means that complete loop unrolling can remove branching and bound checking operations from the original code at the expense of a larger resultant program. A larger program will have a larger representation in their IR form, which will increase the compile time, as well as a larger executable size, which can lead to instruction cache overflows if large enough [32]. In the case of excessive loop unrolling, the performance of the program can actually be degraded, as some modern processors effectively perform loop unrolling in the hardware of the machine [46].

With this in mind, loop unrolling serves mainly to aid the compiler in exposing other areas where further optimizations can be applied [30, 33]. It does this by creating a larger set of instructions inside the body of the loop, and therefore more contextual information for optimization and analysis techniques. Better instruction scheduling and pipelining are some optimizations that can be performed for loops once they have been unrolled to produce more optimized code than if the loop is untouched [46]. Additionally, unrolling can lead to improved data locality, which reduces the latency in accessing consecutive areas of memory, expose cross-iteration redundancies that can be eliminated, and change the register allocation pattern [32]. Loop unrolling has direct effects on performance, such as the removal of branching and bounds checking, as well as indirect effects, such as exposing areas for further optimizations. The performance gains of loop unrolling are directly dependent on all of the effects, both direct and indirect. The loop unrolling strategy is often applied with other loop transformation techniques such as loop un-switching or loop fusion [32].

When viewed from a quantum computing context, loop unrolling becomes even more important than it is for classical computations. In "Programming Languages and Compiler Design for Realistic Quantum Hardware," Chong and colleagues [2017] argue that due to quantum computers possessing a much higher degree of parallelization than classical computers, it is more important to perform loop unrolling for quantum computers than it is for classical computers, particularly when combined with other optimization techniques such as function inlining [11]. Some fully-realized quantum compilers, such as the ScaffCC compiler used for a C-like quantum programming language, implement some form of loop unrolling when optimizing quantum programs. In the case of ScaffCC, this loop unrolling is identical to that performed classically. This is because ScaffCC is built off of the classical LLVM compiler infrastructure [41]. For a classical computer, the compiler does not know exactly how much parallelization it can apply for any given hardware. However, for an ideal quantum computer, the compiler parallelization is equal to the number of quantum bits (qubits) within the computer [39]. Additionally, for current quantum computing architectures and compilers, all loops must repeat a well-known number of times to give the compiler enough information to completely unroll loops to the point of removal if the compiler wishes.

Together, these two quantum computing concepts lead to the potential for more effective use of loop unrolling. This is of particular importance when dealing with quantum computing operations that are intended to be applied to every qubit in the system, for instance when applying the quantum ‘X’ operator to all qubits by looping over each qubit. Consider this as a loop with a single operation within the loop. In a classical loop unrolling strategy, this loop could be unrolled once, which reduces the number of iterations in half, or it could be unrolled more than once but the compiler may not know the best number of times to unroll. This will result in a new loop that performs several operations but repeats fewer times. However, for the quantum case, this operation is applied to all qubits and therefore can be completely unrolled into a single operation that can be applied to all qubits at the exact same time. Not only has the loop been eliminated in the quantum case, but all the operations can be amalgamated into a single cycle that can operate on all qubits at once due to the parallelism of the machine being equal to the number of qubits [39]. This behaviour can result in an even larger change in the number of instruction cycles when loop unrolling for quantum computers is combined with instruction inlining, loop fusion, and other optimization strategies. It should be noted that as in the classical context, loop unrolling should be used judiciously for quantum computers, as it is not always beneficial to the compilation of a quantum program [11]. Loop unrolling should only be performed for a small number of iterations or when loops are not nested too deeply, as the number of instructions for nested loops are exponential in the number of loops. Research remains ongoing to determine when and how much loop unrolling to perform for near-future quantum computers.

Loop un-switching is an optimization strategy that applies only to loops containing a conditional statement. The purpose of loop un-switching is to remove additional overhead from conditional statements, including conditional checks and branching [30]. In loop un-switching, the order of the loop and the conditional are switched such that the conditional is performed only once. After the loop and conditional are switched, the loop must be duplicated into both branches of the conditional. This technique can only be applied if the conditional expression is loop-invariant [33].

Loop collapsing is an optimization strategy in which nested loops can be compressed into a single loop. By doing this, the overhead involved with the nested loop is completely removed. Not many compilers support loop collapsing techniques, but there are a few C compilers that are targeted towards the scientific market which do [33].

Loop fusion is an optimization strategy where adjacent loops can be fused into a single loop. The purpose of loop fusion is to reduce loop overhead by completely removing the second loop, as well as to increase the potential level of parallelism and data locality. However, when fusing loops, additional strain is placed on the hardware registers which may cause register spilling [47]. This theoretically improves run time performance; however, in practice this depends on the context of the loops. There are many cases where performing two separate loops can be faster than a single loop, especially when dealing directly with computer memory and data locality [33]. It is for this reason that loop fusion is not commonly supported by many compilers but can be used as a part of different algorithms for other optimization strategies. Loop fusion will end up changing the order in which instructions are executed and, as such, is only valid if all data dependencies are preserved [47].

Loop fission, otherwise known as loop distribution or loop splitting, is the complete opposite of loop fusion. Where loop fusion fuses adjacent loops into a single loop, loop fission breaks a loop up into multiple adjacent loops [47]. Like loop fusion, data dependencies need to be preserved since the order in which instructions are executed will be changed from the original program. The main advantages of performing loop fission come from an increase in the potential for utilizing hardware instruction pipelining or exposing groups of operations as vector computations on vector computers. These concepts are important for both modern single instruction multiple data (SIMD) machines, as well as for potential near-future quantum computers. Besides the main benefits of loop fission, there may be a reduction in cache misses and pressure on register allocation [47]. Loop fission also enables other strategies such as loop interchange.

Loop interchange, loop permutation, or loop reordering are loop transformations that swap the inner and outer order of loops, particularly nested loops. The main purpose of loop interchange is to improve data locality, which depends on the programming language and the machine being compiled for [47]. Bacon et al. [1994] states that while this exchange process does not directly lead to any immediate performance gains in the compiled program, it can lead to exposing different situations which can allow for the application of other optimization strategies [30]. Loop interchange is a legal transformation only if the dependencies of all instructions in the loop remain legal.

Loop inversion is a loop optimization strategy that applies specifically to while[13] and do-while[14] loops within C-like languages. In loop inversion, a while loop is converted to a do-while loop, which is then surrounded by a conditional statement that checks for the loop entry condition [48]. The main advantage of this optimization is to allow fall through of the code from the loop body after the loop condition has been satisfied. In a traditional while loop, at the end of a loop the program must jump back to the beginning of the loop, check the loop condition and then, if the condition is satisfied, jump to the end of the loop to continue the rest of the program. These two additional jumps can be completely removed from the while loop by using loop inversion. While this does not result in much direct time savings, the effect is compounded for nested loops where the number of jumps can grow considerably. Unlike some loop optimizations, loop inversion does not create any negative effects or loss of performance, and therefore can be used more liberally than some of the other strategies [48]. Additionally, loop inversion, like other loop strategies, creates the potential for other optimization strategies to be utilized more effectively.

In loop skewing, the data dependencies of a loop are altered by skewing the loop index, which can remove some blockers to identifying loop code that is parallelizable [49]. Loop tiling rearranges loops such that blocks of data can be accessed sequentially with fewer jumps in memory. This is done by exploiting the spatial and temporal locality of the data involved in the loop body, such as for array accesses [47]. One of the more powerful aspects

[13] While loops check a condition and when that condition is true, the code is repeated.
[14] A do-while loop as similar to while loops but the condition is checked after the repeated code rather than before.

of loop tiling is the ability to target different levels of memory, including caches, RAM, or hard drive, allowing for maximum reuse of data at specific levels of the memory hierarchy depending on the chosen block size [47].

Loop parallelization refers to a class of transformations that is designed to restructure loops and their content in order to allow them to be executed in parallel on vector machines and multiprocessor architectures [50]. These techniques involve both transforming loops to expose potential parallelism opportunities, as well as accounting for data distribution and communication in order to generate efficient parallel programs. Many loop parallelization algorithms focus on transforming nested loops [50]. Loop parallelism algorithms can include loop interchange, loop fission and fusion, as well loop skewing and loop tiling.

A major goal within many optimizing compilers to achieve high-performance is to discover and exploit parallelism in loops [30]. As mentioned earlier, quantum computers offer the potential for a higher degree of parallelism than most classical computers due to being able to operate on most, if not all, qubits at the same time. Taking this into account, Chong et al. [2017] indicate that quantum compilers would be able to perform optimizations that are more aggressive than their classical counterparts [11].

Instruction Scheduling

Instruction scheduling is a compiler optimization that is highly dependent on low-level information about the system being compiled to and the length of each of the instructions that can run on the architecture. Instruction scheduling is a subcategory of Automated Planning and Scheduling[15] [51]. It is important only for central processing units that operate with instruction pipelining, which includes most modern processers [52]. This

[15] A set of techniques in which computers automatically plan the based schedule on constraints given by each of the events being scheduled, such as the time to complete the event.

optimization will have no effect on the overall performance of the program for processors that do not have instruction pipelining, as all instructions will be executed entirely sequentially. Instruction scheduling will rearrange basic operations in order to reduce the frequency of pipeline interlocks [52]. To understand why rearranging these basic instructions can increase the overall speed of a program, one must have a basic understanding of how pipelining works in a processor.

When a processor is performing an instruction, what it is doing can be broken down into three stages: fetch, decode, and execute. In a traditional processor with no pipelining, these stages are executed sequentially for each instruction in the program. Every instruction must complete before others can start. In a processor that supports instruction pipelining, different instructions could be in different stages of processing because they can be run independently without worry about when the other instructions complete. This increases the throughput of the processor because it can, in effect, work on three instructions at once, as one instruction does not block other instructions from starting the process. However, some instructions may still require that other instructions be fully completed before they can start. These instructions are referred to as pipeline hazards and are usually in the form of register or memory-based instructions [52]. While instruction scheduling can be performed by hardware, it is limited in scope and can be expensive, which makes performing this operation at compile time much more practical and effective, albeit at the cost of some additional compilation time.

Instruction scheduling algorithms must consider two main issues [52]. The first is how to express the constraint that must be satisfied by any legal reordering of instructions, as some instructions cannot be reordered without changing the semantics on the program. The second issue is how one can determine which instruction order is the most impactful, subject to the aforementioned constraints, without adding too much time to the overall compilation process. Some scheduling algorithms have a basis in graph theory.

Gibbons and Muchnick [1986] developed an algorithm for classical instruction scheduling that operates after register allocation rather than at code generation, on a specially constructed directed acyclic graph, which is derived from the program's basic block representation [52]. This graph representation contains the instructions as nodes of the graph, with the edges between the nodes representing the serialization dependencies. As long as the instructions are executed in some topological order, then the semantics of the original code remain preserved. With that in mind, Gibbons and Muchnick's [1986] algorithm begins to schedule the instructions so as to reduce number of pipeline hazards [52]. To do this, the instructions to schedule are picked such that they do not introduce any new pipeline hazards. If that is not possible, then the instruction will be chosen so that it is more likely to interlock with subsequent operations based on the assumption that these instructions may be able to remove or minimize hazards. Three heuristics govern the selection behavior: 1) whether an instruction interlocks with its immediate successor; 2) the number of immediate successors; and 3) the length of the longest path from the instruction to the leaves of the graph. The run time for Gibbon's and Muchnick's algorithm in the worst case is O(n^2) for a basic block with N instructions, which they claim is faster than other similar algorithms which have a O(n^4) running time [52]. Their algorithm can also be easily modified to add no-op[16] operations in order to completely remove hazards.

[16] A no-op instruction is an instruction that performs nothing (no operation) and can be used to delay subsequent operations.

In quantum computing, the task of instruction scheduling has been referred to by different names, including Temporal Planning [51] and occasionally Instruction Hardware Mapping, since schedulers must also determine which physical qubits are mapped to each instruction [53]. When applied to the context of quantum computers, instruction scheduling techniques can become even more important than they are in a classical context. In the current generation of quantum computers, one of the main obstacles to practical computations is the timeframe over which qubits decohere and are no longer applicable to use in algorithms [54]. This means that the run time of an algorithm is extremely important on quantum computers in order for the computation to be successful. The goal of scheduling in the context of quantum computers is to reduce the latency of the program as well as the effects of noise to minimize the decoherence qubits experience, thereby increasing the chances for the program to return correct results [53]. Classical instruction scheduling techniques should still apply to quantum algorithms, at least while the program is being represented at the quantum gate level. The reason these techniques can still be used on quantum assembly is that it is similar to classical assembly in terms of being laid out by basic instructions from a given instruction set architecture. However, it is slightly more complicated to perform scheduling for quantum computers mainly because many operations can be performed at the same time, similar to many SIMD machines. Given the automated nature of instruction scheduling for near-future quantum computers, using artificial intelligence to assist in instruction scheduling has become an open problem which has begun to attract the attention of researchers within the artificial intelligence community [55].

Instruction scheduling algorithms for quantum computers often operate on sequences of instructions that are already compatible with the underlying hardware [13, 56]. It is assumed that an earlier compiler pass would break incompatible instructions down to optimal sequences of compatible instructions, such as was done by Booth [2018, 57]. This also implies that there are no branches and loops have been completely unrolled [58]. Due to the limited error correction[17] and quick decoherence times present in these machines, instruction scheduling is one of the most important optimizations for quantum compilers to perform, especially for near-future quantum computers [39, 41, 59]. In order to minimize the risk of decoherence from occurring, quantum algorithms should be optimized to take less time to run. Instruction scheduling should be set up to reduce program run time as much as possible by optimally scheduling instructions. It operates similarly to how it would in a classical sense where valid instruction schedules are limited by the data dependencies of each instruction. However, for quantum temporal planning, there are additional constraints that complicate the scheduling operation.

Quantum computer constraints can be grouped into three categories: logical, exclusivity, and connectivity [13]. Logical constraints are based on the order and dependencies of quantum gates in the algorithms, taking into account that some operations commute with each other and can have their order swapped, thus changing the dependencies of each operation.[18] Exclusivity constraints involve constraining the number of operations that can use hardware resources at the same time. For instance, resources such as physical qubits can only be involved in one quantum operation at a time. Connectivity constraints deal with the underlying qubit communication channels and qubit adjacency of the hardware.

[17] A set of techniques which can be used to limit the influence of errors in a quantum system. For example, the Steane code uses seven physical qubits to represent one logical one and can tolerate an arbitrary error in any one qubit without effecting the computation [57].

[18] If matrices A and B commute, $AB = BA$, then the corresponding quantum operations can be reordered and thus do not depend on each other [58].

Quantum scheduling should also take into account that some instructions can cancel each other out when scheduled next to each other [60]. Additionally, some gates, such as the Hadamard Gate, can hinder the commutativity of other gates [60]. Still another constraint is determining which instructions can be executed in parallel [51].

In scheduling algorithms, time can be considered one of two ways. It can be treated as either discrete units referred to as CPU cycles, with the run time of a particular instruction as the number of CPU cycles needed to complete the instruction; alternatively, time can be taken precisely as the exact real-world time of each instruction recorded for the specific hardware being used. In Dousti and Pedram's [2014] algorithm, the latter concept of time is used, whereby various operations are given specific completion times in microseconds (μs), such as moving a qubit takes 1 μs, a gate takes 10 μs and a multiple-qubit gate takes 100 μs [53].

In contrast the algorithm developed by Guerreschi and Park [2017] uses the former concept of time, dealing only in abstract CPU cycles in which operations on single qubits take a single cycle to complete, while operations on multiple qubits require at least two cycles [13]. In their algorithm, each instruction is recorded in an instruction dependency graph. Traversing this graph allows for each instruction to be prioritized based on the number of cycles required to complete the operation, as well as the number of cycles used by any instructions that depend on it. This prioritization based on the instruction timing is at the core of their scheduling implementation. Scheduling instructions then occur in order of priority, where graph colouring and subgraph isomorphism are used to resolve hardware connectivity constraints [13].

Certain classes of quantum algorithms respond better to temporal planning and scheduling optimization strategies. The Quantum Alternating Operator Ansatz class of quantum algorithms has a very high number of commutative gates, allowing for temporal planning and scheduling algorithms to be able to take advantage of this commutativity for more aggressive optimizations, though the added commutativity also leads to challenges in determining the optimal schedule when there are too many groups of commutative operations [51].

Another approach to instruction scheduling for near-future quantum computers comes from the domain of constraint-based programming in which the primary objective is to find a scheduling solution given all the constraints, while the secondary objective is to minimize the number of added swap operations. While constraint programming is very promising for many problems, researchers have shown that pure constraint-based programming does not scale as well for instruction scheduling when compared to industrial temporal planning algorithms [55]. This has led to the development of hybridized algorithms which combine constraint-based programming with temporal planners in order to better utilize the strengths of constraint-based programming while removing some of its weaknesses with scalability.

While the above work indicates that instruction scheduling can work at a gate level, some researchers suggest that this is not the best conceptual model for scheduling quantum algorithms. This is because the gate model is in itself an abstraction of how quantum gates are actually implemented in real quantum hardware [54]. In real quantum hardware, gates are not implemented as discrete operations as they would be classically. Instead, these operations are actually encoded as electrical pulses that control things such as the magnetic fields that will change the state of the qubits [54]. This difference makes temporal planning potentially even more effective in a quantum computing context by taking into account how the hardware actually performs quantum operations.

To deal with this disconnect between the gate model and the underlying gate implementations, Shi and colleagues [2019] developed a new strategy for instruction scheduling which aggregates multiple operations into a single control pulse and then schedules the aggregates [54]. This process involves finding groups of instructions that are commutative, creating custom control pulses from those instructions as if they were a single instruction, and then schedule sending the control pulses. In their experimental setup, they obtain an increased speed in the order of 5-10 times on the algorithms being tested. This speed increase is directly related to the level of commutative operations within the algorithm in question. From their experiments, Shi et al. find that without creating aggregate control pulses, each pulse must be sent individually [54]. This introduces additional latency to the program that decreases the output qubit fidelity, which decays exponentially with increased latency. Reducing this latency by sending fewer instructions to the quantum computer can help ensure that the quantum computations finish before the qubits decohere [54].

While the algorithm implemented by Shi et al. [2019] does not account for the connectivity between qubits, the researchers note that the algorithm could easily be modified to account for the qubit connectivity [54]. The biggest challenge with the instruction aggregation technique is that it could create conflicts with other instructions such that the aggregates cannot be run at the same time, as the aggregated instructions could apply to multiple qubits at a time. Parallelism is an important feature of quantum computation and so aggregated instruction scheduling should be used carefully, with several analysis passes, to avoid causing too many detrimental effects to the potential parallelism.

Existing Quantum Compilers

Quantum compilers are programming language compilers that can read and run quantum algorithms. These compilers can rely on all of the above fields, assembly languages, instruction sets, and optimization strategies. Several compilers currently exist for the first generation of quantum programming languages (including ScaffCC, ProjectQ, Q#, XACC, etc.). Most of these compilers support compilation to some variation of QASM, as QASM is currently considered the standard for quantum assembly [11]. Since these compilers require considerable information about the execution of the program being compiled in order to perform such heavy optimizations, many of the existing quantum compilers for quantum programming languages fall within the domain of "functional" programming languages.

Functional languages are often considered to have much richer type systems[19] than imperative languages do, as well as stricter rules when it comes to side-effects.[20] A rich type system has data categorized by type and that type contains considerable information about the properties of that type. In a rich type system, program data and the use of operations are governed by these types and can be verified for correctness by the compiler. These rich type systems and restrictions allow compilers to perform better program validity checks at compile time rather than relying on runtime systems to handle errors [1, 2]. This may be the reason why much of the recent research in quantum programming languages has led to the development of functional quantum programming languages like Quipper and Q#.

Besides the work done by independent researchers to develop quantum programming languages and their associated compilers, quantum hardware vendors have also been creating

[19] Automated systems that classify data, usually based on programmer annotations, into particular "types" (such as integer or object) and perform analysis between types and their usage.
[20] Side-effect: an additional or unexpected alteration to a program's state which is able to affect the rest of a program outside of the current executing scope or method.

proprietary languages and frameworks for their devices [17]. This gives rise to concerns over the ability to write portable code that can be used on many different physical devices. McCaskey and colleagues [2018] believe this may lead to artificial barriers in the adoption and performance of quantum programming languages and compilers due to the lack of inter-operability provided by these proprietary frameworks [17].

Comparison of Open Source Quantum Compiler Frameworks

The proposed research in this thesis requires the use of a quantum compiler framework. This section compares the strengths and weaknesses of several existing quantum compiler frameworks. Given the lack of availability of proprietary frameworks, only publicly available frameworks are under consideration. This limits the discussion of compiler frameworks to projects such as ScaffCC, ProjectQ, and XACC.

ScaffCC is a scalable compilation and analysis framework based on the high-performance LLVM compiler framework. ScaffCC is the compiler front-end for the scaffold programming language designed for quantum/classical hybrid algorithms that output a custom type of scalable quantum assembly representing the compiled algorithm [61]. ScaffCC is designed to scale effectively to extremely large programs. As with many of the other publicly available compilers for quantum computing, ScaffCC operates on qubits at a 'logical' level, which means that the qubits do not accurately represent physical qubits and no error correction has been applied. Javadi and colleagues [2019] claim that ScaffCC can operate at this level rather than at the physical level because any optimization applied at a logical level will likely have a multiplicative effect on the required computing resources for the physical level [61]. Given that ScaffCC is based on an extended version of LLVM and

Clang, this compiler has access to very mature compilation tools and processes, including all of the classical optimization strategies provided through LLVM.

The ScaffCC compiler has been verified against several different quantum algorithms, each possessing elements that are common to many more algorithms. These elements include the Quantum Fourier Transforms, classical oracles, state distillation, random walks, amplitude amplification, and more.

One of ScaffCC's assets is its use of professional quality and mature compiler infrastructure technologies available within the LLVM framework. This quantum compiler is likely one of the best currently available to the public since it uses efficient, powerful, and widely used compiler techniques. However, the use of the LLVM framework may also be one of ScaffCC's biggest weaknesses. LLVM and Clang are very large and complex software packages that are only well known by a few developers. As a result, compiler extensions are more difficult to develop, drastically reducing the number of extensions that can be developed by the community. This is because an extensive working knowledge of the LLVM framework would be required in addition to ScaffCC's compilation process and project structure.

ProjectQ is an open source software solution for designing, testing, and running quantum algorithms through the use of an extensible compiler framework. Developed by Steiger, Häner, and Troyer [2018], this project is integrated into the Python programming language [62]. When executing Python scripts using the ProjectQ framework, quantum mechanical operations are cached and then passed through a series of optimization stages and transformations as part of the compilation process. The resulting code can then be tested in a quantum computer simulator, analysed using the circuit drawer and resource counters, or run on real hardware through API's provided by IBM's Q-Experience.

One of the biggest strengths of ProjectQ is that it is both simple and highly extensible. With this framework, developers can create new quantum gates as Python objects, controlling how they are decomposed, and they can choose which compiler passes are used or tailor the passes for particular needs. This allows programmers using ProjectQ to determine the effect of each compiler pass by using a resource counter, circuit drawer, or instruction printer before and after each pass [62]. These compiler passes also allow for layered abstractions which let developers ignore the specifics of quantum hardware and instead focus entirely on the algorithm. Programmers can then develop algorithms without having to worry about the hardware that will run the algorithm, and each pass of the compiler will slowly manipulate the algorithm to conform to all of the constraints of the backend hardware.

Unlike traditional compilers, ProjectQ cannot examine all possible paths an algorithm can take. Instead, each time a quantum operation is called, it gets cached to be processed by the compiler. This behavior likely means that in many cases, the classical logic acts more like preprocessor directives in C rather than actual parts of the compiled algorithm; potentially imposing limits on the types of optimization passes that are possible in this framework. Some examples of these limits would be for conditional branches where only one branch would be executed and cached by the compiler, or loop structures where the loop is entirely unrolled for the given loop conditions rather than treated as a looping structure. Adding compilation stages that use branch optimization or loop optimization strategies means that these kinds of control structures would need to be emulated as custom Python classes in order for the

ProjectQ compiler to be able to operate on these control structures. However, due to the simplicity of adding new passes and back-ends, there are several extensions to ProjectQ already available on GitHub.

XACC is a quantum algorithm compilation and execution framework developed under the Eclipse Foundation[21] umbrella. XACC models quantum algorithm execution as a coprocessor model similar to how OpenCL or CUDA (Nvidia GPU parallel computing platform) work for graphical coprocessor programming [17]. In this model, the program is a classical program which can offload quantum mechanical 'kernels' to a supported device. This 'kernel' offloading allows a classical program to pause and wait for the execution of a quantum subroutine to be completed. In XACC, a 'kernel' is a small segment of code within a quantum programming language that is designed to be executed on quantum mechanical hardware. These 'kernels' can be written in many different quantum programming languages, including Scaffold, Quil and others. XACC is written in C++ with the goal being to make it as easy as possible to create bindings for other languages. In this way, developers will not have to learn new languages to use XACC. At the time of writing, XACC could be used on C++ or on Python bindings; however, plans exist for use with other languages such as FORTRAN.

Of all the compiler frameworks discussed, XACC currently has the largest number of supported back-end targets. In XACC, back-end targets are referred to as 'accelerators' which are used to run compiled scripts. XACC currently supports several public quantum computer simulators, as well as several real quantum computers, such as IBM's Quantum Experience and D-Wave's quantum annealer. The overall architecture is similar to that

[21] The Eclipse Foundation is an independent, not-for-profit corporation that acts as a steward of the Eclipse open source software development community.

employed by Project-Q, where quantum algorithms pass through a chain of optimization passes until they get passed off to a back-end interface / accelerator. Unlike ProjectQ, XACC supports pre- and post-processing of quantum kernels, which the developers believe could be used in error mitigation strategies [17]. One of XACC's greatest strengths is its ability to compile multiple quantum programming languages and run them on multiple supported back-ends. XACC does not seem to have the same weaknesses such as ScaffCC's complexity or ProjectQ's potential to not be able to perform certain types of optimizations; but XACC also possesses neither of their strengths, such as the powerful and mature optimization framework employed by ScaffCC or the simplicity of use of ProjectQ.

2.6 Summary

This literature review identified several issues that need to be resolved before practical quantum computers can be developed and fully utilized. The hardware of quantum computers is rapidly evolving but still faces some key challenges, and these have an effect on every part of the software development toolchain, including compilers. Quantum compilers will need to support code portability, especially in lieu of how rapidly quantum hardware, instruction set architecture, and assembly languages are changing. Quantum compilers also need to consider many hardware details such as the connectivity of qubits, as multi-qubit operations often can only be applied to neighboring qubits. This means that not only does the compiler need to generate equivalent code for a particular instruction set architecture, it also needs to add the smallest number of swaps possible in order to maintain performance while still satisfying the neighboring connectivity issue. Quantum compilers should also take into account individual qubit decoherence times, apply the correct error correction codes, generate sequences of built-in gates for any given arbitrary programmer defined gate up to a configurable level of precision, and estimate the number of qubits and execution time for a particular quantum program. More work needs to be put into developing compiler frameworks that are extensible and embrace code portability, as little work has gone into this area of research. In order to generate efficient machine code, quantum compilers will rely on many different optimization strategies.

Many classical compiler optimization strategies will be applicable to quantum algorithms and will likely be used by quantum compilers. While all types of compiler optimization strategies will be important, loop optimization strategies and instruction scheduling will be particularly important due to the increased levels of parallelism available for quantum computers to be able to exploit compared to their classical counterparts.

Chapter 3: Methodology

This research involves the creation of a quantum algorithm compiler, based on the gate model of quantum computing, which is capable of reading and analyzing quantum algorithms, and then conducting an experimental comparison of a specific hardware scheduling algorithm and its effects on various quantum algorithms. The goal of this research is to assess the effectiveness of this particular hardware scheduling algorithm for general usage. This chapter discusses the research methods that were employed, details about the hardware and software used to perform this research, and the data analysis methods.

The chapter starts with a summary of the procedure employed by this experimental work, including the development of the framework required to perform the research, the experimental set up, and the collection of the experimental data. Next, the implementation methodology is discussed to provide insight into the design of the hardware scheduling algorithm which is employed for this research. The experimental setup and procedure is explained, along with a description of the methods used to analyze the experimentally generated data. Additionally, this section discusses how the research ensures rigor, reliability and validity.

3.1 Procedure

The procedure for this research involves software development, experimental work, and some comparative analysis. A quantum compiler for the OpenQASM programming language is developed, using a hardware scheduling algorithm inspired by the work of Guerreschi and Park[22] [2017, 13]. This scheduling algorithm aims to abstract physical qubit

[22] Code for Guerreschi and Park's [2017] scheduling algorithm is not publicly available. All code for this version of the scheduling algorithm is based solely on my interpretation of the broad steps outlined in their paper.

placement away from the developer, allowing him/her to write quantum programs without worrying about hardware specific configurations. To test the efficacy of this particular hardware scheduling algorithm, experiments are performed in which quantum algorithms are optimized for different kinds of real quantum computing hardware, with various metrics gathered.

Equipment

The majority of the work done for this research is on a standard personal computer capable of running code written in C#, with the aid of several software packages. A Dell computer running an Intel Core i7 8700K @ 3.70GHz processor with 32 GB of DDR4 Dual channel RAM is used to run the experiments. The installed version of C# on the computer is .NET Core 3.1.301. On this setup, the Stopwatch class has an accuracy of 10000000 ticks per second, or 1 tick every $1 \cdot 10^{-7}$ seconds, as determined by the Stopwatch.Frequency field.

Development of a quantum computing framework

The first step in developing a hardware scheduling algorithm is to determine the quantum computing framework the scheduler will be developed within since the scheduling algorithm relies on the core components of whichever framework is chosen. There are several quantum computing frameworks available for developing the hardware scheduling algorithm on top of. Some of these frameworks include ProjectQ,[23] ScaffCC,[24] XACC,[25] and QisKit.[26] Each of these frameworks has their own advantages and disadvantages. ProjectQ, for instance, has one of the simplest frameworks to use, in my opinion, but does not contain the ability to load quantum programs written in the OpenQASM assembly language, which is

[23] Project Q official website at https://projectq.ch/
[24] ScaffCC official website at https://github.com/epiqc/ScaffCC
[25] XACC official website at https://github.com/eclipse/xacc
[26] QisKit official website at https://qiskit.org

how many of the quantum algorithms are commonly distributed. ScaffCC is likely the most comprehensive framework of the three as it is based on the LLVM compiler infrastructure which, as a result, also makes it the most complicated to understand and modify. Additionally, ScaffCC cannot load OpenQASM files or run them against the IBM Quantum Experience platform. Instead ScaffCC takes programs written in a special C-based quantum programming language called Scaffold and compiles them to an offshoot of OpenQASM, which it can run in a local quantum computer simulator. Additionally, many of these frameworks are written in either C++ or Python, languages with which I have limited experience, potentially slowing down the development of my scheduling algorithm or resulting in a scheduling algorithm with lower quality code than using a language I am more familiar with. A summary of some of these frameworks can be seen in Table 3.1 below.

Framework	Compiler	Language(s)	Complexity	Url
Project Q	Python	Python DSL	Low	https://projectq.ch/
ScaffCC	C++	Scaffold	High (LLVM)	https://github.com/epiqc/ScaffCC
XACC	C++	Many (OpenQASM)	High	https://github.com/eclipse/xacc
QisKit	Python	Python DSL	Low	https://qiskit.org/

Table 3.1 Quantum Computing Frameworks

For the reasons discussed above, a custom framework is developed in the C# programming language, inspired by the aforementioned frameworks. While not as fully featured, this custom framework contains only what is needed to be able to develop a scheduling algorithm without layers of additional complexity. This includes the ability to load OpenQASM assembly files and create or manipulate the basic operations that are applied to quantum circuits. For loading OpenQASM files, a simple recursive descent parser is developed for the language's grammar, as described by the OpenQASM language specification documentation [63]. For manipulating quantum circuits, a class is created which, when instantiated, contains all the qubits used, as well as a list of quantum operations which are applied to the qubits. This list can have operations added to it, removed from it, or be rearranged.

Implementing the Custom Scheduling Algorithm

The hardware scheduling algorithm tested in this research draws from the general algorithmic steps outlined in Guerreschi and Park's [2017] work from their paper entitled "Gate scheduling for quantum algorithms" [13]; however, my work differs in implementation. In their work, the algorithm assumes that other compiler steps, such as quantum error correction, have already been performed, thereby simplifying the problem to looking only at the scheduling of basic quantum operations to the hardware's qubits. In their algorithm, the process of scheduling quantum operations for specific hardware is broken down into four simple steps. Given a list of quantum operations in a circuit, their first step is to construct a dependency graph of the operations. An operation relies on a previous one if that operation uses any of the same hardware resources. For instance, if two operations use the same quantum bits, then the second operation relies on the first. This becomes a little more complicated given that quantum gates are unitary matrices and, as such, can be commutative with other gates. If the two operations are commutative, then there is no dependency between the operations. If two matrices (operations) are commutative, then the order of operations does not matter. If the order of operations does not matter, then the quantum operations do not rely on one another.

Guerreschi and Park's [2017] second step is to assign a priority value to each quantum operation to create a pseudo-ordering of operations [13]. This is performed by making sure that each operation has a larger priority than any operation that depends on it, but a lower priority than any operation it depends on. Guerreschi and Park accomplish this by assigning a priority to each operation that is the maximum of the priorities of all the operation on which it depends, added to its own latency,[27] which in their work is just an integer value of 1 for basic gates and 2 for controlled gates. The ordering of these priorities allows one to schedule each operation with the same priority at the same time. Generally, each operation in a given group operates on different qubits due to the pseudo-ordering of the priorities. However, in algorithms with a high degree of commutativity, a group could contain multiple commuting operations acting on the same hardware resources, making it ambiguous as to which operation is applied first, creating a conflict.

The third step in Guerreschi and Park's [2017] algorithm is to resolve these conflicts created when the ordering of instructions is ambiguous [13]. This is done by splitting the groups of instructions into subgroups containing no conflicts. The method employed by the authors to create conflict-free subgroups is to create a graph whose vertices are logical qubits and whose edges are operations, then apply a form of edge-based graph colouring to assign different colours to groups of operations such that no group can operate on the same qubits. Additionally, they employ subgraph isomorphism as a strategy to minimize the number of groups created. The fourth and final step is to perform routing in which each of the groups, now unambiguous, are applied to the chosen hardware and qubits are swapped such that the operations in question can be performed. In their paper, Guerreschi and Park look only at hardware connected as a linear list of qubits, resulting in a simple routing strategy [13].

[27] Latency is defined as the amount of time required for a particular operation to be completed.

However, any routing strategy could be employed for more complicated hardware configurations. The resulting operations exploit some level of parallelism and are compatible with the underlying hardware connectivity.

My implementation of this scheduling algorithm follows the basic steps outlined by Guerreschi and Park [2017] above [13]; however, it differs in the actual implementation of each step. One of the major differences in this implementation is that the underlying framework in which the scheduling algorithm is implemented on top of was built from the ground up for this research. The basic functionality of this framework includes reading OpenQASM 2.0 assembly files, creating an object for representing the quantum circuit, and manipulating quantum operations on a quantum circuit. These functions are analogous to the scanning, parsing, and optimizing stages of compiler design as discussed in Chapter 2.

The framework in this research utilizes the Cbit and Qubit classes, as classical and quantum bits are the fundamental data types of a quantum algorithm. The Cbit class represents a classical bit and is used for any OpenQASM 2.0 operations that rely on classical values, such as "if statements" or "quantum measurement." The Qubit class is used to represent a quantum bit, which is used in all quantum mechanical operations, such as when applying quantum operators. Qubits and Cbits are able to be members of user defined registers, which act as arrays for storing a fixed number of bits. OpenQASM only allows Qubits and Cbits to be allocated in fixed size registers and so this is why the framework replicates this idea. The internal representation of a quantum circuit is managed by the Circuit class. This class contains lists of all of the quantum bit registers and all the classical bit registers. Additionally, the Circuit class contains an ordered list of all operations to apply to qubits, which are generically called in this thesis as 'events.' These events are ordered the same as they are in an OpenQASM file, reading from top to bottom. In order to model each of these events within the Circuit class, a list is used to store any kind of event that can happen to the quantum circuit. This list is called the ‘GateSchedule,’ as many of the events are applications of quantum gates. Each event is an object, indicating which classical or quantum bits are involved in the operation. Lastly, the Circuit class contains methods to dynamically allocate new classical and quantum bit registers. Code Sample 3.1 shows the definition of this class.

```
public class Circuit {

    private List<Register<Qubit>> quantumRegisters = new List<Register<Qubit>>();
    private List<Register<Cbit>> classicalRegisters = new List<Register<Cbit>>();

    public LinearSchedule GateSchedule {get; set;}

    public Register<Cbit> AllocateCbits(int classicalCount);
    public Register<Qubit> AllocateQubits(int qubitCount);

}
```
Code Sample 3.1: Definition of a quantum circuit within this framework. Circuits contain classical registers, quantum registers, and a list of instructions to apply to the qubits stored within the ‘GateSchedule’ object.

For loading circuits from OpenQASM 2.0 assembly files, a recursive descent parser is developed from the grammar provided through the official OpenQASM 2.0 language specification document [63]. This parser contains recursive rules based on each of the grammar rules within the specification and outputs a syntax tree describing the parsed circuit. This syntax tree is then used to translate each instruction within an OpenQASM file into either the creation of a register for the circuit or the addition of an event to the circuit’s schedule.

In order to be able to convert the OpenQASM instructions to events for the Circuit’s schedule, classes must be made for each OpenQASM instruction, each of which must implement a common interface so that it can be stored within the gate schedule list. The GateEvent class is used to represent the application of OpenQASM's single qubit quantum U gate operation. The ControlledGateEvent class is used to represent OpenQASM's CX (controlled not) operation. The BarrierEvent class is used for the barrier instruction. The IfEvent class is used to represent OpenQASM's conditional quantum operation instruction. The MeasurementEvent class is for OpenQASM's measurement operation and the ResetEvent is for the reset operation.

Additionally, OpenQASM supports custom gate definitions as well. The compiler recursively unrolls these custom gate definitions until they are only composed of the base gates listed.

With this framework, the scheduling algorithm can be built such that it operates on a circuit's linear sequence of instructions, as parsed from OpenQASM algorithms. At the start of the scheduling algorithm, the quantum operations and other events are supplied as a linear sequence in a class that implements C#'s IEnumerable interface in order to be enumerated over. The base interface that all quantum circuit events implement is entitled 'IEvent.' This interface provides methods to determine which qubits and classical bits are used by the event. Code Sample 3.2 shows how this interface is used with C#'s IEnumerable generic interface to create a class for storing the linear sequence of circuit events which will be scheduled by this algorithm.

```
public class LinearSchedule: ISchedule, IEnumerable<IEvent> { … }
```

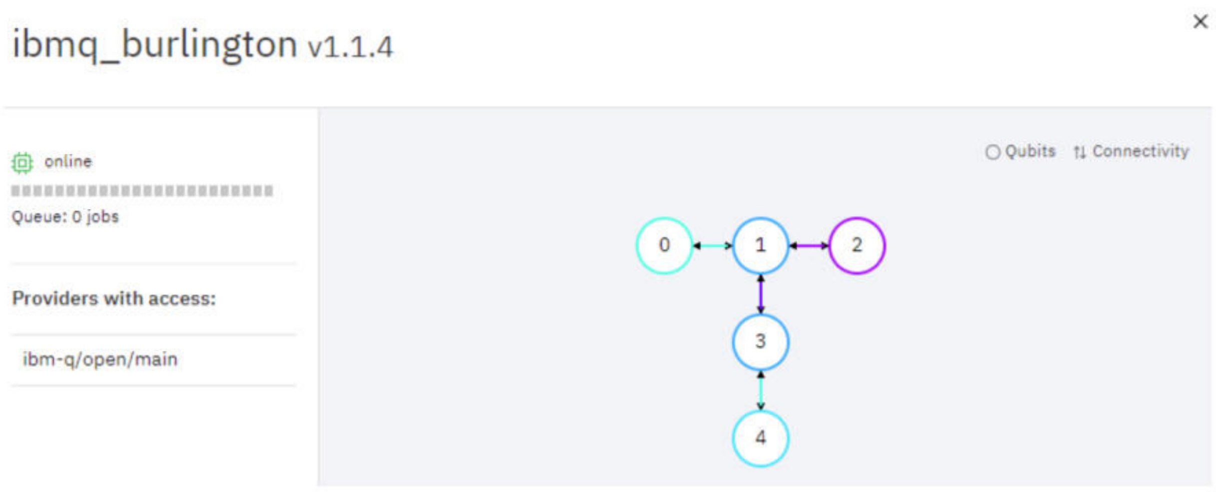
Code Sample 3.2: Definition of a sequence of quantum operations before hardware scheduling.

For this research, a hardware connectivity graph, like the one seen in Figure 3.1, is provided to the scheduling algorithm to describe the physical qubits of the hardware, where the vertices of the graph are the physical qubits and the edges are the adjacencies in which events can be performed between. The hardware connectivity graph is used in the routing step to determine which qubit swaps are valid. Several different hardware connectivity graphs are used across experiments. Graphs representative of real-world quantum computing devices are created from hardware descriptions of IBM devices found on IBM's Quantum Experience[28] portal. For each of the IBM machines, the hardware connectivity graphs are created by copying the configuration shown on the "available devices" list on the homepage of the IBM Quantum Experience platform. These hardware connectivity graphs are loaded from descriptions provided by YAML files where the connectivity property is interpreted as a simplified version of the Graphviz DOT graph description language, as can be seen in Code Sample 3.3 for the IBM Burlington machine. One can see the direct comparison between the hardware connectivity YAML from Code Sample 3.3 and the hardware description image from the Quantum Experience portal in Figure 3.1.

```yaml
Name: Burlington
Alias: ibmq_burlington
Manufacturer: IBM
Technology: Superconducting
Connectivity: >-
    q0 []
    q0 -- q1 []
    q1 -- q2 []
    q1 -- q3 []
    q3 -- q4 []
```

Code Sample 3.3: Hardware description YAML for IBM's Burlington quantum computer. The Connectivity graph is defined here in a domain specific language where qubits are named "q0" – "q4" and connections are indicated by either '--' or '->' for two way or one way connections respectively. The "[]" can be used to describe additional qubit properties but are unnecessary for the scheduler in its current state.

[28] IBM Quantum Experience platform available at https://quantum-computing.ibm.com/


ibmq_burlington v1.1.4

online

Queue: 0 jobs

Providers with access:

ibm-q/open/main

Figure 3.1: The hardware description for IBM's Burlington quantum computer as seen from the IBM Quantum Experience platform. The connectivity graph shown above was used to create the associated hardware description Code Sample 3.3.

Additionally, a bijective[29] map is used to map the logical qubits in the algorithm to the physical qubits available on the hardware. Because the circuit can have fewer logical qubits than the hardware has physical qubits, additional "dummy" logical qubits are made that can be mapped to the remainder of the physical qubits. This allows the remainder of the code to not worry about 'null' values, where a physical qubit is not mapped to any logical qubit, simplifying the code. The implementation for the construction of these "dummy" qubits can be seen in Code Sample 3.4. These "dummy" qubits have no effect on the result of the scheduling algorithm and exist only as placeholders to simplify the implementation of the scheduling algorithm.

[29] A bijective map is a one-to-one mapping between elements in two different sets. The implementation of a bijective map in this research is a custom implementation using two separate instances of C#'s Dictionary class, which is used for one-way mapping.

```
var logicalQubitMap = new BijectiveDictionary<Qubit, PhysicalQubit>(qubitCount);
var dummyQubits = new List<Qubit>();
for (var i = logicalQubits.Count; i < physicalQubits.Count; i++) {
    dummyQubits .Add(new Qubit(null, i));
}
var qubitAssignmentIndex = 0;
foreach (var logical in logicalQubits.Concat(dummyQubits)) {
    logicalQubitMap.Add(logical, physicalQubits[ qubitAssignmentIndex++ ]);
}
```

Code Sample 3.4: Creating a bijective map between logical qubits and physical qubits, adding dummy logical qubits to unused physical qubits in order for all physical qubits to appear within the mapping.

For the first step of the scheduling algorithm, one must construct a dependency graph with all of the circuit events. To construct this graph, each of the events is iterated over and whatever hardware resources they use are recorded. If a prior event uses any of the same hardware resources, then that operation becomes a dependency unless the two events are commutative quantum gates. This is continued for each of the events until the entire dependency graph is filled.

Next, the latencies for each operation are recorded. This is done simply by taking each circuit operation and looking up a latency value from a table. Similar to the work by Guerreschi and Park [2017], the latency values in the table are assigned as integer values with a value of ‘1’ for single qubit gates, ‘2’ for two qubit controlled gates, and ‘3’ for more expensive operations [13]. These values are only used for prioritising the events during scheduling and therefore are different from the time estimations used when estimating how long algorithms are expected to run. For the run time estimations of each event, the latencies are derived from times taken from IBM quantum computers accessible on the qiskit

GitHub.[30] The exact latencies used for each of the possible quantum circuit operations (as defined by the OpenQASM language specification) are recorded in Table 3.2.

Circuit Event (Operation)	Scheduling Latency	Run Time Estimate
Barrier	1	0
Measurement	3	1ms
Reset	3	1ms and 150ns
Classical If	2	1ms and 150ns
Single Qubit Gate	1	150ns
Controlled 2 Qubit Gate	2	211ns

Table 3.2: Latency times for various quantum circuit events. These times are semi-arbitrary. Measurements, on average, take drastically more time than gate applications and that is indicated in the latency times. Resets are usually a measurement potentially followed by the application of an ‘x’ gate. “If” statements require classical control and so are given the same time as a reset.

With the finalized dependency graph and all event latencies computed, the priority values can be calculated. The dependency graph can be traversed depth first, with priorities calculated in a post-order fashion. However, experimental trials and resulting errors indicated that it is not an optimal method in the case of several algorithms used for this research. Many of the dependency graphs contain diamond patterns which, when traversed, naïvely result in huge growths in the run time of the scheduling algorithm, as work that has already been completed gets recalculated many times. Additionally, with larger graphs, languages with a limited stack size such as C# may crash as the stack overflows. A better method in these cases is to pre-compute a linear ordering and simply iterate over all graph nodes in that order. These linear orderings can be pre-computed during earlier analysis passes and, as such, do not require much additional run time when used. By iterating over graph nodes instead of using a post-ordered recursive traversal method, the stack depth is minimized and the chance

[30] Yorktown Qiskit GitHub accessed March 13, 2022 from https://github.com/Qiskit/ibmq-device-information/blob/master/backends/yorktown/V1/version_log.md

of a stack overflow decreases. A simple choice of linear ordering is to reverse the ordering of the original instructions as defined in the source code file. Alternatively, as done in this project, a depth value is computed when constructing the dependency graph such that the depth of a graph node is equal to the maximum of all the nodes it relies on plus 1; then the order is in decreasing depth. This depth-based ordering is very similar to reversing the ordering of instructions from the source file. By iterating over a list of graph nodes rather than recursively post-order traversing the whole graph, larger graphs can be supported without overflowing C#'s stack. This speeds up the computation of the priorities without adding additional complexity. Regardless of the method used to compute priorities, the priority calculation should be set up in such a way that it creates pseudo-ordering whereby all events must have a lower priority than those events they depend on and a higher priority than those that depend on it. The exact computation used to compute the priority of each event node in the dependency graph is given by Equation 3.1

(3.1)

$$P_E = L_E + Max(P_{d1} \dots P_{dN}) \text{ where}$$
$$P_E = \text{ priority of the current event}$$
$$L_E = \text{latency of current event}$$
$$P_{d1} \dots P_{dN} = \text{ priorities of all events that depend on the current event}$$

Equation 3.1: Priority computation for circuit events. This can be efficiently computed through the dependency graph in which the latency of each event is recorded in the graph vertices and the edges can be used to find all the events that depend on it for ease of computing the maximum.

Once priorities have been determined, the events can be grouped by their priority values in descending order such that those groups of events with the highest priority are scheduled first.

The next step in the scheduling algorithm is to break apart groups where instruction scheduling is ambiguous. To determine whether there are ambiguous events within a group, each event is checked to ensure it does not use any of the same hardware resources (qubits, classical bits, or otherwise) as any of the other operations in the group. If there are no ambiguous conflicting instructions, the group can be scheduled immediately. If there are conflicts, the group must be split into multiple unambiguous groups. For this task, an interaction graph is defined such that each vertex is a logical qubit and each edge represents an event that uses those qubits. If an edge is a "self loop," meaning that the edge starts and ends at the same vertex, then it is a single qubit operation that only depends on that qubit. If the edge spans two different qubits, it is a two qubit operation. This is the same kind of data structure as employed by Guerreschi and Park [2017, 13]. However, the ambiguity resolution in this research's hardware scheduling algorithm differs from their algorithm in the usage of the interaction graph data structure for ambiguity resolution.

In Guerreschi and Park's [2017] research, ambiguity resolution uses graph colouring along with subgraph isomorphism to allow for the discovery of better unambiguous groupings [13]. In my research, a more naïve approach is taken where colours are assigned to each edge by iterating over each edge such that no other edges connecting to the same qubit vertices share the same colour. This assignment is biased such that operations are more likely to be grouped into the first few groups rather than creating new groups. This is performed by trying to start with the first colour on each edge rather than simply generating a new colour for each edge. This strategy is quick and effective, though not necessarily optimal in terms of minimizing the number of colour groups. Code Sample 3.5 below shows the code used to implement this naive form of edge colouring. The non-conflicted groupings created are passed back to be scheduled along with all of the unambitious groups. The decision to use this more naïve method rather than Guerreschi and Park’s [2017, 13] more intelligent strategy is due to the added computational complexity of using their particular strategy.

```
foreach (var edge in this.Edges) {
    // Check if edge is already coloured
    if (edge.Data.HasColour()) {
        continue;
    }

    // Get all edges coming off this edge's endpoints
    var startEdges = this.IncidentEdges(edge.Startpoint);
    var endEdges = this.IncidentEdges(edge.Endpoint);

    // Always bias towards 1 (force more things to be 1 than not 1)
    var colour = 1;

    // Select a list of already used colours
    var coloursItCantBe = startEdges.Select(e => e.Data.Colour).Concat(endEdges.S
elect(e => e.Data.Colour)).ToList();

    // Pick a colour that is not already used
    while (coloursItCantBe.Contains(colour)) {
        colour++;
    }

    // Assign colour
    edge.Data.Colour = colour;
}
```

Code Sample 3.5: Naïve edge colouring algorithm with an attempt to bias towards using the first colours as much as possible.

To schedule each of the unambiguous groupings of events, only one other operation is required — routing. Routing is the act of inserting swap gates such that qubits can be moved around the hardware to facilitate performing each event. This is required due to the fact that two-qubit operations can only be performed on neighbouring qubits and current hardware has limited connectivity between neighbours [8]. As noted previously, swaps can be an expensive quantum operation; as such, the goal of any routing algorithm is to minimize the number of added swaps. Reducing the total number of swaps globally, across the entire quantum program rather than for particular pieces, is an open and difficult problem to solve. For Guerreschi and Park's [2017] implementation, the researchers only examined hardware with qubits arranged in a linear array [13]. For this linear hardware configuration, routing between qubits is simple as qubits can only be swapped left or right along the linear array. My research examines hardware scheduling for realistic quantum hardware, of which there are many possible qubit configurations. Therefore, unlike Guerreschi and Park's research, where routing is a strictly linear process, in order to allow for more realistic and complicated hardware configurations, the A* search algorithm is used for routing.

The A* algorithm is a graph traversal and search algorithm which can be viewed as an improvement or extension of Dijkstra's shortest path algorithm [64]. This algorithm is often used to find the shortest path from one node in a graph to another node. What constitutes a node varies depending on the purpose of the graph. What makes the A* algorithm an improvement on Dijkstra's is an additional heuristic function which is used to apply additional weighting to each path. Specifically, the heuristic is used in addition to the currently explored path lengths in order to sort which paths it should continue exploring. The goal of the added heuristic is to attempt to traverse the most promising paths first, resulting in a better average run time than other search algorithms. This combined weighting is often called the 'F' score for a particular node [64]. Code Sample 3.6 shows pseudocode describing a simplistic version of the A* search algorithm.

```
def AStar (start, goal)
    openSet = {start}

    while openSet is not empty
        current = select node from openSet with min(f-score)
        if current = goal
            return backtrack_path(current)

        openSet.remove(current)
        for neighbour of current
            g-score = current.g-score + distance(current, neighbour)
            if g-score < neighbour.g-score
                neighbour.previous = current
                neighbour.g-score = g-score
                neighbour.f-score = g-score + heuristic(neighbour)
                if neighbour not in openSet
                    openSet.add(neighbour)

    return failure # no path found
```

Code Sample 3.6: Pseudocode for simplistic implementation of the A* Search algorithm.

The A* search algorithm, in this case, operates on a qubit connectivity map. Each possible mapping of logical qubits to physical qubits is a node of the graph. The edges of the graph are represented by swapping particular logical to physical qubits, as restricted by the connectivity map. The initial state of the graph is represented by the current mapping between logical and physical qubits at the start of the routing stage. This search finds the shortest sequence of qubit swaps from the current mapping of logical to physical qubits to a new mapping in which all the operations being scheduled can be performed on the hardware. For this research, a naïve heuristic is used where the qubits associated with each instruction are assigned a colour, and for each qubit that is not connected to others of the same colour, then the heuristic value increases by 1. Since the A* search algorithm is guaranteed to return the shortest path, this in turn guarantees the shortest number of added swaps for the given configuration.

Even though the A* search algorithm is guaranteed to return the shortest path after searching, this does not mean that the number of swaps is at a global minimum; rather, it means that the number of swaps for each individual step is at a local minimum. In order to globally minimize the total number of qubit swaps, a lookahead[31] strategy must be employed. This research, like Guerreschi and Park's research [2017, 13], leaves this as a problem to be solved in future research.

The setup of this search is as follows. Using the logical to physical qubit mapping and the hardware connectivity graph, each of the events for the group is marked onto the physical qubit graph. The A* search algorithm is used to search the state-space moving from one configuration of qubits to another by applying qubit swapping until there is no longer any physical gap between any of the qubits involved in all of the operations. These swaps are then added to the schedule and the logical to physical qubit map is updated with the new physical positions for each logical qubit. These steps are repeated for each of the conflicting groups of circuit events.

The scheduling algorithm returns a schedule which contains groups of operations that are capable of being performed at the same time on realistic near-future quantum computing hardware, with additional steps taken to insert swapping, allowing circuits that are not designed for a particular hardware to work. In summary, this algorithm follows the broad steps outlined by Guerreschi and Park [2017, 13], with my own interpretation of all steps, particularly with respect to ambiguity resolution, which uses a more naïve strategy, as well as the routing algorithm, which is less hardware limiting due to use of the A* search algorithm.

[31] Lookahead is a process in which subsequent passes of an algorithm are examined in order to make better decisions on the current pass.

Experimentation

Experimentation for this research involves running a program that performs hardware scheduling for selected quantum programs or circuits on a variety of selected hardware configurations, using a custom hardware scheduling algorithm. A variety of statistics are recorded at each step of the process, stored in different tables for later analysis. Details of the overall experimental process are presented in Code Sample 3.7 in pseudo-code.

```
for each circuit
    record circuit statistics

    for each hardware
        start timer
        schedule circuit for hardware
        read timer

        record scheduling statistics
```

Code Sample 3.7: Pseudo-code detailing the experimental procedure.

The hardware devices used in the experiments are freely accessible devices accessed through IBM's Quantum Experience, of which there are nine different hardware devices with five different qubit arrangements. These include: the Armonk machine (1 qubit); the Burlington, Essex, London, Ourense, and Vigo (five qubit T arrangement) machines; and the Yorktown (5-Qubit diamond arrangement), Rome (five Qubit linear arrangement), and 32 Qubit fully connected IBM High Performance Simulator. Each of these hardware devices is listed in Table 3.3.

Hardware Name	Qubits	Connectivity Configuration
Armonk	1	N/A
Burlington	5	T
Essex	5	T
Ourense	5	T
Vigo	5	T
Yorktown	5	Diamond
Rome	5	Linear
IBM Simulator	32	Fully

Table 3.3: List of all hardware configurations used with their number of qubits and connectivity.

For each of the quantum hardwares chosen, 39 different quantum algorithms are tested, 21 of which are provided through QisKit's OpenQASM repository.[32] Some of these 21 algorithms have already been optimized for the Yorktown arrangement (otherwise known as ibmqx2), and as such should require minimal, if any, changes after hardware scheduling. The remaining 17 tested quantum algorithms were created by generating the circuit for a given set of input parameters. These circuit generators were developed by studying the algorithm definitions as identified in the June 2020 version of the online QisKit textbook.[33] While most of these circuit generators were built from scratch, parts of them were translated from their QisKit Python originals, such as the majority of Shor's algorithm.[34] All algorithms being tested, both generated and those provided by Qiskit, are listed in Table 3.4.

ID	Algorithm Name	Arguments	Source	Purpose
1 (1.0)	Quantum Teleportation		Generator derived from the QisKit textbook	Teleport the state of one qubit to another entangled qubit
2 (1.1)	Super dense Coding	00	Generator derived from the QisKit textbook	Send 2 bits of classical data to another party with 1 qubit
3 (1.2)	Super dense Coding	01	Generator derived from the QisKit textbook	Send 2 bits of classical data to another party with 1 qubit
4 (1.3)	Super dense Coding	10	Generator derived from the QisKit textbook	Send 2 bits of classical data to another party with 1 qubit
5	Super dense	11	Generator derived from the	Send 2 bits of classical data to

[32] QisKit repository found at https://github.com/Qiskit/openqasm/tree/master/examples
[33] QisKit textbook found at https://qiskit.org/textbook/ch-algorithms
[34] See https://qiskit.org/documentation/locale/de_DE/_modules/qiskit/aqua/circuits/gates/multi_control_u1_gate.html

ID	Algorithm Name	Arguments	Source	Purpose
(1.4)	Coding		QisKit textbook	another party with 1 qubit
6 (1.5)	Deutsch Algorithm	constant true	Generator derived from the QisKit textbook	Determine if a function f {0,1} → {0,1} is constant
7 (1.6)	Deutsch Algorithm	constant false	Generator derived from the QisKit textbook	Determine if a function f {0,1} → {0,1} is constant
8 (1.7)	Deutsch Algorithm	Identity	Generator derived from the QisKit textbook	Determine if a function f {0,1} → {0,1} is constant
9 (1.8)	Deutsch Algorithm	Flip	Generator derived from the QisKit textbook	Determine if a function f {0,1} → {0,1} is constant
10 (1.9)	Deutsch Josza	3 qubits , balanced oracle	Generator derived from the QisKit textbook	Determine if a function f $\{0,1\}^n$ → {0,1} is constant
11 (1.10)	Bernstein Vazrani	3 qubits, value 11	Generator derived from the QisKit textbook	Determine if the input falls into one of two classes based on a secret string.
12 (1.11)	Simon's Algorithm	11	Generator derived from the QisKit textbook	Detect the period of a binary function
13 (1.12)	Quantum Fourier Transform	3 qubits	Generator derived from the QisKit textbook	Quantum version of the discrete Fourier Transform for 3 qubits
14 (1.13)	Quantum Fourier Transform	4 qubits	Generator derived from the QisKit textbook	Quantum version of the discrete Fourier Transform for 4 qubits
15 (1.14)	Quantum Fourier Transform	5 qubits	Generator derived from the QisKit textbook	Quantum version of the discrete Fourier Transform for 5 qubits
16 (1.15)	Grover's Search	9 items, phase oracle search	Generator derived from the QisKit textbook	Unstructured search of a collection of items using a black box as a filter
17 (1.16)	Max Cut Problem	Triangle plus edge graph	Generator derived from the QisKit textbook	Partition a graph's vertices into two sets with a maximum cut
18 (1.17)	011_3_qubit_ grover_50_		https://github.com/Qiskit/openqasm/tree/master/examples	3 Qubit Grover amplification repeated twice
19 (1.18)	Adder		https://github.com/Qiskit/openqasm/tree/master/examples	4 bit quantum ripple carry adder
20 (1.19)	bernstein-vazirani		Quantum algorithm implementations for beginners, Coles et al, 2018	Determine if the input falls into one of two classes based on a secret string.
21 (1.20)	Bigadder		https://github.com/Qiskit/openqasm/tree/master/examples	8 bit ripple carry adder made from 2 4 bit adders
22 (1.21)	Deutsch_Algorithm		https://github.com/Qiskit/openqasm/tree/master/examples	Determine if a function f {0,1} → {0,1} is constant
23 (1.22)	inverseqft1		https://github.com/Qiskit/openqasm/tree/master/examples	Quantum Fourier Transform and measure
24 (1.23)	inverseqft2		https://github.com/Qiskit/openqasm/tree/master/examples	Quantum Fourier Transform and measure second version
25 (1.24)	ipea_3_pi_8		https://github.com/Qiskit/openqasm/tree/master/examples	Iterative phase estimation
26 (1.25)	Iswap		https://github.com/Qiskit/openqasm/tree/master/examples	Swap 2 qubit states and phases by amplitude 'i'

ID	Algorithm Name	Arguments	Source	Purpose
27 (1.26)	pea_3_pi_8		https://github.com/Qiskit/openqasm/tree/master/examples	Phase estimation
28 (1.27)	Qec		https://github.com/Qiskit/openqasm/tree/master/examples	Repetition code syndrome measurement
29 (1.28)	qe_qft_3		https://github.com/Qiskit/openqasm/tree/master/examples	Quantum version of the discrete Fourier Transform for 3 qubits
30 (1.29)	qe_qft_4		https://github.com/Qiskit/openqasm/tree/master/examples	Quantum version of the discrete Fourier Transform for 4 qubits
31 (1.30)	qe_qft_5		https://github.com/Qiskit/openqasm/tree/master/examples	Quantum version of the discrete Fourier Transform for 5 qubits
32 (1.31)	Qft		https://github.com/Qiskit/openqasm/tree/master/examples	Quantum version of the discrete Fourier Transform for 4 qubits
33 (1.32)	Qpt		https://github.com/Qiskit/openqasm/tree/master/examples	Quantum phase transition algorithm
34 (1.33)	Rb		https://github.com/Qiskit/openqasm/tree/master/examples	Randomized benchmarking sequence
35 (1.34)	Teleport		https://github.com/Qiskit/openqasm/tree/master/examples	Teleport the state of one qubit to another entangled qubit
36 (1.35)	teleportv2		https://github.com/Qiskit/openqasm/tree/master/examples	Improved teleportation circuit
37 (1.36)	W-state		https://github.com/Qiskit/openqasm/tree/master/examples	Prepare qubits in the W state
38 (1.37)	W3test		https://github.com/Qiskit/openqasm/tree/master/examples	Test of preparing qubits in the W state
39 (1.38)	Shor's Algorithm	15	Generator derived from Qiskit Python implementation	Find a factor of the given integer

Table 3.4: List of all quantum algorithms implemented for this experiment. ID numbers were provided for each algorithm used in the experiments; these numbers are used to index the algorithms from a list in the experiment program. The source for each algorithm is provided and if the algorithm was generated, the parameters used for generating that particular algorithm's circuit are also shown.

Data collection

The data collected in these experiments are metrics about quantum algorithms both before and after scheduling, stored in Microsoft Excel compatible CSV (comma separated value) files, where each row represents a circuit and each column represents a specific hardware implementation, with the cell values being that particular metric for that circuit when being optimized for that hardware. The metrics recorded are the number of qubits used in each algorithm, the number of instructions/steps before and after scheduling, the time taken to perform the scheduling optimization, an estimation of how long each algorithm takes to execute, and the additional number of swaps required to make the circuit compatible with the given hardware arrangement. Qubit and instruction counts are easily determined by manual analysis of each algorithm's OpenQASM assembly code. Qubits are declared at the top of OpenQASM files and can be summed to get the total number of used qubits. Similarly, the number of instructions in the assembly file can be counted before optimization, and the number of parallel steps can be counted in the scheduled circuit. The time taken to perform the scheduling is captured by the use of C#'s Stopwatch class, which is started before the scheduling and then stopped after scheduling. The scheduling time is measured for each of the algorithms being tested against each of the different hardware configurations. The run-time estimations are based on assigning estimated run time values for each type of OpenQASM instruction per Table 3.2. Before scheduling, the run time is determined by iterating through all the OpenQASM instructions and summing up the operation's estimated run time. After scheduling, the estimation is undertaken by summing up the maximum operation estimation from each of the parallel operations within each of the scheduled steps. Once all trials have run and the metrics are gathered for each, the averages and standard deviations are computed for each metric across hardware and algorithm so they can be more easily analysed.

Program Validation

Before running the experiments, the experiment program was subjected to several tests to ensure the experiments were working. The first test simply attempted to schedule each algorithm against a single hardware. It was during this test that Shor's algorithm was noted to be causing a stack overflow within the .NET virtual machine. This indicated an issue with the scheduling algorithm created by a recursion issue, resulting in traversing over graph nodes multiple times, which created too much stack depth. This was subsequently fixed and the test redone. Subsequent tests found additional minor issues with the implementation of pathfinding and graph colouring algorithms used by the hardware scheduler. When testing was completed, the experiment was run and repeated many times to collect enough data. The procedure used for analyzing the algorithms is described in the Data Analysis section.

Performance

Trials of the experiment took only a few seconds to complete. This meant that large numbers of trials could be run over a very short period of time. As a result, 100 different trials were run to accumulate enough information. With this much data, analysing the results can become a time-consuming task. As a result, an auxiliary program was created which could sift through the data and compute averages, sums, as well as standard deviations, creating summaries which are easier to analyse.

3.2 Data Analysis

Each experiment went through the 39 quantum algorithms and scheduled them for all 10 types of hardware. Each experiment generated nine main files, each containing the data for a single metric recorded against each algorithm and each of the compatible hardware. The first of these tables is simply entitled "index.csv" and is used to record the algorithms tested and the running time of the experiment. The second table, "matrix.qubitCountBefore.csv," records how many qubits are involved in a particular quantum circuit before hardware scheduling is performed. The third table, "matrix.estimatedRuntimeBefore.csv," contains the estimated run time of the given algorithm before it has been optimized. This estimate is the sum of individual run-time estimates for each of the instructions within the OpenQASM file. For single qubit and controlled qubit operations, the instruction run-time estimations are based on performance metrics from IBM quantum computers. Other OpenQASM instructions, such as those for barriers, resets, and measurements, are approximations, as no official metrics could be found for them on the official IBM documentation repository. These estimations are identical to those shown in the "Procedure" section in Table 3.2. The fourth table, "matrix.estimatedRuntimeAfter.csv," contains the estimated run time post-scheduling for each algorithm on each hardware configuration. Since scheduling allows for several operations to be run synchronously, only the longest time of each synchronous group is used for the sum. The "matrix.estimatedRuntimeDelta.csv" table contains the difference between the estimated run times after scheduling and before scheduling. The "matrix.eventCount.csv" table is a count of the total number of OpenQASM instructions before scheduling. The "matrix.eventCountDelta.csv" table shows the difference between the original number of instructions and the number of synchronous groups of instructions in the scheduled version. The "matrix.eventSwapCountAfter.csv" table shows the number of swap events added to the circuit to meet hardware connectivity constraints. Finally, the "matrix.optimizationTime.csv" table shows how long it took to schedule each of the algorithms for each hardware device. This metric is measured by using C#'s stopwatch object, which tracks how long it takes to execute desired parts of a program. Additionally, for verification purposes only, the data structures used for the hardware scheduling algorithm, such as the logical data precedence graph and the physical data precedence table, are also saved to files.

Using the auxiliary program mentioned earlier, each metric, across all trials, is compiled into a single Excel spreadsheet based on the average of the related data. For each spreadsheet, the algorithms are sorted by a "complexity factor" based on how complex the quantum algorithm is. The "complexity factor" is computed as the product of the number of instructions and the number of qubits assigned to each algorithm, which is a simple metric that may not capture all of the complexities of a quantum algorithm accurately. Once sorted, the data can then be plotted in order to see trends for each metric as the quantum algorithms increase in complexity.

3.3 Rigour, Reliability, and Validity

To ensure that this research is rigorous, reliable, and valid, each of these properties is addressed through some part of the methodology employed in this research. Cypress [2017] defines rigour as "the quality or state of being exact, careful, or with strict precision" [65, p. 254]. To address this particular quality, the experimental procedures explained in the earlier sections required that the scheduling algorithm be run multiple times in order to ensure that as many potential sources of error were eliminated as possible.

Reliability refers to the consistency of a given measure and, in research, refers to three different types of consistency. Test-retest reliability refers to how consistent a measure is over time, internal consistency refers to the consistency across different items, and inter-rater reliability refers to how consistent measures are among different researchers [66]. To address test-retest reliability, the experiment is repeated multiple times and the recorded values are checked to see that they are similar. This validation, therefore, ensures that the experiment has been checked for its ability to be repeated. The re-tests occur at a variety of different times and days in order to make sure that timing of the test is not influencing the results. To address internal consistency, the scheduling algorithm is tested against many different quantum computing algorithms to verify that the strategy is reliable for use across a variety of circumstances. To address inter-rater reliability, each experimental trial is well documented. By using the same steps and documenting every trial, the experiment itself will also be reliable in that it will be reproducible by anyone with the same equipment. Additionally, in order to be transparent, after this thesis becomes publicly available, the source code for the entirety of this research will be available on my personal GitHub[35] or Zenodo (DOI: 10.5281/zenodo.6944561).

Validity is the extent to which scores from a measure represent the variable they are intended to measure [66]. Validity is usually categorised as internal or external validity. Internal validity refers to the degree to which the results are attributed to the associated variable rather than some other explanation. To address this type of validity, experiments are executed multiple times across a wide range of hardware and times of the day, thereby allowing several variables to be eliminated as potential reasons for the obtained results. External validity is the degree to which the results of the research can be generalized or applied outside of the research context. Since this research is meant to assess hardware-independent optimization strategies, it should, by nature, be generalizable. Each of the optimization strategies is logically checked such that if a strategy should, in theory, improve performance in common use cases, then experimentally this should be observed in the results of the experiments.

[35] Colin Halseth's personal GitHub found at https://github.com/qkmaxware/DotQASM

Chapter 4: Key Findings & Analysis

4.1 Introduction

The purpose of this chapter is to present the results of the experimental work outlined in Chapter 3: Methodology. The chapter begins with a section which discusses the compatibility of each of the tested quantum algorithms against each of the hardware configurations. This is followed by a discussion of each of the subsequent experimentally gathered metrics. Sections are provided for "changes in instruction stages," "added number of swap gates," "estimated algorithm run time change," and "scheduling algorithm run time." Each metric is introduced, followed by the results for that metric. The results are displayed in tabular form, with the metric displayed for each algorithm and for all five of the different hardware configurations. Additionally, the metric is shown visually in a chart where the algorithms are ordered by their complexity factor to observe how the metric changes as algorithms become more complex. The charts are represented with points for each discrete measured value, connected by lines for better visualization of trends. For all charts, the algorithms are spaced apart uniformly rather than being proportionally spaced by their computed complexity. Lastly, each section includes a description summarizing the trends that can be gleaned from both the tables and the charts. The final section shows how the metrics are used to determine the effectiveness of the hardware scheduling algorithm implemented in this research.

4.2 Algorithm Hardware Compatibility

Before discussing results for each metric, it is important to document which algorithms are able to be run to completion on which hardware configurations, as not all algorithms are able to run to completion on all hardware configurations due to some algorithms requiring more qubits than some hardware can support. There are five different types of hardware configurations: "single qubit," which is a hardware composed of only one qubit; "5 Linear," which is a linear array of five qubits in which each qubit is connected to its neighbors in the array; "5 T," which is an arrangement of five qubits in the shape of a 'T' with 3 qubits across the top and 3 qubits along the vertical; "5 Diamond," which is a five qubit arrangement in the shape of a diamond, with one qubit at each corner and one in the centre; and "32 Full Connectivity," which is an arrangement of 32 qubits in which all qubits are connected to all other qubits. As can be seen from Table 4.1 below, only one algorithm (1.32, the quantum phase transition algorithm) could be run on the Single Qubit configuration as the algorithm itself only uses a single qubit. As a result, values for all other algorithms remain empty for the single qubit configuration. Algorithm 1.18, the 4-bit quantum ripple carry adder, is only able to be run on the 32-qubit full connectivity hardware, due to requiring 8 qubits – 4 for each added number.

Circuit Id	**Single Qubit**	**5 Linear**	**5 T**	**5 Diamond**	**32 Full Connectivity**
1.0	No	Yes	Yes	Yes	Yes
1.1	No	Yes	Yes	Yes	Yes
1.2	No	Yes	Yes	Yes	Yes
1.3	No	Yes	Yes	Yes	Yes
1.4	No	Yes	Yes	Yes	Yes
1.5	No	Yes	Yes	Yes	Yes
1.6	No	Yes	Yes	Yes	Yes
1.7	No	Yes	Yes	Yes	Yes
1.8	No	Yes	Yes	Yes	Yes
1.9	No	Yes	Yes	Yes	Yes
1.10	No	Yes	Yes	Yes	Yes
1.11	No	Yes	Yes	Yes	Yes
1.12	No	Yes	Yes	Yes	Yes
1.13	No	Yes	Yes	Yes	Yes
1.14	No	Yes	Yes	Yes	Yes
1.15	No	Yes	Yes	Yes	Yes

Circuit Id	Single Qubit	5 Linear	5 T	5 Diamond	32 Full Connectivity
1.16	No	Yes	Yes	Yes	Yes
1.17	No	Yes	Yes	Yes	Yes
1.18	No	No	No	No	Yes
1.19	No	Yes	Yes	Yes	Yes
1.20	No	No	No	No	Yes
1.21	No	Yes	Yes	Yes	Yes
1.22	No	Yes	Yes	Yes	Yes
1.23	No	Yes	Yes	Yes	Yes
1.24	No	Yes	Yes	Yes	Yes
1.25	No	Yes	Yes	Yes	Yes
1.26	No	Yes	Yes	Yes	Yes
1.27	No	Yes	Yes	Yes	Yes
1.28	No	Yes	Yes	Yes	Yes
1.29	No	Yes	Yes	Yes	Yes
1.30	No	Yes	Yes	Yes	Yes
1.31	No	Yes	Yes	Yes	Yes
1.32	Yes	Yes	Yes	Yes	Yes
1.33	No	Yes	Yes	Yes	Yes
1.34	No	Yes	Yes	Yes	Yes
1.35	No	Yes	Yes	Yes	Yes
1.36	No	Yes	Yes	Yes	Yes
1.37	No	Yes	Yes	Yes	Yes
1.38	No	No	No	No	Yes

Table 4.1: Ability of algorithm to run on specified hardware configuration

4.3 Change in Instruction Stages

The experimental work captures metrics about the total number of instructions in the original algorithms before they are scheduled, as well as the number of scheduled steps in the optimized version as determined by the output schedule. Before scheduling, the number of steps in each algorithm is equal to the number of OpenQASM instructions in the algorithm, as each instruction is run sequentially one after the last. After hardware scheduling, the number of steps is the number of columns in the output schedule where each column indicates groups of instructions that can be run at the same time. The difference between running the algorithm sequentially on an idealized quantum computer with full connectivity and running the version optimized for the given hardware can then be obtained. Ideally, the lower this number, the better the result. A negative number indicates that several instructions can be performed at the same time, thereby reducing how long it takes the algorithm to run to completion.

Circuit Id	Single Qubit	5 Linear	5 T	5 Diamond	32 Full Connectivity
1.0		1	1	0	0
1.1		-1	-1	-1	-1
1.2		-1	-1	-1	-1
1.3		-1	-1	-1	-1
1.4		-1	-1	-1	-1
1.5		-1	-1	-1	-1
1.6		0	0	0	0
1.7		-1	-1	-1	-1
1.8		0	0	0	0
1.9		-2	-5	-5	-8
1.10		-6	-6	-8	-8
1.11		-1	0	-3	-5
1.12		-3	-3	-5	-5
1.13		-7	-7	-10	-13
1.14		-18	-17	-21	-26
1.15		-13	-13	-20	-20
1.16		-9	-8	-13	-13
1.17		-17	-17	-40	-40
1.18					-31
1.19		-6	-6	-8	-8
1.20					-92
1.21		-1	-1	-1	-1
1.22		0	0	0	0
1.23		0	0	0	0
1.24		-4	-4	-4	-4
1.25		-3	-3	-3	-3
1.26		-6	-8	-11	-15
1.27		9	6	3	0
1.28		1	1	-1	-1
1.29		-21	-24	-27	-33

Circuit Id	Single Qubit	5 Linear	5 T	5 Diamond	32 Full Connectivity
1.30		-48	-44	-55	-55
1.31		-9	-8	-11	-14
1.32	0	0	0	0	0
1.33		-1	-1	-1	-1
1.34		0	0	0	0
1.35		-1	-1	-1	-1
1.36		-5	-5	-8	-8
1.37		-2	-2	-4	-4
1.38					-8357

Table 4.2: Decrease in number of schedule stages as a result of hardware scheduling. A lower number means fewer scheduled steps in the optimized algorithm compared to the original algorithm (run sequentially).

Table 4.2 above shows the differences between the number of instructions in the unscheduled algorithm and the number of steps in the scheduled version. As can be seen, the single qubit hardware configuration column is mostly empty since most of the quantum algorithms tested in this research require more than a single qubit. The algorithms that most benefitted from hardware scheduling appear to be Experiment 1.38, Shor's algorithm, and Experiment 1.30, which is a five qubit Quantum Fourier Transform (QFT) (qe_qft_5), with other QFT variants such as 1.29 and 1.31 closely following. Other algorithms, such as 1.17 (011_3_qubit_grover_50_), show considerable changes to the instruction change count, especially for the diamond and full connectivity hardware. There are also several algorithms which show no changes across each of the hardware configurations, including Algorithm 1.6 (constant false Deutsch algorithm), 1.8 (flip function Deutsch Algorithm), 1.22 (inverseqft1), 1.32 (QPT), and 1.34 (teleport). In the case of the Deutsch Algorithm variations, the lack of change most likely reflects the fact that these algorithms are already very small and therefore do not allow for any parallel task scheduling. The QPT, inverseqft1, and teleport algorithms suffer from a lack of parallelizable instructions. In some cases, this is the result of barrier instructions which explicitly limit parallelization; in other cases, it is the result of a large number of subsequent measurements which also serve to limit parallelization.

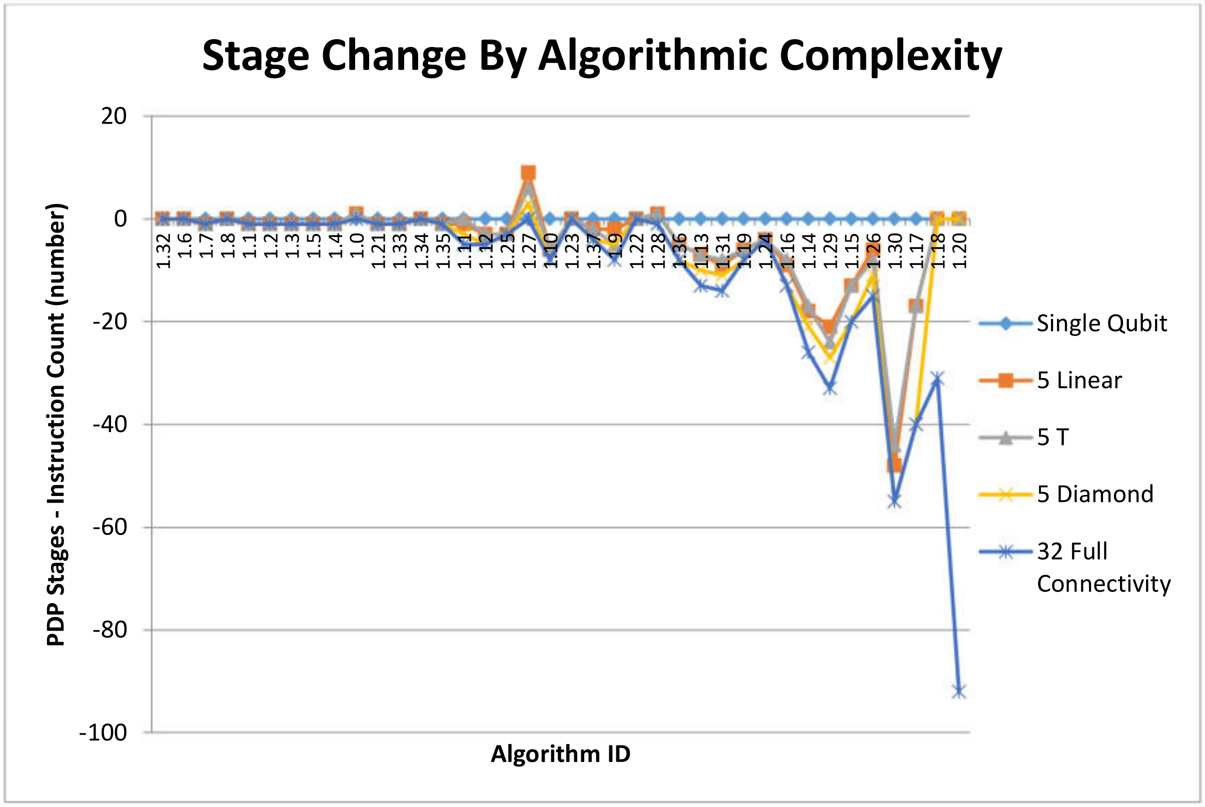


Figure 4.1: Decrease in number of schedule stages as a result of hardware scheduling plotted in order of each algorithm's algorithmic complexity.

The change in algorithm steps is then plotted in order of each algorithm's complexity value, the results of which can be seen in Figure 4.1, with the exception of Algorithm 1.38 (Shor's Algorithm). This is because its change is so large compared to the other algorithms that including it in the figure would change the scale such that all other algorithms would no longer be visibly discernable from each other. The algorithms are indicated by their ID number and sorted from left to right, along the x-axis of the chart, by their algorithmic complexity value (see Chapter 3). Each hardware configuration is represented as a different coloured line. When shown in this way, it can be seen that there is a general trend across all hardware configurations that the more complex an algorithm is, the more it benefits from hardware scheduling. There is a general pattern of decreasing instruction stages in the scheduled algorithms and increasing complexity as one moves from left to right. Despite this general trend, the lines are not smooth, resulting in many peaks and dips. The peak on algorithm 1.27 is most likely the result of the small size of the algorithm, combined with controlled operations, resets, and measurements, leaving little room for any parallelism to save on algorithmic run time; as a result, the only changes in instructions will be additions due to routing. The peaks and dips of other algorithms are most likely the result of sorting by the algorithmic complexity factor, which may not completely capture the true complexity of the algorithm as it does not include any information about the relationship between instructions and their dependencies, which does play a role in how effective hardware scheduling can be.

4.4 Added SWAP gates

Another metric obtained from the experiments is the number of SWAP gates that were added to schedules in order to conform to the underlying hardware connectivity. The routing step of the hardware scheduler created in this research adds additional SWAP gates if qubit states need to be exchanged in order to satisfy hardware connectivity constraints. From the literature review, it was noted that SWAP gates are one of the most time-consuming quantum gates, so minimizing the number of these is important for hardware scheduling algorithms [38]. Table 4.3 below shows the number of SWAP gates that this scheduler adds to each algorithm to make them compatible with each of the five different hardware configurations. Like Table 4.2, empty cells in Table 4.3 are the result of the quantum algorithm not being able to be scheduled on the specific hardware. This occurs mainly for the Single Qubit hardware which can only run one algorithm, as all the other algorithms require more qubits. However, these empty cells also occur for Algorithm 1.18 (adder) and 1.20 (big adder, each of which could only be run on one of the hardware configurations.

Circuit ID	Single Qubit	5 Linear	5T	5 Diamond	32 Full Connectivity
1.0		2 (25%)	2 (25%)	0	0
1.1		0	0	0	0
1.2		0	0	0	0
1.3		0	0	0	0
1.4		0	0	0	0
1.5		0	0	0	0
1.6		0	0	0	0
1.7		0	0	0	0
1.8		0	0	0	0
1.9		12 (67%)	6 (33%)	6 (33%)	0
1.10		4 (29%)	4 (29%)	0	0
1.11		8 (67%)	10 (83%)	4 (33%)	0
1.12		4 (22%)	4 (22%)	0	0
1.13		12 (35%)	14 (41%)	6 (18%)	0
1.14		20 (36%)	18 (33%)	10 (18%)	0
1.15		14 (11%)	14 (11%)	0	0
1.16		8 (18%)	12 (27%)	0	0
1.17		46 (37%)	46 (37%)	0	0
1.18					0
1.19		4 (12%)	4 (12%)	0	0
1.20					0
1.21		0	0	0	0
1.22		0	0	0	0
1.23		0	0	0	0
1.24		0	0	0	0
1.25		0	0	0	0
1.26		22 (22%)	18 (18%)	8 (8%)	0
1.27		20 (182%)	16 (145%)	6 (55%)	0
1.28		4 (21%)	4 (21%)	0	0
1.29		32 (44%)	20 (28%)	12 (17%)	0
1.30		14 (13%)	24 (22%)	0	0
1.31		10 (26%)	14 (37%)	6 (16%)	0
1.32	0	0	0	0	0
1.33		0	0	0	0
1.34		0	0	0	0

Circuit ID	Single Qubit	5 Linear	5T	5 Diamond	32 Full Connectivity
1.35		0	0	0	0
1.36		6 (18%)	6 (18%)	0	0
1.37		4 (31%)	4 (31%)	0	0
1.38					0

Table 4.3: Number of SWAP gates added to each algorithm when scheduled for each of the 5 different hardware configurations. Beside each cell entry is a percentage showing the percent increase in instructions by adding SWAP gates.

As can be seen in Table 4.3 above, Algorithm 1.17 (two qubit Grover) needs the most SWAP gates for both the Linear (5 Linear) and the T shaped (5 T) hardware configurations, but none are needed for the higher connectivity diamond and fully connected configurations. Surprisingly, Algorithm 1.29 (four qubit QFT) requires the most SWAPS for the diamond configuration, for which very few other algorithms even needed SWAPS. Interestingly, a large number of algorithms require no additional SWAP gates. This could be due to the fact that many of these algorithms are already designed for such limited connectivity hardware. In particular, this is true for the five-diamond arrangement, as many of the algorithms tested in these experiments were originally designed for this configuration by IBM researchers before they were used in this research. This means that the algorithms are unlikely to require any SWAP gates as they were designed with the restrictions of the five-diamond hardware in mind.

Additionally, Table 4.3 shows the percentage increase in algorithm instructions as a result of adding SWAP gates. For instance, Algorithm 1.19 (Bernstein-Vazirani) has four added instructions, which creates a 12% increase in the total number of instructions in the algorithm. For the most part, these percentages are fairly small with the exception of Algorithm 1.27 (QEC Repetition code syndrome measurement), which more than doubles the number of instructions after SWAP gates are added (106% - 128%). This makes sense as the algorithm exposes very few opportunities for parallelism and is quite short, such that any number of added SWAP gates would constitute a larger percentage change [67].

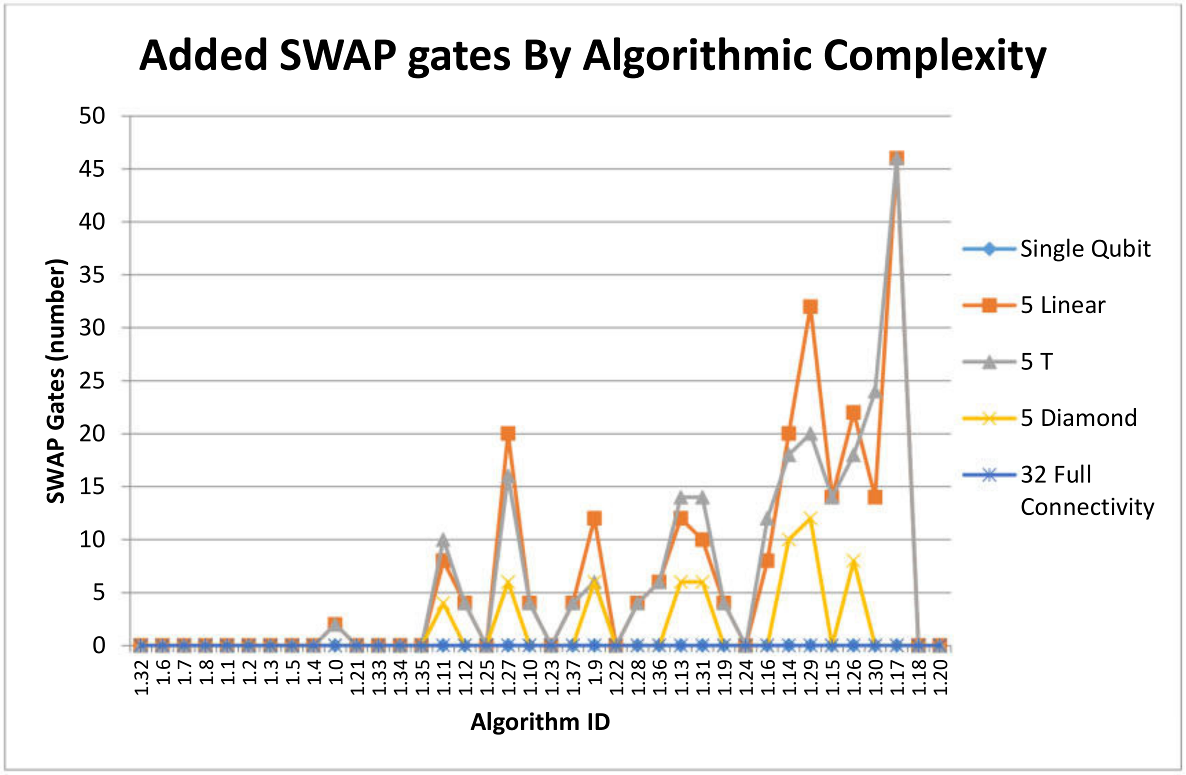


Figure 4.2: The number of added SWAP gates introduced to the algorithms by the routing step of hardware scheduling. A larger number indicates more SWAP gates added to the algorithm.

Figure 4.2 shows the number of SWAP gates added to each algorithm, ordered by its computed algorithmic complexity. As can be seen, there is a visible trend towards more complex algorithms requiring more SWAP gates to be added as there are generally higher peaks on the chart as you move from left to the right (increasing complexity). However, this trend is not as pronounced as the one for instruction stages in Figure 4.1, particularly with some algorithms, such as 1.11 and 1.27, creating several spikes earlier than any of the others with similar magnitudes. This is most likely due to the fact that the algorithmic complexity value does not take into account how instructions interact with each other. For instance, the complexity metric could be improved by taking into account the number of qubits that each instruction uses or by accounting for instructions that block or rely on others to be completed.

Additionally, there is a large difference in the number of added SWAP gates between the various tested hardware configurations. In general, it appears as if the more connected hardware configurations require fewer SWAP gates to be added to the scheduled algorithm. This makes sense in that SWAP gates are only required to conform to hardware restrictions [28]. Less restrictions means that less SWAP gates are required. The 5-qubit 'T' configurations requires the most SWAPs, followed by the five qubit linear arrangements, the five qubit diamond arrangement and lastly, the fully connected 32-qubit arrangement, which does not add any SWAPs to the algorithms tested. This finding is expected as per the existing literature, as routing is only important in near-future quantum computing due to the fact that qubit connectivity is still very much constrained [8]. Quantum computers like the 32-qubit machine used in this research have such a high degree of qubit connectivity that they require minimal to no hardware scheduling. This is visible for all the quantum algorithms tested in this research. However, devices with full connectivity, like the 32-qubit machine, will most likely not be widely available for near-future computers [8].

4.5 Estimated Run Time

Another metric that is gathered through the experiments is an estimation of how long each quantum algorithm takes to run. These estimates are made both before scheduling, by assuming that each instruction is run sequentially to the last, as well as after scheduling. Each type of instruction has an estimated run time as determined by other literature, including IBM's device specifications as described in Chapter 3, and the total run time is just the sum of all the individual instruction estimates. For more details about how these run time estimations are determined, refer to Chapter 3. Using these estimated times, the amount of time saved by using the scheduled version of the algorithm over the original can be estimated. This is important due to the decoherence experienced by qubits. Since qubits decohere so quickly, saving as much time as possible is important to ensure quantum algorithms are able to successfully complete before they decohere [54]. The shorter the time it takes to complete a quantum algorithm, the better that algorithm will perform on near-future quantum computers.

Circuit Id	Single Qubit	5 Linear	5 T	5 Diamond	32 Full Connectivity
1.0		0.0006	0.0006	0	0
1.1		-1	-1	-1	-1
1.2		-1	-1	-1	-1
1.3		-1	-1	-1	-1
1.4		-1	-1	-1	-1
1.5		-0.0001	-0.0001	-0.0001	-0.0001
1.6		0	0	0	0
1.7		-0.0001	-0.0001	-0.0001	-0.0001
1.8		0	0	0	0
1.9		-1.997	-1.9988	-1.9988	-2.0006
1.10		-1.9993	-1.9993	-2.0005	-2.0005
1.11		-2.9979	-2.9973	-2.9991	-3.0003
1.12		0.0007	0.0007	-0.0005	-0.0005
1.13		0.0021	0.0021	0.0003	-0.0015
1.14		0.0016	0.0022	-0.0002	-0.0032
1.15		0.0022	0.0022	-0.002	-0.002
1.16		-3.9985	-3.9979	-4.0009	-4.0009
1.17		-1.99	-1.99	-2.0038	-2.0038
1.18					-4.003
1.19		-0.9995	-0.9995	-1.0007	-1.0007
1.20					-8.0097
1.21		-0.0001	-0.0001	-0.0001	-0.0001
1.22		0	0	0	0
1.23		0	0	0	0
1.24		-0.0004	-0.0004	-0.0004	-0.0004
1.25		-1.0002	-1.0002	-1.0002	-1.0002
1.26		-2.996	-2.9972	-2.999	-3.0014
1.27		0.0054	0.0036	0.0018	0
1.28		0.0011	0.0011	-0.0001	-0.0001
1.29		0.0031	0.0013	-0.0005	-0.0041

Circuit Id	Single Qubit	5 Linear	5 T	5 Diamond	32 Full Connectivity
1.30		-0.0025	-0.0001	-0.0067	-0.0067
1.31		0.0014	0.002	0.0002	-0.0016
1.32	0	0	0	0	0
1.33		-0.0001	-0.0001	-0.0001	-0.0001
1.34		0	0	0	0
1.35		-1	-1	-1	-1
1.36		-1.9988	-1.9988	-2.0006	-2.0006
1.37		-1.999	-1.999	-2.0002	-2.0002
1.38					-8.03

Table 4.4: Estimated run time savings for each algorithm, measured in milliseconds, after hardware scheduling across all 5 hardware configurations. Negative values indicate that the scheduled algorithm was faster than the original by the given amount.

Table 4.4 shows the results of computing the estimated time saved by performing hardware scheduling on each of the tested algorithms across each hardware configuration. The raw data for the before and after run-time estimates can be found with my source code, but are too difficult to show cleanly on the table. Algorithm 1.18 (adder) only runs on the 32-qubit hardware and, as such, only has run-time estimates for that hardware configuration. A negative time from the table indicates that the optimized version is faster than the original, whereas a positive value indicates that the scheduled version is slower than the original by that amount due to the addition of the SWAP instructions. Zero indicates that the algorithm takes the same amount of time to run before hardware scheduling as it does after hardware scheduling. For the most part, all the algorithms benefit in terms of estimated run time by undergoing hardware scheduling. Algorithm 1.38 (Shor's Algorithm) and algorithms 1.16 (the max cut problem) and 1.18 (the adder) seem to benefit the most from hardware scheduling even though they only run on the 32-qubit machine. This is most likely because they are quite large algorithms and possess more opportunities for parallelism. However, there are some algorithms which, on certain hardware configurations, do not benefit from scheduling. This could be because there are very few operations which are parallelizable, meaning they could not be scheduled at the same time as other operations, or because a lot of SWAP gates are added to the algorithm through hardware routing to be compatible with the underlying connectivity of the quantum computer.

Figure 4.3 shows the estimated run-time savings as a result of applying the hardware scheduling algorithm to each input quantum algorithm. The results in the chart have been sorted in order of increasing algorithmic complexity. Unlike the previous comparisons, there does not appear to be much of a correlation between the algorithmic complexity and the estimated run-time savings. It appears as if the run-time savings is more dependent on the implementation specifics for each quantum algorithm rather than on any kind of generalized trend around an abstract definition of complexity. This makes sense in many respects because the choice of instructions is up to the programmer and there are likely several different sequences of instructions which create equivalent results. A sequence could be chosen such that it uses more parallelizable instructions over one that makes use of barriers, measurements, or other instructions which can limit parallelization. However, even considering that, some of the more complex algorithms that benefitted from this scheduling seem to benefit more than some of the simpler ones. This is emphasized by Algorithm 1.20 (big adder), which has a much larger peak than any of the other algorithms.

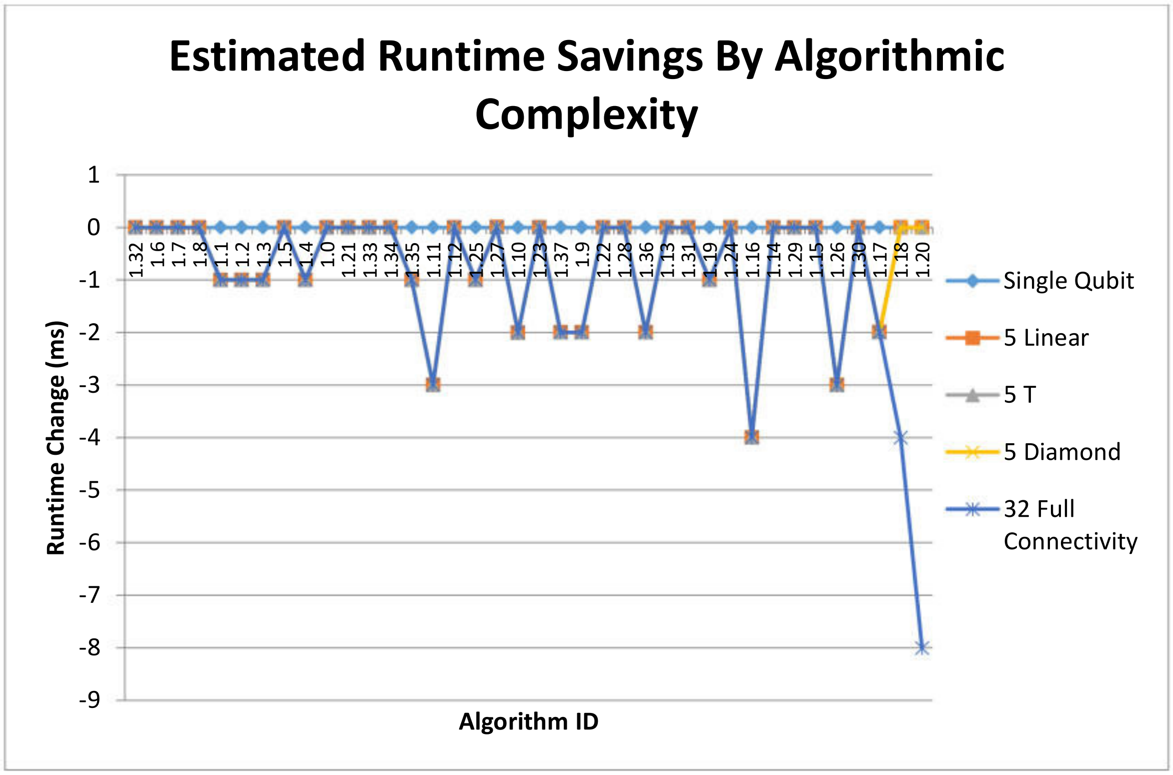


Figure 4.3: Difference between estimated run time before and after hardware scheduling for all experimented algorithms on all five hardware configurations. A negative value indicates that the scheduled algorithm is that much faster than the original sequential algorithm.

4.6 Scheduling Algorithm Run Time

The last metric gathered by the experimental work is how long the hardware scheduling optimization takes to perform. Intuitively, it is desirable for this time to be as small as possible so that compiling large quantum algorithms would not be too time consuming. Table 4.5 below contains a list for how long each scheduling pass took (on average) for each algorithm on each of the five hardware configurations.

Circuit Id	Single Qubit	5 Linear	5 T	5 Diamond	32 Full Connectivity
1.0		1.8251	2.0426	1.7782	1.8393
1.1		1.7601	1.8483	3.7205	1.7110
1.2		1.6959	1.7286	1.7071	3.6003
1.3		1.8558	1.7796	1.7631	1.7201
1.4		1.8035	2.1585	1.7969	1.8362
1.5		1.8717	1.7975	1.8861	1.7891

Circuit Id	Single Qubit	5 Linear	5 T	5 Diamond	32 Full Connectivity
1.6		1.6916	2.9826	1.7535	2.1182
1.7		1.7955	1.7533	1.7864	1.9535
1.8		1.8029	1.9694	2.1931	1.9091
1.9		2.0938	2.0024	1.9751	1.9440
1.10		2.0125	2.0095	1.9732	1.9886
1.11		2.0830	2.4467	2.2388	1.9285
1.12		2.0072	2.0771	1.8872	2.0331
1.13		2.4174	2.6492	2.5436	2.1987
1.14		2.9430	3.3876	2.7554	2.6383
1.15		4.0355	4.2812	3.7617	4.1300
1.16		2.3743	2.4139	2.4416	2.3891
1.17		4.3804	5.2432	4.5246	4.3519
1.18					5.1424
1.19		2.4053	2.4509	2.5699	2.3142
1.20					10.1794
1.21		1.7230	1.8018	1.7267	1.7688
1.22		2.0383	12.1105	2.1668	2.5940
1.23		2.0024	2.1174	1.9864	2.0387
1.24		3.7081	3.9219	3.4061	3.7924
1.25		1.8969	1.9118	1.8662	1.8984
1.26		3.6271	4.1528	3.6565	3.8832
1.27		2.1257	1.9774	1.9208	1.8737
1.28		1.9522	2.0631	1.9406	2.1116
1.29		3.8121	3.5147	3.1090	8.4315
1.30		3.6569	4.1657	3.2881	3.4693
1.31		2.3598	2.6799	2.7155	2.2130
1.32	1.6691	1.5868	1.6313	1.5659	1.6026
1.33		1.8650	1.9880	1.8425	1.9452
1.34		1.7567	2.5974	1.7232	1.8330
1.35		1.7813	1.8797	1.7060	1.8552
1.36		2.2176	2.1496	2.2185	2.1693
1.37		1.8030	1.8410	1.7601	2.1340
1.38					24296.24

Table 4.5: Compilation times for the hardware scheduling compiler pass applied to each algorithm in the experiment for each of the five different hardware configurations. All of the times in the table are measured in milliseconds.

From Table 4.5, it should be noted that all the algorithms take quite a bit less than a second to perform hardware scheduling, except for the longest, which is 1.38 (Shor's algorithm) at 24s. As for the rest of the algorithms, the longest running are algorithms 1.22 (the inverse Quantum Fourier transform) on the T-configuration, and 1.20 (the big adder) and 1.29 (Quantum Fourier transform) on the 32 qubit fully connected hardware. In the case of Algorithm 1.20, this time can be explained by it being the largest and most complicated of the algorithms, with the exception of Shor's Algorithm. For Algorithms 1.29 and 1.20, these seem to be outliers in the data. In the case of 1.29, the slow scheduling time is on the 32-qubit hardware, which should be the fastest hardware due to its full connectivity. Additionally, 1.29 is scheduled quickly on all other hardware. This indicates that the comparatively large scheduling time for this algorithm on the 32-qubit hardware may be the result of something external that might be slowing down the experiment's run time, such as garbage collection or computer updates. In the case of Algorithm 1.22, the scheduling time is short on all hardware with the exception of the T configuration. This may be due to that algorithm being poorly suited for that particular hardware configuration, thus requiring more time in the routing algorithm portion of the scheduler. Even with both of these algorithms being outliers to the general curve, it should be noted that their scheduling times are still much less than a second, meaning the added scheduling time should not be overly apparent to the end user.

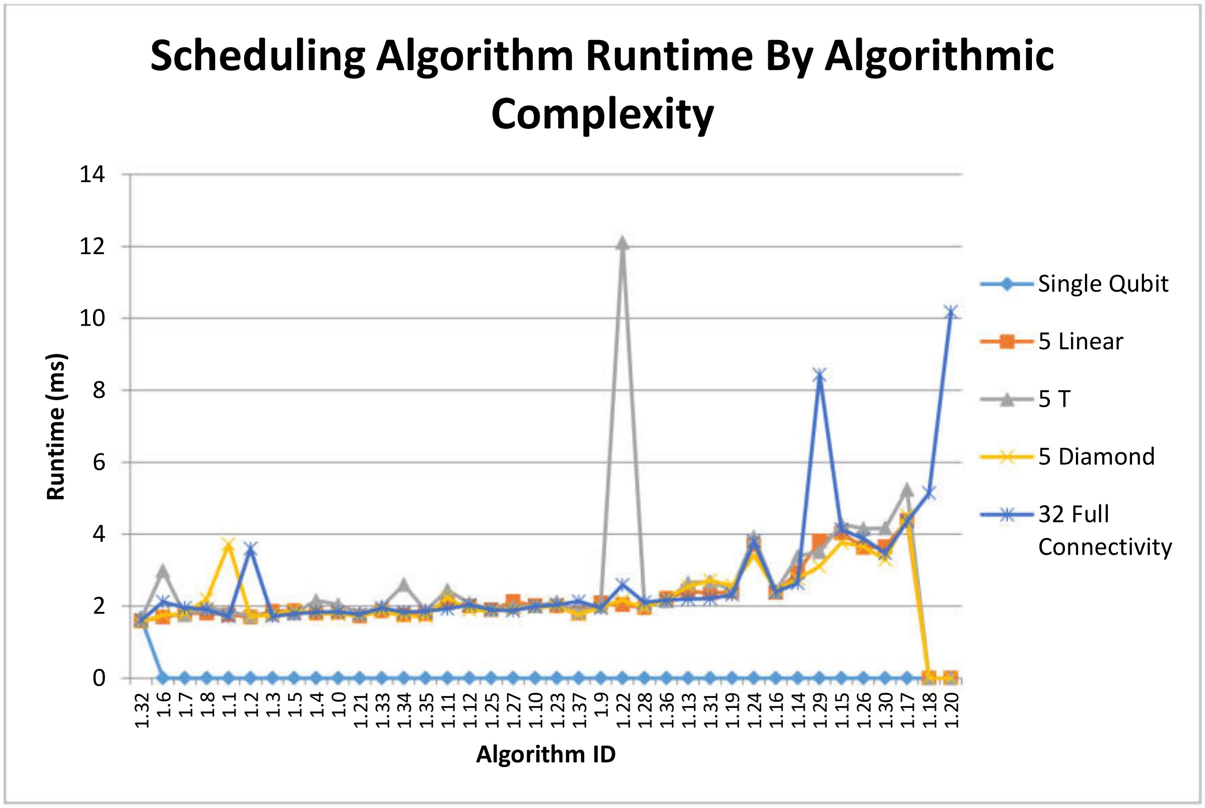


Figure 4.4: The amount of time taken by the hardware scheduling algorithm for each algorithm across each of the five hardware configurations.

Figure 4.4 above shows the compilation times from Table 4.5, sorted in order of algorithmic complexity, with the exception of Shor's algorithm since it would skew the chart too much. From the figure, one can see the same trend as with the table. The majority of algorithms take virtually no time to perform hardware scheduling on. Generally speaking, increased complexity results in increased time required to perform the scheduling operation. Algorithm 1.22's outlying behavior on the T configuration can be easily discerned from the graph compared to its behavior on the other hardware configurations. Additionally, it should be noted that algorithms 1.20 and 1.18 only work on the 32-qubit simulator, resulting in their data dropping to 0 at the right-hand side of the chart.

4.7 Effectiveness of Hardware Scheduling

In order to determine how efficient or effective this particular hardware scheduling algorithm is at scheduling various quantum algorithms, an "efficiency" measure needs to be defined. The dictionary defines efficiency[36] as an activity performed in the best possible manner with the least waste of effort or time, while effectiveness[37] is described as an activity adequate for accomplishing a goal. While both effective and efficient actions aim to accomplish a goal, an effective action does not have to be the best or most efficient one. For this research, the efficiency measure is defined according to the following equation.

(4.1)

$$\text{efficiency} = \max\left(\frac{t_i - t_f}{t_c}, 0\right) \quad \text{where}$$

t_f = running time of the algorithm after scheduling

t_i = running time of the algorithm before scheduling

t_c = compilation time

Equation 4.1: Definition of the efficiency measure.

The measure is defined such that it is a positive value if any amount of run time is saved through the application of the hardware scheduling algorithm ($t_f < t_i$). If the value is equal to '1,' then there is an equal amount of run time saved to the amount of additional time spent compiling the algorithm with this researcher's hardware scheduler. The maximum function (max) will return the value of the division if the value is greater than 0. The purpose of the application of the maximum function is to limit the results to strictly positive values for the case where algorithms actually become longer to run after hardware scheduling. For these cases, applying the hardware scheduling algorithm is considered to not be efficient at

[36] Definition is derived from dictionary.com
[37] Definition is derived from dictionary.com

all and is set to 0. Both run time and compilation time are important to quantum computing as the run time saved helps avoid the issue of qubit decoherence whereas the compilation time spent is important because it is experienced each time the algorithm input is changed as the input gets compiled into the algorithm.

Table 4.6 below shows the computed normalized efficiency values for each of the tested quantum algorithms for all five hardware configurations. With this measure, values greater than 0 are considered effective because they do optimize the original quantum algorithms; however, they may not the most efficient since it takes more time to optimize them than is saved. Values greater than 1 are considered efficient because more time is saved when running them than is spent optimizing them.

Circuit Id	Single Qubit	5 Linear	5 T	5 Diamond	32 Full Connectivity
1.0		0	0	0	0
1.1		0.5681	0.5410	0.2688	0.5844
1.2		0.5897	0.5785	0.5858	0.2778
1.3		0.5388	0.5619	0.5672	0.5813
1.4		0.5545	0.4633	0.5565	0.5446
1.5		5.34E-05	5.56E-05	5.3E-05	5.59E-05
1.6		0	0	0	0
1.7		5.57E-05	5.7E-05	5.6E-05	5.12E-05
1.8		0	0	0	0
1.9		0.9538	0.9982	1.0120	1.0291
1.10		0.9934	0.9949	1.0138	1.0060
1.11		1.4392	1.2250	1.3396	1.5557
1.12		0	0	0.0003	0.0003
1.13		0	0	0	0.0007
1.14		0	0	7.26E-05	0.0012
1.15		0	0	0.0005	0.0005
1.16		1.6841	1.6562	1.6386	1.6747
1.17		0.4543	0.3795	0.4429	0.4604
1.18					0.7784
1.19		0.4155	0.4078	0.3894	0.4324
1.20					0.7869

Circuit Id	Single Qubit	5 Linear	5 T	5 Diamond	32 Full Connectivity
1.21		5.8E-05	5.55E-05	5.79E-05	5.65E-05
1.22		0	0	0	0
1.23		0	0	0	0
1.24		0.0001	0.0001	0.0001	0.0001
1.25		0.5273	0.5232	0.5359	0.5269
1.26		0.8260	0.7217	0.8202	0.7729
1.27		0	0	0	0
1.28		0	0	5.15E-05	4.74E-05
1.29		0	0	0.0002	0.0005
1.30		0.0007	2.4E-05	0.0020	0.0019
1.31		0	0	0	0.0007
1.32	0	0	0	0	0
1.33		5.36E-05	5.03E-05	5.43E-05	5.14E-05
1.34		0	0	0	0
1.35		0.5614	0.5320	0.5862	0.5390
1.36		0.9013	0.9298	0.9018	0.9222
1.37		1.1087	1.0858	1.1364	0.9373
1.38					0.0003

Table 4.6: Normalized efficiency metric for all quantum algorithms on all five hardware configurations. Values greater than 1 mean that more time was saved during run time than was lost due to the compilation procedure.

As can be seen in Table 4.6, there are many algorithms for which hardware scheduling appears to be efficient. Examples of such algorithms include 1.9 (Deutsch Josza), 1.10 (Bernstein Vazrani algorithm), 1.11 (Simon's algorithm), 1.16 (Max Cut Problem), 1.20 (8 bit ripple carry adder), 1.25 (Iterative phase estimation), and 1.37 (W3test algorithm), since their efficiency metric is greater than '1' for all, or most, hardware configurations. This means they save more run time than is spent in scheduling, and are thus classified in this research as efficient. However, there are other algorithms where hardware scheduling is effective, though not necessarily efficient. These algorithms contain positive efficiency values between 0 and 1. For these algorithms, run-time benefits are received as a result of hardware scheduling, but it takes longer to schedule them compared to the amount of run time saved. Most of the algorithms benefit somewhat from hardware scheduling. Examples of algorithms in which hardware scheduling is considered effective, but not necessarily efficient, include 1.1, 1.2, 1.3, 1.4 (superdense coding), 1.33 (benchmark), 1.35 (improved teleport), and 1.36 (w state).

When each algorithm is ordered by their algorithmic complexity and the efficiency measure is plotted (Figure 4.5), one can see that the hardware scheduling algorithm developed for this research is effective for the majority of the algorithms. There is very little pattern with respect to how effective hardware scheduling is when compared to the complexity metric of the algorithm. More of the effective algorithms are in the middle of the graph (middle complexity) rather than the edges. However, it appears that the effectiveness of hardware scheduling is more dependent on the algorithms themselves rather than any generalized concept of complexity. That being said, it does appear that lower complexity algorithms have a more likely chance to be considered not effective at all. This is most likely due to the simpler algorithms being so small that they do not expose many optimisation opportunities. This could also indicate that as algorithms become more complex, there is a greater likelihood of hardware scheduling being effective on it.

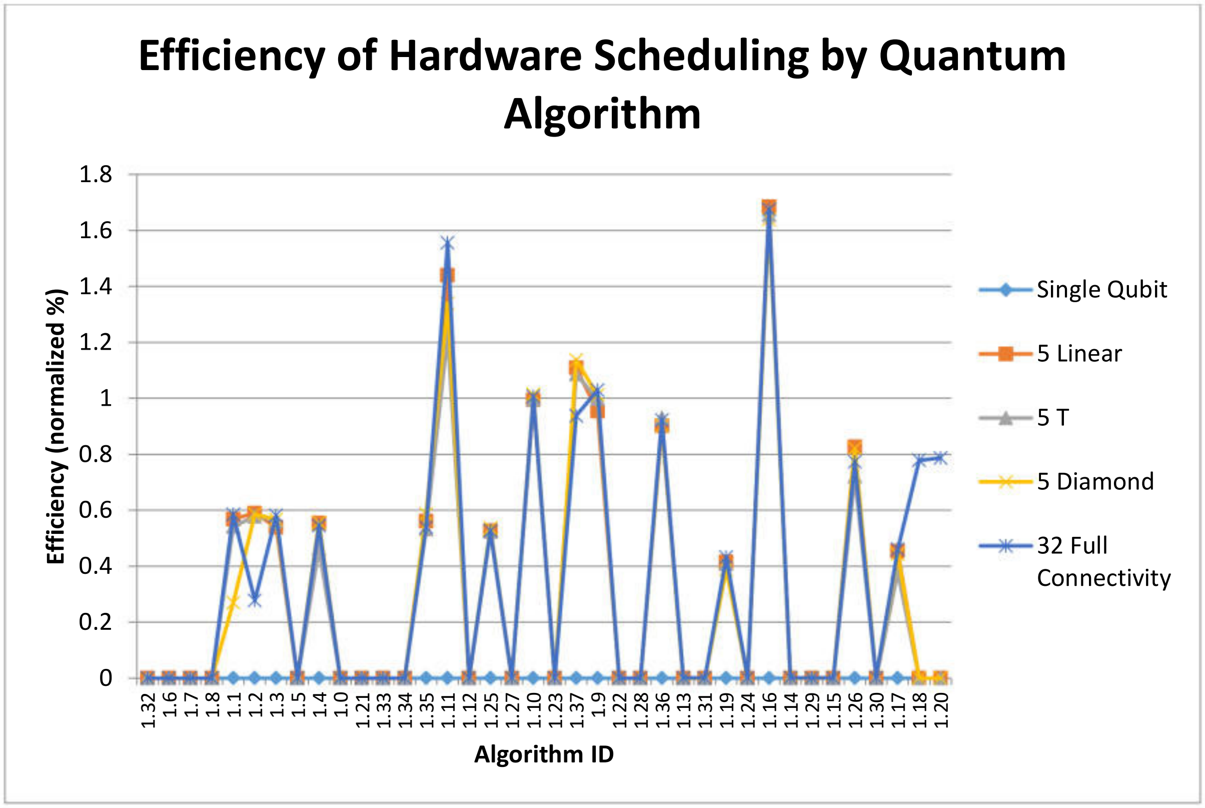


Figure 4.5: Efficiency measure for each quantum algorithm for all five hardware configurations, ordered by algorithmic complexity.

4.8 Summary

From this research, one can see the benefits and disadvantages of applying this particular hardware scheduling strategy to various quantum algorithms across a variety of publicly available quantum computer hardware configurations. It has been shown that as quantum algorithms get more complex, they are more likely to expose potential parallelism opportunities, resulting in a larger decrease in total instruction stages. When looking at the added number of SWAP gates, one can see how the number of SWAPs is greater for lower connectivity hardware compared to hardware with higher degrees of connectivity. The estimated run-time savings of each of the tested algorithms seems more dependent on specific algorithm implementation, though there does appear to be a general trend of more run time being saved through hardware scheduling as algorithms increase in complexity.

Lastly, for this particular hardware scheduling algorithm, the scheduler is determined to be effective for most of the algorithms, with a few that it is very efficient for, and some for which the scheduler is not effective at all, particularly the simplest algorithms. This may be due to a lack of opportunities to apply optimizations.

Chapter 5: Discussion

5.1 Introduction

For the practical realization of quantum computing, several issues need to be addressed, both in terms of viable hardware containing enough qubits for practical computations and in terms of software and compilers to produce efficient programs that are able to make full use of the available hardware. Since authors such as Häner et al. [2018, 39] and Murali et al. [2019, 41, 59] have indicated that instruction scheduling is one of the most important optimizations than can be employed by near-future quantum compilers, this research intends to add to the body of research on the effectiveness of a particular hardware scheduling algorithm for various quantum algorithms across a wide variety of different quantum computing hardware. To the author's knowledge, this kind of experiment, in which a scheduling algorithm is examined using many different input algorithms across a wide variety of quantum hardware, has not been performed before. For this purpose, a quantum compiler is developed in the C# programming language for OpenQASM 2.0 quantum assembly code in which a hardware scheduling algorithm is implemented. Data for key metrics is collected while running the scheduling algorithm on various quantum algorithms and hardware configurations.

This section discusses the research provided within this framework, including the clarity with which the scheduling algorithm can be implemented, the effects it has when applied to quantum algorithms, and the performance of the scheduling algorithm. The discussion within these sections draws on the analysis presented in Chapter 4, "Key Findings & Analysis." Lastly, this chapter addresses the limitations of this particular scheduling algorithm.

5.2 Clarity of Implementation

The hardware scheduling algorithm employed by this research is based generally on the ideas behind Guerreschi and Park's [2017, 13] scheduling algorithm. This algorithm was chosen specifically from other scheduling algorithms, such as the one implemented by Shi et al. [2019], due to the ease in understanding the steps involved and the effects of each step [54]. Similarly, the algorithm employed in this research is also designed around ease of implementation and understanding. The algorithm is broken up into distinct stages and passed from one stage to the next until the scheduling is done. These stages are: determining the dependencies of each quantum algorithm, assigning priorities such that one can create a preliminary ordering of operations and schedule operations with the same priorities at the same time, resolving ambiguous instruction orderings within groups, and lastly, adding routing to ensure the instructions are compatible with the underlying hardware connectivity.

Given the clarity of the hardware scheduling algorithm, improvements and modifications are easy to perform. The author considers this to be one of the most important aspects for any algorithm to possess.

5.3 Effect on Quantum Algorithms

While a clear and concise algorithm possesses the advantages of being easy to change or improve upon, such an algorithm does not always make the fastest algorithm and so the effect it has on its inputs must be examined. From the works of various researchers such as Shi et al. [2019], hardware scheduling for quantum algorithms is one of the most important optimizations to perform [54]. This is due, in part, to the limited connectivity offered by near-future quantum computers [8], as well as the short decoherence times which qubits experience [22]. In order to satisfy underlying hardware connectivity constraints, the hardware scheduling algorithm is forced to add SWAP gates to move information around the hardware and into compatible spots to perform subsequent operations [38]. At the same time, the hardware scheduler needs to groups multiple operations that can run at the same time together so that the total run time of the algorithm can be reduced in order to be performed within the hardware's qubit decoherence times [54]. Since these two operations are partially in contrast to each other (reduce sequential operations by scheduling vs. add operations by routing), it is important to reduce the number of total SWAP gates that are added to the quantum algorithm so that more effort is focused on scheduling parallel operations as opposed to adding many new sequential SWAP operations.

From the research of Gambetta et al. [2017], we know that the current generation of quantum computers experience qubit decoherence in the order of 100 microseconds [14]. Based on Figure 4.3 from the analysis section, we can see that many of the algorithms tested in this experiment are estimated to be faster after hardware scheduling than before. The largest amount of time saved is around 8 milliseconds for Shor's algorithm. Given the qubit decoherence rate mentioned earlier, a savings of 8 milliseconds would be helpful for being able to run those algorithms on near-future quantum computers. This speed increase is the largest for higher connectivity hardware configurations like the 32-qubit fully connected configuration and the diamond configuration. This is most likely due to not requiring as many SWAP gates to meet the underlying hardware connectivity constraints. By not adding SWAP gates, which add run time to quantum algorithms, the scheduling algorithm can focus entirely on parallel scheduling of operations, thereby reducing the overall algorithm run time. While the 8 milliseconds saved is quite a substantial amount of time saved when compared to qubit decoherence times, that amount of time only applies to the fully connected 32-qubit machine, with the other machines having less substantial time savings. This means that this kind of performance increase will not be practically achieved for near-future quantum computers. Britt and Humble [2017] identified that the degree of connectivity between qubits will continue to be an issue for near-future quantum computers [8]. Considering this, the performance increase as a result of applying this particular hardware scheduling algorithm will more likely be in line with the increases for the Diamond and T hardware configurations as opposed to the fully connected configuration. There would be an increase in performance, but this increase would be better with an increasing degree of connectivity between qubits.

Authors such as Gambetta et al. [2017] have noted that quantum gate implementations are not perfect on the current generation of quantum computers [14]. Each instruction not only consumes some of the coherence time for the qubit, but additionally introduces some error into the system. In this respect, not only is the algorithm run time important for executing quantum algorithms, but so is the total number of instructions. The hardware scheduling algorithm used in this research does not reduce the number of instructions; instead, it groups operations together that can be executed in parallel. Since no instructions are removed via hardware scheduling, any change in instructions will be the result of adding new instructions. For this particular implementation, instructions can only be added to quantum algorithms as a result of the routing stage of hardware scheduling, which adds SWAP gates to make groups of operations compatible with the underlying hardware. This routing occurs before each group of parallelizable operations. The largest amount of additional SWAP gates was 46 and occurred for quantum Algorithm 1.17 (3 qubit Grover). Since Algorithm 1.17 started off with 126 instructions originally, an addition of these 46 SWAP gates amounts to a total increase in instructions by 37%. Of the algorithms that required SWAP gates, this increase is consistent with the majority of other increases. While these increases are a little higher than were hoped for, the increase in instructions is still quite small with the largest increase being 46 SWAP gates. Since Shor's algorithm only ran on the 32 Qubit hardware, it did not require any SWAP gates, though it would likely have overtaken Algorithm 1.17 if the hardware was not fully connected.

One of the reasons that the increases are higher than expected might be due to this hardware scheduling algorithm not taking into account the best possible qubit mappings, partially due to a lack of lookaheads in the algorithm, as pointed out by Guerreschi and Park [2017, 13]. This leads to situations where, locally, the number of SWAP gates are minimized between scheduled stages but may not be globally minimized. For a globally minimized solution, all the stages should be considered at the same time, including those before and after the current stage. However, one notable exception to this is Algorithm 1.27 (QEC), which experienced an increase of over 100% as a result of hardware scheduling. Due to this algorithm having a small number of instructions (11), any increase as a result of added SWAPS constitutes a large percentage. Given that Algorithm 1.27 is an outlier in terms of added SWAP gates, it can be assumed that it is not well tailored for the limited connectivity hardware, particularly the Linear and T shape hardware, with which Algorithm 1.27 experienced the largest percent increases.

Given that SWAP gates have been found to potentially impose significant run time drawbacks and reduce reliability [28], it is important to examine how much of a run time impact routing has for this particular hardware scheduling algorithm. One would expect that more complex algorithms would require more SWAP gates, which would result in detrimental changes to the estimated run time. However, Figure 4.3 shows an increase in time saved as algorithms become more complex, even though the number of SWAP gates also increases (Figure 4.2). This means that a hardware scheduler which can maximize the number of instructions that can be performed in parallel could, in theory, offset any added run time as a result of qubit routing. In this way, more time is saved through parallelization than is added through the use of SWAP gates needed to satisfy hardware connectivity constraints.

5.4 Performance

From the previous section, we can see that the hardware scheduling algorithm, on average, saves algorithmic run time despite adding operations necessary to satisfy hardware connectivity constraints. However, the performance of the hardware scheduler was not addressed in the previous section. This section summarizes how the hardware scheduling algorithm performs for the various experimental test cases and determines if the usage of this hardware scheduling algorithm is effective given both its effect on the scheduled algorithms and the scheduler's performance.

Even though each of the experiments for this hardware scheduling algorithm resulted in faster estimated run-time performance, the scheduler is not considered efficient for many quantum algorithms due to how much additional time is required to perform the scheduling when compared against the run time saved. By increasing the performance of the scheduling algorithm (decrease the run time of the scheduler), more algorithms may switch from being effective to being efficient. One possible avenue for improvement is to improve the heuristic used in the A* routing stage. The heuristic employed for the A* search used in this research is a very simple one, which means that the pathfinding algorithm will probably take longer than its "best-case" performance, which is heavily influenced by the heuristic chosen. Even though this scheduler may not be considered efficient for many algorithms, the individual compilation times are still less than a second in most cases. This means that almost all algorithms benefit from hardware scheduling and, while not efficient, are still fast enough for users of the compiler to not notice any major delays. This is most likely true because the hardware configurations used in this research are quite small. However, as hardware becomes larger and more complex, the state space for the A* search will rapidly increase, making the heuristic play a larger role in determining the search's run time. While a better heuristic is not needed now, it may be required for future hardware configurations.

The hardware scheduling strategy employed by this research does appear to decrease the run time of various quantum algorithms, in some cases by a significant amount, even after requiring SWAP gates to match hardware connectivity, without requiring too much additional compilation time to perform. There is a trend towards this algorithm being at least effective, if not efficient, for larger algorithms and given that quantum algorithms are likely to become even more complex over time in order to solve larger problems, this particular scheduler could be an effective tool for scheduling on those kinds of algorithms because it does create a positive run time benefit without costing too much in additional compilation time.

5.5 Limitations

This research is subject to several limiting factors. These factors come both from the environment in which the algorithm was developed and the environment which performed the experiment, as well as from issues encountered through experimental testing. This section will discuss these limitations beginning with those that resulted from the various environments which were used for the research, followed by the limitations from encountered issues.

Since the research was performed using C# .NET Core 3.1.301, on a single thread of an i7 8700K processor, the prior discussion related to the performance of the hardware scheduling algorithm is specific to that particular device and software configuration. As a result, changes to any of these could greatly impact the performance of hardware scheduling. For instance, a different version of .NET could result in different performances as the compiler may perform different optimizations or the run-time environment could implement base functions differently, thereby changing how they perform. Faster processors or modifications to the scheduling algorithm enabling it to make full use of multiple threads would greatly affect the overall performance of the hardware scheduling.

This hardware scheduling algorithm was also only tested theoretically against hardware configurations like those provided by the IBM Quantum Experience platform. Ideally, it would be better to have been able to verify these findings on real quantum computer hardware. Originally, the plan was to compare the estimated run-time estimates after hardware scheduling with the actual run times of the algorithms run through the IBM Quantum Experience API to determine the accuracy of the analysis. This was unable to be performed for this particular research due to several unexpected issues that arose with using the IBM Quantum Experience API. Firstly, there was no publicly available documentation describing the API's available service URLs and the expected parameters of each service. This meant that the interface had to be reverse engineered from their official Python library. Some documentation suggested that the IBM Quantum Experience API had its own compiler for submitted algorithms, which would behave differently from the one in this research, thus making any results incomparable. Additionally, undocumented behaviour occurred during testing in which algorithms that were not compatible with the IBM quantum computers seemingly ended up getting stuck in a queue on IBM’s server rather than returning some kind of failure or cancellation code, preventing further algorithms to be run on that machine for my IBM profile.

Lastly, this hardware scheduler was only really tested against five different hardware configurations. As such, the findings are only applicable to those hardwares. As new hardware is developed with potentially greater numbers of qubits, it will be important to examine the performance of this scheduler under those conditions.

Chapter 6: Summary & Conclusion

This chapter provides a summary of the research performed and the key findings. It then discusses future avenues for research and offers some concluding remarks.

6.1 Research Summary

For this thesis, the goal was to examine the effectiveness of a particular hardware scheduling algorithm when applied to real quantum algorithms across a variety of currently existing and publicly available quantum computing hardware. In order to do this, an OpenQASM compiler was developed in the C# programming language, which included a hardware scheduling algorithm inspired by the previous work of Guerreschi and Park [2017, 13]. This hardware scheduling algorithm followed the same broad steps as those indicated in Guerreschi and Park's work, but differed in the implementation of each of the algorithm's steps. Of particular note is the difference to the routing stage of the hardware scheduler in which this research used the A* graph search algorithm to perform routing on arbitrary hardware qubit connectivity configurations. The compiler was used to test 39 quantum algorithms, chosen from various sources, against nine different quantum hardware configurations, chosen from the IBM Quantum Experience, spanning five different connectivity configurations. This research was intended to have the scheduled algorithms run on real quantum computing hardware through the IBM Quantum Experience, though complications with this API prevented this from being done within the time frame of this research.

To accomplish the task of developing a quantum compiler, literature was reviewed covering topics across the field of quantum computing and compiler design. This included how quantum computer hardware works, the kinds of instruction sets that are currently being used or researched for quantum computers, what assembly languages exist for quantum computing, as well as existing quantum compilers and compiler optimization strategies including instruction scheduling. This literature was fundamental in the development of the compiler which was implemented in the C# programming language for the IBM OpenQASM 2.0 quantum assembly language. This research's hardware scheduling algorithm provides a clear concise implementation which is broken up into distinct stages that are easy to understand and modify.

In order to test the effectiveness of the implemented hardware scheduling algorithm, a testing strategy was developed which involved several experimental trials. A select subset of the 39 quantum algorithms was chosen for each trial. Each algorithm was analyzed and then scheduled on a subset of the nine different quantum computing devices. Since the IBM Quantum Experience API was not actually used to run the scheduled algorithms, results from several machines were identical due to there being only five distinct qubit connectivity configurations. Data were collected for eight different metrics, including the qubit count, estimated run time before scheduling, estimated run time after scheduling, change in the estimated run time, the number of instructions, the change in the number of instructions before and after scheduling, the number of swaps that were added to the algorithm through routing, and lastly the time taken to perform the hardware scheduling. These metrics were summarized in a series of tables, converted into charts, and used in equations to analyze the effect of the hardware scheduling on each algorithm for all five qubit configurations and to determine the effectiveness of the hardware strategy.

By analyzing the results of the experimental trials, the research showed that the particular hardware scheduling algorithm implemented in this research provided varying levels of effectiveness for the tested quantum algorithms. Certain algorithms were unable to be scheduled for certain hardware configurations. For instance, Algorithm 1.18 (adder) was only able to be scheduled on the 32-qubit quantum computer since it requires more qubits than the other hardware configurations could support. When examining how the number of algorithm stages change when hardware scheduling is applied, Algorithm 1.38 (Shor's Algorithm) and Algorithm 1.30 (five qubit Quantum Fourier Transform) benefitted the most from scheduling, saving a large number of stages over all hardware configurations (8357 for 1.38, 50 for 1.30). Other Quantum Fourier Transform variants also showed a large change in the number of algorithmic steps. This research found that as the complexity of algorithms increased, the potential for saving algorithmic steps through hardware scheduling also increased. Most algorithms required a small number of added SWAP gates to meet the connectivity constraints of the underlying hardware, many of which only needed 20 percent more instructions than the original algorithm. However, Algorithm 1.27 was a glaring exception to this as the added SWAP gates nearly doubled the number of instructions in the algorithm. This was most likely due to the algorithm being an error correction algorithm, which other researchers have noted could introduce significant run-time overhead [67]. Even though the number of algorithmic steps decreased with increasing complexity, the estimated amount of time saved by hardware scheduling was more dependent on the individual algorithm rather than on the complexity trend. For the most complicated algorithm, Shor's algorithm, the performance increase was about 8 milliseconds faster than the original unscheduled algorithm. While Shor's algorithm showed considerable run-time savings, the numbers were much smaller for the less complicated algorithms, likely due to fewer opportunities for parallelism. Besides less complicated algorithms showing less change in their run times, the savings were dependent on the connectivity of the hardware, with larger run time savings for higher connectivity hardware. This makes sense as fewer SWAPs would be required to move qubits around the hardware in order to perform multi-qubit instructions. By observing the number of case where scheduling was not effective, there was an indication that as algorithms increased in complexity, they were more likely to benefit in some way from hardware scheduling.

Considering all the compiled metrics, the hardware scheduling algorithm can be considered effective for most cases since there was almost always some amount of run-time savings. For algorithms 1.9 (Deutsch Josza), 1.10 (Bernstein Vazrani algorithm), 1.11 (Simon's Algorithm), 1.16 (Max Cut Problem), 1.20 (8 bit ripple carry adder), 1.25 (Iterative phase estimation), and 1.37 (W3test algorithm), hardware scheduling can be considered not only effective, but efficient as well, since the algorithms, when scheduled, saved more run time than they spent during the scheduling. For other algorithms of middling complexity and Shor's algorithm, the scheduler was quite effective but not necessarily efficient. However, for algorithms that were too simple, this particular hardware scheduler could not be considered effective at all. For simple algorithms, there were simply not enough instructions to rearrange in order to gain performance increases. Even for the cases where the scheduling was not considered effective, the scheduling procedure was fast. This implies that this scheduling algorithm would benefit any quantum compiler due to its potential for improving quantum algorithms without much of a noticeable impact to the users of the compiler.

6.2 Future Research

This research adds to the body of research on the effectiveness of hardware scheduling since, to the author's knowledge, no other experiment has been performed examining hardware scheduling across different algorithms on a wide variety of current generation quantum computers. This research suggests several different avenues for pursuing further investigations on the topic of hardware scheduling for near-future quantum computers. Some of these avenues come from the limitations posed by this research; other avenues come from alternative approaches to the problem of hardware scheduling.

One obvious avenue for future research comes in the form of testing this particular hardware scheduling strategy across different computer configurations, .NET versions, or even across different programming languages entirely in an effort to determine if there is potential for even better performance of the scheduler under different conditions or environments. Different .NET versions could have different performance costs or compiler optimizations that could lead to changes in the algorithm's performance. Different languages, such as Rust, could help determine if garbage collection created any impact on the scheduling times, since manual memory management avoids any overhead that garbage collection might incur on program execution. Another avenue for future research would be in modifying the scheduling algorithm used within this research, such as running it in parallel. By making full use of the number of processors available in the computing environment, performance could be drastically changed. With a faster scheduling algorithm, even more quantum algorithms could be considered efficient under this research's metric.

Another possible modification to the scheduling algorithm would be to modify the A* search heuristic, as different heuristics could allow for more efficient qubit routing. This is especially true since the heuristic involved in this research was quite basic and could have impacted overall run-time performance of the scheduler. Testing several heuristics and comparing time taken by the heuristic to total time taken to perform the hardware routing search could lead to a better choice of heuristic that could further increase the performance of the hardware scheduler. While a better heuristic is not needed now, it may be required for future, more complicated, hardware configurations. This research also does not address any lookahead strategies which could be used to make globally better routing choices and further reduce the number of added SWAP gates that are required in order to allow algorithms to meet the connectivity constraints for a given hardware, which might allow for compiled algorithms to have further reductions in run time.

Another idea for future research comes in the form of testing hardware scheduling with more complicated algorithms and a wider variety of hardware configurations. This research only covered five different hardware configurations, but more complicated and powerful hardware is being created all the time which could drastically impact the results of the experiments. As for the algorithms being tested, researchers are making increasingly more complicated algorithms, much more complicated than Shor's Algorithm, and this research does not cover the hardware scheduling effects of these kinds of quantum algorithms.

This research is also only done from a theoretical perspective. It would be a benefit to this field of research to actually be able to verify these performance changes against real quantum computing hardware, either by integration with the IBM Quantum Experience API, overcoming the challenges like queuing or compilers changing the algorithms, or using some other service in which those issues are non-existent.

Lastly, other researchers such as Shi et al. [2019] have identified that the gate approach, as is utilized in this research, might not be the best model for implementing hardware scheduling [54]. In the gate approach, the problem of hardware scheduling is similar to that of instruction scheduling in classical computing where discrete gates are reorganized according to how long they take and which resources they need exclusive access to. Shi et al. [2019] argue that because quantum computers do not operate using gates but use electrical control pulses instead, other methods of hardware scheduling using these control pulses could be more effective [54]. This avenue of research would involve the design of computing models, either as abstractions of the control pulses or the control pulses themselves, or the design of new hardware scheduling strategies unlike those used for the gate model.

6.3 Concluding Remarks

This thesis investigated the effectiveness of a hardware scheduling algorithm for quantum computers that employ the A* search algorithm in order to support arbitrary hardware qubit configurations. For this, a compiler was developed which used the implemented hardware scheduling strategy and a methodology was designed to test this strategy to determine its effectiveness. The strategy implemented in this research was effective for the majority of quantum algorithms, with the potential to continue to be effective as the algorithms get more complicated. However, the scheduling algorithm implemented in this research was not effective for the most basic complexity quantum algorithms and it was only really efficient for the middle complexity algorithms. This means that additional research is required in order to further improve scheduling performance for near-future quantum computers to make this algorithm not only effective, but efficient for larger and more complex algorithms considering algorithms are quickly growing more complex as time progresses.

References

[1] Roland Rüdiger. 2007. Quantum programming languages: An introductory overview. *Comput J,* 50, 2, 134-150, 16 pages. DOI: https://doi.org/10.1093/comjnl/bxl057

[2] Peter Selinger. 2004. A brief survey of quantum programming languages. In Yukiyoshi Kameyama and Peter J. Stuckey (Eds.), *Functional and Logic Programming. FLOPS 2004. Lecture Notes in Computer Science*, Vol. 2998 (pp. 1-6). Springer, Berlin & Heidelberg, Germany. DOI: https://doi.org/10.1007/978-3-540-24754-8_1

[3] Phillip Kaye, Raymond Laflamme, and Michele Mosca. 2007. *An Introduction to Quantum Computing.* Oxford University Press, New York, NY, USA.

[4] Anderson Avila, Adriano Maron, Renata Reiser, Mauricio Pilla, and Adenauer Yamin. 2014. GPU-aware distributed quantum simulation. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC '14)*, March 24-28, 2014, Gyeongju, Republic of Korea, pp. 860-865. DOI: https://dl.acm.org/doi/10.1145/2554850.2554892

[5] Willem Fouche, Johannes Heidema, Glyn Jones, and Petrus H. Potgieter. 2008. Universality and programmability of quantum computers. *Theor. Comput. Sci.*, 403*,* 121-129. https://dl.acm.org/doi/10.1016/j.tcs.2008.05.007

[6] Robert S. Smith, Michael J. Curtis, and William J. Zeng. 2016. A practical quantum instruction set architecture. *arXiv:* 1608.03355. https://doi.org/10.48550/arXiv.1608.03355

[7] Richard J. Lipton and Kenneth W. Regan. 2014. *Quantum Algorithms via Linear Algebra*. The MIT Press, Cambridge, MA, USA.

[8] Keith A. Britt and Travis S. Humble. 2017. Instruction set architectures for quantum processing units. *arXiv*: 1707.06202. Retrieved from https://doi.org/10.48550/arXiv.1707.06202

[9] Bernhard Ömer. 2005. Classical concepts in quantum programming. *Int. J. Theor. Phys.*, 44, 7, 943-955. https://doi.org/10.1007/s10773-005-7071-x

[10] Rodney Van Meter and Clare Horsman. 2013. A blueprint for building a quantum computer. *Commun. ACM*, 56, 10, 84-93. https://doi.org/10.1145/2494568

[11] Frederic T. Chong, Diana Franklin, and Margaret Martonosi. 2017. Programming languages and compiler design for realistic quantum hardware. *Nature*, 549, 180-187. https://doi.org/10.1038/nature23459

[12] Olaf Chitil. 1997. Common subexpression elimination in a lazy functional language. In Chris Clack, Kevin Hammond, & Tony Davie (Eds.), *Draft Proceedings of the 9th*
134
International Workshop on Implementation of Functional Languages, September 10-

12, 1997, St. Andrews, Scotland, UK, pp. 501-16. Retrieved March 13, 2022 https://kar.kent.ac.uk/21455/1/Common_Subexpression_Elimination_in_a_Lazy.pdf

[13] Gian G. Guerreschi and Jonsoo Park. 2017. Gate scheduling for quantum algorithms. *arXiv:* 1708.00023. Retrieved March 13, 2022 from https://arxiv.org/abs/1708.00023v1

[14] Jay M. Gambetta, Jerry M. Chow, and Matthias Steffen. 2017. Building logical qubits in a superconducting quantum computing system. *Npg Quantum Inf* 2, 1-6. https://doi.org/10.1038/s41534-016-0004-0

[15] Kristel Michielsen, Madita Nocon, Dennis Willsch, Fengping Jin, Thomas Lippert, and Hans De Raedt. 2017. Benchmarking gate-based quantum computers. *Comput. Phys. Commun.* 220, 44-55. https://doi.org/10.1016/j.cpc.2017.06.011

[16] Milan Spišiak and Ján Kollár. 2017. Quantum programming: A review. In *IEEE 14th Annual International Scientific Conference on Informatics*, November 14-16, 2017, Poprad, Slovakia, 353-358. DOI: 10.1109/INFORMATICS.2017.8327274. Retrieved March 13, 2022 from https://ieeexplore.ieee.org/document/8327274

[17] Alexander McCaskey, Eugene Dumitrescu, Dmitry Liakh, and Travis Humble. 2018. Hybrid programming for near-term quantum computing systems. *arXiv*: 1805.09279. https://doi.org/10.48550/arXiv.1805.09279

[18] Norbert M. Linke, Dmitri Maslov, Martin Roetteler, Shantanu Debnath, Caroline Figgatt, Kevin A. Landsman, Kenneth Wright, and Christopher Monroe. 2017. Experimental comparison of two quantum computing architectures. *arXiv*: 1702.01852v1. https://doi.org/10.48550/arXiv.1702.01852

[19] Mathias Soeken, Thomas Haener, and Martin Roetteler. 2018. Programming quantum computers using design automation. *arXiv*: 1803.01022v1. https://doi.org/10.48550/arXiv.1803.01022

[20] Xiang Fu, L.Riesebos, M.A. Rol, J. van Straten, J. van Someren, N. Khammassi, I. Ashraf, et al. 2018. eQASM: An executable quantum instruction set architecture. *arXiv*: 1808.02449v2. https://doi.org/10.48550/arXiv.1808.02449

[21] N. Khammassi, G.G. Guerreschi, I. Ashraf, J.W. Hogaboam, C.G. Almudever, and K. Bertels. 2018. cQASM v1.0: Towards a common quantum assembly language. *arXiv*: 1805.09607. https://doi.org/10.48550/arXiv.1805.09607

[22] Patrick J. Coles, Stephan Eidenbenz, Scott Pakin, Adetokunbo Adedoyin, John Ambrosiano, Petr Anisimov, William Casper, et al. 2018. Quantum algorithm implementations for beginners. *arXiv*: 1804.03719v1. https://doi.org/10.48550/arXiv.1804.03719

[23] Shusen Liu, X. Wang, Li Zhou, Ji Guan, Yinan Li, Yang He, R. Duan, and M. Ying. 2017. Q|SI>: A quantum programming environment. *arXiv:* 1710.09500. https://doi.org/10.48550/arXiv.1710.09500

[24] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, and Koen Langendoen. 2012. *Modern Compiler Design* (2nd Ed.). Springer Science + Business Media, New York, NY, USA.

[25] Andrew W. Appel and Jens Palsberg. 2002. *Modern Compiler Implementation in Java.* Cambridge University Press, Cambridge, UK.

[26] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. 2007. *Compilers, Principles, Techniques, & Tools* (2nd Ed.*).* Pearson Education Inc., Essex, UK.

[27] Krysta M. Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q#: enabling scalable quantum computing and development with a high-level domain-specific language. *arXiv*: 1803.00652v1. https://doi.org/10.48550/arXiv.1803.00652

[28] Alwin Zulehner, Alexander Paler, and Robert Wille. 2018. An efficient methodology for mapping quantum circuits to the IBM QX architectures. *arXiv*: 1712.04722. https://doi.org/10.48550/arXiv.1712.04722

[29] Faisal Shah Khan, Nada Elsokkary, and Travis S. Humble. 2018. Compiling adiabatic quantum programs. *arXiv:*1808.06926. https://doi.org/10.48550/arXiv.1808.06926

[30] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler transformations
for high-performance computing. *ACM Comput. Surv.* 26, 4, 345-420. DOI: https://doi.org/10.1145/197405.197406

[31] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. 2001. Operator strength reduction. *ACM Trans. Program. Lang. Syst. (TOPLAS)* 23, 5, 603-25. DOI: 10.1145/504709.504710. https://dl.acm.org/doi/10.1145/504709.504710

[32] Keith Cooper and Linda Torczon. 2012. *Engineering a Compiler* (2nd Ed.). Elsevier Inc., Burlington, MA, USA.

[33] Nullstone Corporation. 2012. *Compiler Optimizations.* Nullstone Corporation, Pleasanton, CA, USA. Retrieved March 13, 2022 from https://compileroptimizations.com/index.html

[34] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Partial dead code elimination. In *Proceedings of the ACM SigPlan 1994 Conference on Programming Language Design and Implementation*, June 20-24, 1994, Orlando, FL, USA, 147-58. Retrieved March 13, 2022 from https://dl.acm.org/citation.cfm?id=178256

[35] Pohua P. Chang and Wenwei Hwu. 1989. Inline function expansion for compiling C programs. *ACM SigPlan Notices* 24, 7, 246-57. DOI: https://doi.org/10.1145/74818.74840

[36] Manuel Serrano. 1997. Inline expansion: When and how? In Hugh Glaser, Pieter Hartel,
and Herbert Kuchen (Eds.), *Programming Languages, Implementations, and Programs: Lecture Notes in Computer Science* (Vol. 1292). Springer-Verlag, Berlin & Heidelberg, Germany, 143-157. https://doi.org/10.1007/BFb0033842

[37] Keith Cooper, Mary Hall, and Ken Kennedy. 1992, Procedure cloning. ICCL'92, *Proceedings of the 1992 International Conference on Computer Languages*, April 20-23, 1992, Oakland, CA, USA. DOI: 10.1109/ICCL.1992.185472.

[38] Davide Venturelli, Minh Do, Bryan O'Gorman, Jeremy Frank, Eleanor G. Rieffel, Kyle E.C. Booth, Thanh Nguyen, Parvathi Narayan, and Sasha Nanda. 2019. Quantum circuit compilation: An emerging application for automated reasoning. *ICAPS Scheduling and Planning Applications Workshop, International Conference on Automated Planning and Scheduling*, July 11-15, 2019, Berkeley, CA, USA. Retrieved March 13, 2022 from https://openreview.net/forum?id=S1eEBO3nFE

[39] Thomas Häner, Damian S. Steiger, Krysta Svore, and Matthias Troyer. 2018. A software methodology for compiling quantum programs. *Quantum Sci. Technol.* 3, 2, 1-18. https://doi.org/10.1088/2058-9565/aaa5cc

[40] Matthew Amy. 2019. *Formal Methods in Quantum Circuit Design*. Ph.D. Dissertation. University of Waterloo, Waterloo, Ontario, Canada. Retrieved March 13, 2022 from https://uwspace.uwaterloo.ca/handle/10012/14480

[41] Prakash Murali, Ali Javadi-Abhari, Frederic T. Chong, and Margaret Martonosi. 2019. Formal constraint-based compilation for noisy intermediate-scale quantum systems. *Microprocess. Microsyst.* 66, 102-112. DOI: https://doi.org/10.1016/j.micpro.2019.02.005

[42] Marcos Siraichi, Sylvain Collange, Vinicius Fernandes dos Santos, and Fernando Magno Quintao Pereira. 2017. Qubit allocation for quantum circuit compilers. *JIQ Conference*, November 10, 2017, Quebec City, QC, CAN.

[43] Kaitlin N. Smith and Mitchell A. Thornton. 2019. A quantum computational compiler and design tool for technology-specific targets. In *Proceedings of the 46th International Symposium on Computer Architecture*, June 22-26, 2019, Phoenix, AZ, USA, 579-88. Retrieved March 13, 2022 from https://dl.acm.org/citation.cfm?id=3322262

[44] Peng Liu, Shaohan Hu, Marco Pistoia, ChunFu Richard Chen, and Jay M. Gambetta. 2019. Stochastic optimization of quantum programs. *Computer* 52, 6, 58-67. DOI: 10.1109/MC.2019.2909711. Retrieved March 22, 2022 from https://ieeexplore.ieee.org/document/8732140

[45] Paul Colea. 2015. *Generalizing loop-invariant code motion in a real-world compiler.* Imperial College London, Department of Computing, MEng Computing Individual Project, London, UK. Retrieved June 1, 2019 from https://www.doc.ic.ac.uk/teaching/distinguished-projects/2015/p.colea.pdf

[46] Jim Kukunas. 2015. *Power and performance: Software analysis and optimization.* Morgan Kaufman Publishers, Burlington, MA, USA.

[47] Jaöa Manuel Paiva Cardoso, José Gabriel de Figueiredo Coutinho, and Pedro C. Diniz. 2017. *Embedded computing for high performance: Efficient mapping of computations using customization, code transformations and compilation*. Elsevier Inc., Cambridge, MA, USA.

[48] Chae Jubb. 2014. *Loop optimizations in modern C compilers*. Columbia University, New York, NY, USA. Retrieved March 22, 2022 from http://www.cs.columbia.edu/~ecj2122/research/x86_loop_optimizations/x86_loop_optimizations.pdf

[49] André Platzer. 2010. *Lecture Notes on Loop Transformations for Cache Optimization*. Carnegie Mellon University, School of Computer Science, Pittsburg, PA, USA. Retrieved June 30, 2019 from http://www.cs.cmu.edu/~aplatzer/course/Compilers/24-cacheloop.pdf

[50] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. 1998. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Comput.* 24, 421-44. https://doi.org/10.1016/S0167-8191(98)00020-9

[51] Davide Venturelli, Minh Do, Eleanor Rieffel, and Jeremy Frank. 2017. Compiling quantum circuits to realistic hardware architectures using temporal planners. *arXiv*: 1705.08927v2. https://doi.org/10.48550/arXiv.1705.08927

[52] Phillip B. Gibbons and Steven S. Muchnick. 1986. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the 1986 SigPlan Symposium on Compiler Construction,* June 25, 27, 1986, Palo Alto, CA, USA, 11-16. Retrieved March 13, 2022 from https://dl.acm.org/citation.cfm?id=13312

[53] Mohammad J. Dousti and Massoud Pedram. 2014. Minimizing the latency of quantum circuits during mapping to the ion-trap circuit fabric. *arXiv*: 1412.8003. https://doi.org/10.48550/arXiv.1412.8003

[54] Yunong Shi, Nelson Leung, Pranav Gokhale, Zane Rossi, David I. Schuster, Henry Hoffman, and Fred T. Chong. 2019. Optimized compilation of aggregated instructions for realistic quantum computers. *arXiv*:1902.01474. https://doi.org/10.48550/arXiv.1902.01474

[55] Kyle E.C. Booth, Minh Do, Christopher Beck, Eleanor Rieffel, Davide Venturelli, and Jeremy Frank. 2018. Comparing and integrating constraint programming and temporal

planning for quantum circuit compilation. *arXiv:* 1803.06775. https://doi.org/10.48550/arXiv.1803.06775

[56] Gian Guerreschi and Jongsoo Park. 2018. Two-step approach to scheduling quantum circuits. *arXiv:* 1708.00023. https://doi.org/10.48550/arXiv.1708.00023

[57] Jeffrey Booth Jr. 2012. Quantum compiler optimizations. *arXiv*:1206.3348v1. https://doi.org/10.48550/arXiv.1206.3348

[58] Tzvetan S. Metodi, Darshan D. Thaker, Andrew W. Cross, Frederic T. Chong, and Isaac
L. Chuang. 2006. Scheduling physical operations in a quantum information processor. In E.J. Donkor et al. (Eds.), *Proceedings of SPIE 6244: Quantum information and computation IV* (Vol. 6244), Defense and Security Symposium, 2006, Orlando, FL, USA. DOI: 10.1117/12.666419. Retrieved March 13, 2022 from http://feynman.mit.edu/ike/homepage/papers/spie06.pdf

[59] Prakash Murali, Norbert M. Linke, Margaret Martonosi, Ali Javadi Abhari, Nhung H. Nguyen, and Cinthia Huerta Alderete. 2019. Full-stack, real-system quantum computer studies. In *Proceeding of the 46th International Symposium on Computer Architecture*, June 22-26, 2019, Phoenix, AZ, USA, 527-40. https://doi.org/10.1145/3307650.3322273

[60] Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. 2018. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Inf* 4, 23. https://doi.org/10.1038/s41534-018-0072-4

[61] Ali Javadi Abhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. 2015. ScaffCC: scalable compilation and analysis of quantum programs. *arXiv*: 1507.01902. https://doi.org/10.48550/arXiv.1507.01902

[62] Damian S. Steiger, Thomas Häner, and Matthias Troyer. 2018. ProjectQ: An open source software framework for quantum computing. *arXiv:* 1612.08091v2. https://doi.org/10.48550/arXiv.1612.08091

[63] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open quantum assembly language. *arXiv*:1707.03429. https://doi.org/10.48550/arXiv.1707.03429

[64] Antti Autere. 2005. *Extensions and Applications of the A* Algorithm.* Helsinki University of Technology Laboratory for Theoretical Computer Research Reports 8, Helsinki, Finland. Retrieved March 22, 2022 from https://aaltodoc.aalto.fi/bitstream/handle/123456789/2647/isbn9512279487.pdf?sequence=1&isAllowed=y

[65] Brigitte S. Cypress. 2017. Rigor or reliability and validity in qualitative research: Perspectives, strategies, reconceptualization, and recommendations. *Dimens Crit Care Nurs* 36, 4, 253-263. DOI: 10.1097/DCC.0000000000000253. Retrieved March 22,

2022 from https://journals.lww.com/dccnjournal/fulltext/2017/07000/rigor_or_reliability_and_validity_in_qualitative.6.aspx

[66] Paul C. Price, Rajiv Jhangiani, I-Chant A. Chiang, Dana C. Leighton, and Carrie Cuttler. 2017. Reliability and validity of measurement. In Rajiv S. Jhangiani et al., *Research Methods in Psychology* (4th Ed.). KPU Press, Surrey, BC, Canada, Sect. 4.2. Retrieved March 13, 2022 from https://opentext.wsu.edu/carriecuttler/chapter/reliability-and-validity-of-measurement/

[67] Norbert M. Linke, Dmitri Maslov, Martin Roetteler, Shantanu Debnath, Caroline Figgatt, Kevin A. Landsman, Kenneth Wright, and Christopher Monroe. 2017. Experimental comparison of two quantum computing architectures. *arXiv*: 1702.01852v1. https://doi.org/10.48550/arXiv.1702.01852