TOWARDS A SCALABLE SOFTWARE STACK FOR RESOURCE ESTIMATION AND OPTIMIZATION IN GENERAL-PURPOSE QUANTUM COMPUTERS

ALI JAVADI-ABHARI

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
ELECTRICAL ENGINEERING
ADVISERS: MARGARET R. MARTONOSI AND FREDERIC T. CHONG

JUNE 2017
All rights reserved.
Abstract

Recent years have seen tremendous progress in quantum computing research, due in no small part to significant investment from major companies (Google, NASA, IBM, Intel, Microsoft) and governments (USA, EU, Japan, Australia). Small quantum computers have already been built, and medium-size computers with 100 - 1000 qubits are expected in the next few years. Building larger scale computers can have profound effects on the fundamentals of feasible computation, with major applications in cryptography, medicine and artificial intelligence. In contrast to mathematical algorithms and physical fabrication, quantum software tools have received relatively little attention. This is despite the fact that there is still a wide gap between the resource requirements of quantum applications (qubits and time) and feasible physical implementations. Good software tools can help bridge this gap. This thesis conducts an end-to-end study of system design issues for error-corrected, general-purpose quantum computers, focusing especially on design decisions arising in compilation, mapping, and scheduling stages. While drawing from decades of classical system design, it proposes novel techniques tailored to the specific constraints of the quantum domain, such as the central importance of error correction, problems of scalability, and alternative forms of data communication. By evaluating a wide range of potential applications, architectures and target technologies, valuable insights are obtained regarding design tradeoffs, guiding future developments on more efficient paths. Crucially, this thesis answers a question that could not have been accurately answered without the tools for realistic modeling of such systems: what is the best type of error correction for running a certain program on a given device? This question gains significant importance given that error correction is the leading source of cost overhead in highly fragile quantum systems. Various conditions are identified under which certain error correction codes are more efficient than others. It is shown that the rate of communication contention, which is directly related to application characteristics, can play a central role in determining the best method of communication and hence the type of error correction.
Acknowledgements

This thesis would not have been possible without the guidance of my advisors Margaret Martonosi and Fred Chong. Through countless hours of meetings and an abundance of feedback, they helped shape this work. Margaret’s considerations for the broader impacts of this research, as well as her emphasis on effective writing and presentation, were critical in every step of my Ph.D. Fred’s sharp intellect at identifying key issues and his clarity of vision were essential to my progress.

I benefited immensely from conversations with Ken Brown and Austin Fowler, who provided key advice on the theoretical aspects underlying this thesis. My appreciations are also due to members of my committee: Stephen Lyon, Diana Franklin and Niraj Jha. Steve was the first person who taught me about quantum computing, and I learned many things from him up until my last semester. Diana provided great help in writing the multiple papers which comprise this thesis.

I thank the many people with whom I collaborated during my Ph.D. Shruti Patil, Jeff Heckey, Adam Holmes, Pranav Gokhale, Daniel Kudrow and Alexey Lvov were especially instrumental as contributors and co-authors over the course of this work.

The year I spent as a visiting student at the University of Chicago was a time of great productivity, as well as personal fulfillment for me. I owe most of it to my dear friends Soroush Khaleghi, Sepideh Roghanchi and Zia Ghiasi, who opened their warm homes to me on more occasions than I can count.

I shared many pleasant moments with my fellow group members, past and present: Yavuz Yetim, Ozlem Bilgir Yetim, Daniel Lustig, Wenhao Jia, Logan Stafman, Themis Melissaris, Caroline Trippel, Yatin Manerkar, Tae Jun Ham and Elba Garza. Their feedback and insights helped improve this work.

Many thanks to Colleen Conrad and Roelie Abdi. They made life as a Ph.D. student much simpler, and the process of graduation much smoother.

Christiane Fellbaum enriched my last semester at Princeton by graciously accepting me in her class, and offering her time to indulge my interests in linguistics. She is one of the best teachers I have had, and has inspired me both intellectually and personally.

I thank my like-minded friends, as well as SPEAR and PTI, for creating some of the most meaningful moments of my time at Princeton. In particular, Jill Knapp, Hessam Akhlaghpour and Daniel Teehan — your impact on me extends beyond these years and beyond these words.

I am forever indebted to my parents for their sacrifice. My brother Mohammad and my sister Houra have been sources of comfort from far away. All of my love goes out to my new family: my wife Lindsey, and our little daughter. Lindsey supported me more than anybody else during these years, and for that I am eternally grateful.
To my parents.
## Contents

Abstract ................................................................. iii  
Acknowledgements .................................................... iv  

1 Introduction ......................................................... 1  
1.1 Motivations Behind this Thesis ................................. 2  
1.1.1 Why is Quantum Computing Important? .................... 2  
1.1.2 The Missing Pieces in Building a Quantum Computer .... 4  
1.2 Thesis Objective .................................................. 5  
1.3 Thesis Contributions ............................................. 5  
1.3.1 Comprehensive Study of Error Correction Implementation Tradeoffs 5  
1.3.2 Abstractions for Quantum-to-Classical Domain Translation ... 6  
1.3.3 Scalable End-to-End Toolflow ............................... 7  
1.3.4 Lessons in Quantum System Design ......................... 8  
1.4 Methodology and Toolflow ....................................... 9  
1.5 Outline of Thesis Chapters ...................................... 11  
1.6 Relation to Prior Work .......................................... 12  

2 Scaffold: Designing a Quantum Programming Language .......... 14  
2.1 Introduction ....................................................... 14  
2.2 Scaffold Design Overview ....................................... 14  
2.2.1 Language Design Goals ..................................... 15  
2.2.2 Scaffold Design Decisions ................................... 15  
2.2.3 Scaffold Program Example .................................... 16  
2.3 Scaffold Language Features ..................................... 17  
2.3.1 Data Types .................................................... 17  
2.3.2 Quantum Gates ............................................... 18  
2.3.3 Modules ...................................................... 18  
2.4 CTQG Modules: Using Classical Code in Scaffold ............ 19  
2.4.1 Motivation ................................................... 19  
2.4.2 Programming a CTQG Module ............................... 20  
2.5 Applications Written in Scaffold ................................ 21  
2.6 Related Work ..................................................... 23  
2.7 Future Extensions ............................................... 25  
2.7.1 Major Issues for Future QPL Design ....................... 25  
2.7.2 Possible Extensions to the Scaffold Language ............. 26  
2.8 Chapter Summary ................................................ 29
# ScaffCC: A Framework for Compilation and Analysis of Quantum Applications

## 3.1 Introduction

## 3.2 Internal Structure of ScaffCC

- 3.2.1 Compilation of Quantum vs. Classical Modules
- 3.2.2 Generating QASM-Language Output
- 3.2.3 Program Analysis

## 3.3 Managing Scalability Through Choice of QASM Format

- 3.3.1 Hierarchical QASM format (QASM-H)
- 3.3.2 Hierarchical QASM with Loops (QASM-HL)
- 3.3.3 QASM Code Size Comparison

## 3.4 Managing Scalability through Instrumentation-Based Compilation and Analysis

- 3.4.1 Dual Nature of Quantum Programs
- 3.4.2 Pass-Driven Approach
- 3.4.3 Instrumentation-Driven Approach
- 3.4.4 Compilation Speed Comparison

## 3.5 Compiling Classical CTQG Modules

## 3.6 Analyzing Scaffold Programs

- 3.6.1 Entanglement Analysis
- 3.6.2 Resource Analysis
- 3.6.3 Timing Analysis
- 3.6.4 Remodularization

## 3.7 Related Work

## 3.8 Future Extensions

- 3.8.1 Circuit-Level Optimizations Using Gate Identities
- 3.8.2 Decomposition in Terms of Other Gate Sets
- 3.8.3 Recycling Qubits and Reducing Circuit Footprints
- 3.8.4 More Intelligent Code Flattening

## 3.9 Chapter Summary

---

# Case Study: Characterizing the Effect of Trials and Rotations in QPE

## 4.1 Introduction

## 4.2 Quantum Phase Estimation

- 4.2.1 Quantum Rotation Decomposition

## 4.3 Implementations of Quantum Phase Estimation

- 4.3.1 Kitaev’s Hadamard Test with S gate (KHT)
- 4.3.2 QFT and Approximate QFT (AQFT)
- 4.3.3 Arbitrary Constant Precision Algorithm (ACPA)
- 4.3.4 Phase Kickback ($U_f$)

## 4.4 Analysis of the Design Space

- 4.4.1 Factors Affecting the Parallelization of Trials
- 4.4.2 Factors Affecting Circuit Size
- 4.4.3 Empirical Evaluation
- 4.4.4 Summarizing the Analyzed Tradeoffs
4.5 Case Studies ................................................................. 69
4.5.1 Case Study 1: Shor’s Algorithm .......................... 70
4.5.2 Case Study 2: The Ground State Estimation Algorithm ...... 71
4.6 Related Work ............................................................ 72
4.7 Chapter Summary ....................................................... 72

5 Compiler Management of Communication and Parallelism 74
5.1 Introduction ............................................................... 74
5.2 Multi-SIMD Architectural Model .................................. 75
  5.2.1 Motivation Behind SIMD-Style Computation .............. 76
  5.2.2 Adaptability to Different Technologies .................... 77
  5.2.3 Handling Data Movement and Teleportation ................ 77
  5.2.4 Optimizing Communication using Local Memory ........... 80
5.3 Scheduling Operations on Data ................................. 80
  5.3.1 Communication-Aware Scheduling with RCP ............... 81
  5.3.2 Communication-Aware Scheduling with LPFS .............. 81
  5.3.3 Hierarchical Scheduling ....................................... 84
  5.3.4 Scheduling Communication using Local Memory .......... 84
5.4 Results ................................................................. 86
  5.4.1 Characterizing Logical Parallelism ......................... 86
  5.4.2 Runtime Speedup with Data Movement Analysis ......... 87
  5.4.3 Runtime Speedup with Local Memories .................... 88
  5.4.4 Sensitivity to $d$ and $k$ parameters: .................... 90
5.5 Related Work .......................................................... 90
5.6 Chapter Summary ..................................................... 91

6 Quantum Error Correction: A Comparative Study of the Topological Surface Code and Concatenated Steane Code 92
6.1 Introduction ............................................................. 92
6.2 Device Technology ..................................................... 95
6.3 Quantum Error Correction (QEC) ............................... 95
  6.3.1 Concatenated Codes ........................................... 95
  6.3.2 Surface Codes ................................................. 98
  6.3.3 Comparing QEC Codes ...................................... 98
6.4 Methodology and Toolflow ....................................... 99
6.5 Optimized Teleportation ......................................... 100
  6.5.1 Effectiveness of a Multi-SIMD Architecture ............. 100
  6.5.2 Smoothing Ancilla Distribution ............................ 102
6.6 Optimized Braiding ............................................... 102
  6.6.1 Braid Simulation using Message Passing ................ 102
  6.6.2 Optimizing Qubit Arrangement ............................. 105
  6.6.3 Optimized Braid Priorities ................................ 105
6.7 Comparing QEC Overheads .................................... 107
  6.7.1 Finding Regions of QEC Favorability ...................... 107
  6.7.2 Sensitivity to Application Characteristics ............... 108
Chapter 1

Introduction

Quantum computing is at the cusp of a revolution. Machines with 100 quantum bits (qubits) are under development [62, 168] and 1000 qubits are around the corner\(^1\). This march towards reality must be accompanied by software tools that can help speed up the development of better applications, shed light on opportunities for optimization that may not be obvious from purely algorithmic or fabrication perspectives, and help anticipate roadblocks in near- and long-term implementations.

In addition to helping accelerate the roadmap to quantum computation, better software tools will also play a central role in deciding the best type of design among many available options. Despite many proposals, from various physical technologies to different forms of error correction, it is still unclear which combination of design choices is suitable for a given goal and resource budget. Any design must aim to minimize the cost in terms of time and space, currently the main factors prohibiting the realization of large application on an actual quantum computer. An integrated and automated approach to the translation and simulation of high-level algorithms on low-level technological implementations is the only way to compare various designs and choose the best.

This thesis is aimed at answering some of the aforementioned questions related to best design practices in future quantum machines. Through a combination of programming language techniques, compilation methods, architecture design and simulation infrastructures, it investigates the resource overheads associated with implementing a wide ranging array of quantum computing applications. Throughout this thesis, we use resources to indicate both space and time resources, meaning the number of qubits necessary and the time it takes to execute. On one hand, we make observations that were previously unknown: for example, that communication congestion will favor a particular form of error correction. On the other hand, the set of softwares developed as part of this thesis can serve as valuable tools for future research.

In this introductory chapter, we aim to define the motivating factors, questions, scope, methodology and contributions of this thesis. The reader unfamiliar with quantum computation may benefit from reading Appendix A, which presents a brief background that covers the basic necessary concepts in quantum computing to allow understanding of the rest of the thesis.

\(^1\)Throughout this thesis, our work is focused on general-purpose quantum computing. Specialized processors such as quantum annealers [86] have been proposed with more than 1000 qubits [1].
1.1 Motivations Behind this Thesis

1.1.1 Why is Quantum Computing Important?

End of Moore’s Law Approaches

Moore’s Law [129], the observation that the number of transistors in an integrated circuit can be doubled approximately every two years, has remarkably proven to be true for over 50 years. This fast growth rate has given birth to the digital revolution and the ubiquity of low-cost and powerful computers.

Unfortunately, it now seems that the end of Moore’s Law is approaching [158]. There are two reasons for this: economically, the cost of super dense transistor placements may soon have diminishing returns. Technologically, certain physical phenomena in nature will severely impede the prospect of successfully deploying tiny transistors. These are typically known as quantum tunneling effects: particles may penetrate a “wall” of very thin material (transistors in this case), whereas in a classical world this would have been impossible. This illustrates the difficulty of controlling extremely small systems — unpredictable quantum effects will start to cause unforeseen problems in the design of such computers.

In its current roadmap, Intel is due to fabricate and ship 10 nm semiconductor chips in 2017. However, the next few years will see progressively more difficult scaling as quantum effects start manifesting themselves in 5-7 nm processes [42]. Clearly, there is not much time left before Moore’s Law will encounter a wall from quantum mechanics.

Interestingly, while quantum mechanical effects pose problems in extremely small classical chip designs, they open up entirely new paradigms of computing. Quantum Computing is the practice of exploiting quantum mechanical objects as carriers of information and quantum state manipulations as means of doing computation.

One simple example is Adiabatic quantum computing proposed by Farhi et. al. [57]. In this new paradigm of computing, it is possible to map certain difficult optimization problems onto the state of materials that behave quantum mechanically (e.g. atoms or electrons, which are not bound by laws of classical mechanics). Since matter tends to relax to its base ground energy state, we can leverage this natural relaxation to find the optimized answer. Natural state evolutions would yield the minimum of a function as dictated by its ground energy level.

Quantum computing, therefore, through a shift in computation paradigm, allows the creation of computers that are not bound by semiconductor scaling limitations and can be used in solving several currently intractable problems.

Theoretical Implications on Computability

Another reason to pursue quantum computation is simply due to the profound implications that a real quantum computer will have on our understandings of the limits of computability. The class of problems that can be solved efficiently if we had access to a quantum computer is known as the BQP (Bounded-error Quantum Polynomial-time) complexity class, and is conjectured to be different from the P (Polynomial-time) or even the BPP (Bounded-error Probabilistic Polynomial-time) class that can be solved efficiently on classical computers. If this is the case, quantum computers may be inherently more powerful.
than classical computers: BQP may be found to be a strict superset of BPP, challenging
the strong Church-Turing thesis [40, 184]. The success or failure of efforts to build an
actual quantum computer that can demonstrate quantum supremacy (solving classically
intractable problems) is tightly tied to this notion, fundamentally defining the concept of
computability in the natural world. For this reason, regardless of its practical implications,
success (or failure) at building a quantum computer is a goal worth pursuing.

Richard Feynman’s original proposal for building a quantum computer is a good il-
lustration of possible quantum supremacy [58]. He observed that classical simulation of
nature as we know it (quantum mechanical) is impractical for even a modest-sized physical
system. This is due to the exponential growth of the state space of qubits as they grow
in numbers. For example, currently the most powerful supercomputer in the world, the
Sunway TaihuLight, has a memory of 1.31 petabytes and a clock rate of 93 petaFLOPS.
However, it can only hold the state of 47 qubit in memory. What if, then, we wanted to
simulate a molecule containing 270 particles (or qubits)? On a classical computer, this
would require $2^{270}$ bytes of memory, which is close to the number of atoms in the observ-
able universe. This fundamental disparity between the natural world and classical models
of computing led Feynman to propose building a quantum computer — using quantum
mechanical objects to simulate quantum mechanics itself.

However, this quantum-over-classical advantage is an asymptotic one, and will only
manifest itself in large problem instances. Therefore, one of our main goals is to study
quantum computing “at scale.”

Practical Applications

It is certainly helpful that certain problems of high practical interest have been proven to be
in BQP, greatly accelerating the research in quantum computing. Quantum computing is the
only method we know that may be able to exponentially speed up certain important prob-
lems, which would otherwise remain intractable. Integer factorization (related to breaking
RSA cryptography [153]), discrete logarithm (also related to cryptography), and simulating
quantum mechanical behavior (as discussed above) are shown to be in BQP [28, 165].

Factoring a $b$-bit number using the best classical algorithm known to date (Number
Field Sieve by John Pollard [113]) has a complexity of $O(exp(\sqrt[64]{\frac{b}{9}}b/log(b)^2))$. In contrast,
Peter Shor’s quantum factoring algorithm is polynomial-time $O(b^3)$ [165]. This is remark-
able, as it means that finding the prime factors of any number on a quantum computer is
as fast as multiplying those primes. This destroys RSA cryptography, which is fundamen-
tally based on the hardness of the factoring task (a “1-way function”). Quantum computing
can thus pose challenges to the many software [133, 153] and hardware [91, 92, 156, 178]
security protocols that are designed to protect today’s computers.

Quantum computers’ ability to invert hard functions also has applications in solving
systems of linear equations [71]. Furthermore, being able to efficiently simulate large
quantum mechanical systems has implications for drug design and protein folding [88].
Such practical applications are the driving force behind the surge of quantum computing
research in recent years.
1.1.2 The Missing Pieces in Building a Quantum Computer

The fact is that many of the hardest obstacles in making quantum computing a reality lie in fabrication laboratories, where the vision of large scale computers are hindered by experimental difficulties in controlling large numbers of qubits and shielding them from the surrounding environment. However, the question of viable quantum computing for large, interesting problems also depends to a great extent on the full system design. Research into a comprehensive and dependable software stack is required at this moment to guide architectural and organizational decisions, which in turn impact the vision of device characteristics to be developed. It is also important for the software stack to start maturing now in order to avoid imposing further delays in the future. Below we discuss some of the benefits of developing such system-level understandings.

Choosing the Right Design Points

The choices for implementing a quantum computer are many. The research community has still not come to a firm consensus regarding the exact choice of technology to act as the basis for a future device, the sort of error correction that it should run, its specific computer architecture, or the type of applications that it can support with a limited resource budget. Active research in fabricating qubits and manipulating them is being pursued using many different technologies: Trapped ions [41], superconductors [130], quantum dots [117] and nuclear magnetic resonance (NMR) [44] to name a few. Similarly, several different quantum error correction codes are candidates for a future quantum computer: Steane [170] Bacon-Shor [14, 166], defect-based surface [60], planar surface [33, 48], lattice surgery [76], and color codes [104] are some of the proposed ideas.

A valid comparison of the different design decisions is only possible through a comprehensive system-level modeling that involves all the possible aspects of such combinations. Furthermore, resource and scaling information from realistically modeled large systems can guide device technology development, as well as provide algorithm developers with better tools with which to implement and develop quantum applications.

Dominance of Error Correction

The most important reason why even modest quantum applications are hard to implement with current technology is the heavy need for error correction. Quantum states are especially susceptible to corruption due to a variety of noise, imprecise operations, or couplings with the environment. This lack of accuracy stems from the analog nature of quantum states, whereby rather than susceptibility to bit flips alone, they may experience countless digressions from the intended state. Fortunately, quantum error correction can be a defense. However, it is extremely costly: one ideal error-free operation may take a combination of thousands of realistic noisy operations to implement.

Program runtimes play a crucial role in the success of error correction. The overall runtime must be substantially optimized for error correction to be useful: longer program runtimes invite more error accumulation, imposing more need for expensive error correction. A mathematical description of a quantum algorithm cannot accurately describe the
runtime of the actual implementation of that program. This is due to the fact that algorithm specifications almost always ignore error. Furthermore, they ignore the realities of the computer. For example, if parallel operations were not possible, errors would accumulate too quickly for error correction to cope. It is therefore exceedingly important to use a full software stack to accurately model program runtimes, and optimize them when possible. Only then can we have an accurate estimate of the implementation cost, and know that error correction is not consuming more resources than necessary as a result of runtime inefficiencies.

1.2 Thesis Objective

The main objective of this thesis is to study the resource overheads of a future quantum computer at large scales. The reason is that the lower complexities of quantum algorithms are only useful when classical executions fail — at very large scales. In order to do a comprehensive study, it is crucial to take into account a variety of quantum applications, quantum error correcting codes, and quantum technologies; because while the theoretical framework behind each of them is maturing, an integrated comparison of the benefits of combining them in various forms remains lacking.

Additionally, we aim to understand the extent to which software optimizations at various levels of the translation process, from compilers to architectures and schedulers, can speed up the execution of quantum workloads. This speedup is important: noise-related errors in quantum computation are high, and shortening the time of computation can mitigate error accumulation, drastically improving the cost of error correction and thus the overall implementation.

Finally, we aim to arrive at a comprehensive and robust set of tools for an algorithm-to-technology-level mapping. Such tools will empower researchers to investigate a wide range of questions pertaining to quantum algorithms and their implementation. For example, we apply these tools to the investigation of three different methods for implementing Quantum Phase Estimation, a particularly ubiquitous module in quantum applications. We are able to empirically characterize various workloads based on the different implementations, and observe their tradeoffs in terms of time and space.

1.3 Thesis Contributions

1.3.1 Comprehensive Study of Error Correction Implementation Tradeoffs

A principal contribution of this thesis is its comparative study of various forms of error correction, and their tradeoffs under different implementations. We study the concatenated and surface code families, as the main candidates proposed for protecting quantum states. Among our most important observations are:

1. Surface codes are better when strong error correction is needed (i.e. when there is a wide gap between the reliability rate requested by the application and the reliability
rate provided by the physical technology — this could be either due to a very faulty device, or a very large computation that needs to sustain fidelity over a long period of time). In the opposite case, concatenated codes are better.

2. The aforementioned observation only describes approximate regions of favorability. This thesis also quantifies, for the first time, the exact transition boundary between the favorability of surface codes versus concatenated codes. This boundary is very similar in most applications. However, in highly-parallel applications the boundary shifts in favor of concatenated codes. This is due to two factors: first, the bitwise nature of computation in the concatenated code allows leveraging data- and operation-parallelism which favors more parallel applications. Second, movement of data in concatenated codes can be based on teleportation, a quantum-specific form of communication that decouples the physical motion of qubits from the actual communication of data, thus allowing for an intelligent scheduling of qubit motion to reduce conflicts. No such decoupling exists for communications in the surface code, which suffer from more conflicts in highly-parallel applications.

3. We observe that in addition to application characteristics, technology characteristics also affect the choice of the best error correction. Our experiments show that the aforementioned transition boundary is sensitive to the relative speed of operations, which are constantly evolving in experimental setups and differ in different technologies. We quantify how variations in operation latencies (specifically in single-qubit, double-qubit, and measurement operations) affects the favorability of one form of error correction over the other.

### 1.3.2 Abstractions for Quantum-to-Classical Domain Translation

This thesis identifies abstractions to approximate and translate from the quantum domain to the classical domain, and then solve them using known classical techniques. Some examples of this are as follows:

1. We have identified a new method for treating communications on the surface codes, which is much more scalable than previous hand-optimized approaches yet achieves performance results close to the critical path in most cases. These communications, known as braiding, have been traditionally done using manual compressions of 3D topologies [50, 61]. By approximating braids as messages in a classical network, and using the distinct property of $n$-hop/cycle travel for braids, we are able to avoid cycle-by-cycle simulations and quickly lay out the braids in a circuit-switched manner. Due to the serial nature of most applications, we found this to sacrifice only small amounts of performance as compared to optimal scheduling.

2. Although some principles of classical computer architecture and compiler research apply, some important differences necessitate new considerations. For example, the physical difficulty of controlling many different qubits necessitates the simplification of control circuitry; SIMD computation is therefore a suitable choice. This, coupled
with the mostly serial nature of the algorithms we have studied (all but two of the algorithms we evaluated are more than 60% serial) led us to find that qubit movements are the dominant cost. We propose a communication-centric scheduling approach to minimize this cost, by targeting both increased parallelism and increased locality.

3. Ancilla qubits are a new phenomenon, distinct to QC (and reversible computing [5]). They are temporary qubits designated as facilitators of specific tasks, such as communication of qubits or performing certain “difficult” gates. Ancillas are therefore distinct from data; whereas data usually persists during a long computation, a host of ancillas are frequently created, used, and recycled at different stages of that computation. Keeping the number of ancillas reasonable is therefore an important goal. We propose ancilla management strategies to manage different tradeoffs: Greedy ancilla distribution has the best time, but worst bandwidth usage. On the other hand, smoothed-out ancilla distribution using simple predictive windows achieves minimal bandwidth and simultaneous qubit usage, while sacrificing modest time overheads.

1.3.3 Scalable End-to-End Toolflow

This thesis proposes the first scalable end-to-end toolflow from quantum algorithms to technology-specific quantum architectures [3]. This allows for automated design and evaluation. Some of the insights and benefits that can be gained from such end-to-end tool design are as follows:

1. There is yet no firm consensus in the community on the exact choice of qubit technology and error correction scheme for the next generation of quantum computers. Automated design tools are essential in evaluating the tradeoffs and navigating the design space in accordance with specific resource budgets or available device fidelities.

2. It is easy to extend integrated toolflows in the future, as our knowledge evolves on a host of related topics, from available optimizations to new fault-tolerant design protocols. As an illustrating example, we were able to extend our infrastructure to evaluate a question which had not been fully answered: how to implement surface codes using trapped-ion technology, and how it would compare in terms of time and area resources to other technologies or other error correction methods. The modular design and abstraction layers of the software allow for drop-in replacements of various technologies, architectures, schedulers, error correction codes, and compiler optimizations.

3. Vertical integration allows for better management of scalability, and better iterative design. Using multiple optimization heuristics, we were able to reduce the compilation and analysis time of our infrastructure to handle problems of up to $10^{20}$ physical gates — problem sizes which ultimately fall in the desired range if we are to take over classical computing. Furthermore, any real design will likely be iterative, and an end-to-end tool with feedback capabilities will be immensely beneficial. As an example, the degree of program modularity which is decided by the compiler and
implemented through module flattening, can be guided and modified by later lower-level discoveries. This was found to be useful when, for example, we found a certain optimization on ancilla qubit management to work well only if a certain degree of flattening had been applied at higher levels.

4. Quantum computing can often be quite different from classical computing. This is manifested in how qubits behave (lack of data fanout, disallowing of copies), how communication occurs (difficulty of long-range communication, constant latency over arbitrary distances), and what design aspects are the bottlenecks (error correction is by far the most dominant component). In that regard, even though we can draw valuable lessons from classical designs, our intuitions may or may not always apply. Software tools are important for guiding design decisions. Our work tries to determine where designs should be similar to classical computers, and where they should differ.

### 1.3.4 Lessons in Quantum System Design

This thesis identifies some important differences in system design for classical computation versus quantum computation, which could be drawn upon for guidance in future quantum system research endeavors:

1. Quantum circuits, unlike classical circuits, are almost always specialized to a certain concrete problem with known inputs (i.e. factoring a specific 1024-bit number). This static availability of circuit information allows for deep analysis. This is important to allow for optimality and to reduce precious quantum resources. In general, resources such as qubits are much more scarce in the quantum domain, and specializations such as this are imperative to keep overheads low.

2. With deep analysis come problems of scalability. Here again unique quantum properties can be leveraged to better manage issues of scale:

   (a) The classical/quantum dual-nature of quantum programs yields itself well to an instrumentation-driven approach similar to dynamic interpretation, which scales better in compilation.

   (b) Regularities of some quantum circuit subroutines, such as error correction subroutines and state preparation subroutines, allow for abstractions that mitigate scalability costs. For example the preparation circuitry of various ancillas, although different internally, all have the same interfaces: a steady supply of prepared qubits to the interconnection network. This common case can be exploited to keep simulations of quantum workloads manageable.

   (c) The unique difference between data and ancilla qubits allows separate scheduling, where the dominant ancilla qubits can be taken off the scheduler critical path, and be optimized separately in a more scalable fashion, due to their complete lack of data dependency.
(d) The capacity of mapped registers can be variable, and is limited by the fanout capabilities of the microwave or voltage multiplexer. Unlike classical computing, it is not desirable to unify register lengths, as this would have an adverse effect on available parallelism.

1.4 Methodology and Toolflow

This section describes software stack and the methodology we employ. We begin by formulating in greater detail the main questions of the thesis, and relate them to the necessity of designing a robust toolflow in support of experiments that answer those questions. Finally, we provide an overview of the stages involved in this software stack. We see that these stages are closely related to the outline of the remaining chapters, each of which highlights in detail one aspect of the overall toolflow.

As mentioned, high resource consumption is the most important challenge in quantum computing. Designing a successful quantum computer will require a dependable software stack that is able to quickly analyze and offer insights into a wide range of design parameters, to help choose the lowest-overhead architectures, error correction codes, and technologies under various constraints. In this section we outline the set of software tools that underpin this thesis, and are necessary to make progress towards its central question: how to characterize and optimize the resource requirements of implementing a future quantum computer, and what the tradeoffs are in the various design choices available to researchers today.

One fundamental question that currently lacks a clear answer is the choice of error correction. Error correction is the single most resource-consuming aspect of a general-purpose quantum machine, having a higher overhead than the computation itself. It is therefore crucial that in a given scenario, the right error correction type is used from the many available choices — one with the lowest overhead. Furthermore, we do not want to pay for error correction more than it is absolutely necessary. Therefore, even after choosing the most efficient error correcting code, we still want our program to complete in the least amount of time possible. Longer running times contribute to more noise, necessitating more error correction.

In our methodology, we achieve the goal of choosing the best QEC code in two phases: first, a compilation and scheduling phase that is able to reduce the overall program runtime to a minimum (akin to optimizing the logical quantum circuit); second, through accurate simulations of many possible design points (combinations of architecture, QEC code and technology) in order to find the best one.

It is clear a comprehensive and fair comparison of the different designs is dependent upon an automated optimization and evaluation process. Figure 1.1 shows this “software dependency” chart in detail.

Part A at the top of Figure 1.1 is a simple view of what we hope to achieve. It is a graph that decides which kind of error correction (among the main contenders, concatenated and surface codes) is suitable for running a specific application on a specific machine. This is specified by a single coordinate on the plane. The horizontal axis’s value is indicative of the available error rate from the device. The vertical axis’s value is proportional to the
Figure 1.1: Software dependency chart to achieve the goals of this thesis, which are to draw tradeoff graphs similar to (A). A series of simulation (B), physical space-time optimization (C), logical communication reduction (D), and compilation and code analysis tools (E) tools are developed.
size of the application (or inversely proportional to the applications expected error rate). Therefore, specific \((x, y)\) value pairs indicate the strength of needed error correction — the amount of redundancy that must be added to fill the gap between physical and logical realms. Researchers have known for some time that concatenated codes fare better for low-strength error correction (bottom left — where the gap between physical and logical error are not huge), and surface codes are more efficient for high-strength error correction (top right). What is not known however is the transition boundary between the two, and whether this depends on characteristics of the application or of the technology in question. These questions are answered in Chapter 6.

Part B, therefore, is a simulation framework designed to estimate realistic overheads for all combinations of technologies and error correction codes, for any given application. Each combination is amenable to a particular computation and communication model, elaborated on in Chapter 6.

In order to obtain a fair estimate, we should compare optimized resource consumptions from each scenario. Parts C and D are respectively designed to reduce the overhead from error correction itself (physical overhead) and from expensive communication among data (logical overhead). These are elaborated on in Chapters 5 and 6.

Finally, the point of entry must be a compilation and analysis framework that can supply the code, its data-dependency graphs, its communication patterns, and its parallelism potential to further optimization and simulation stages. It must also be scalable, as we aim to investigate very large programs, in the range suitable for beating classical algorithms. Part E is dedicated to this purpose, and is elaborated on in Chapter 3.

1.5 Outline of Thesis Chapters

The rest of this thesis is organized into the following chapters. Figure 1.2 illustrates how each chapter fits in the overall flow of mapping quantum algorithms to physical hardware.

---

**Figure 1.2:** Layout of chapters in this thesis, and their relation to the overall toolflow.

Chapter 2 is our point of entry in the toolflow. It presents Scaffold, our approach to designing an easy-to-use and adaptable quantum programming language. In addition, it
discusses several important applications developed in Scaffold, which form the basis of our experiments throughout this thesis. In this chapter we will also discuss various other QPL proposals that have recently been published.

Chapter 3 discusses compilation. We discuss the opportunities and challenges of designing a compiler for large scale applications, the scale at which quantum may overtake classical computation in speed. We discuss the unique properties of quantum workloads which inform the compiler design, and how the compilation process as well as the target language must be tailored to adapt to the demands of scalability. This work is the basis for ScaffCC, an open-source toolbox for the compilation and analysis of quantum applications.

Chapter 4 is a comprehensive case study in the utilities of the language and compiler discussed in its preceding chapters. We will characterize several use cases of the Quantum Phase Estimation module (a ubiquitous module in quantum algorithms), and discuss the space and time tradeoffs in each scenario.

Chapter 5 discusses an architecture for leveraging parallelism and simplifying control in quantum computers. It also illustrates the importance of communication as the highest source of latency in application runtimes. Communication-aware schedulers are thus designed to leverage the architecture in an efficient manner.

Chapter 6 ties together all the components of our toolflow by proposing a mapping and simulation backend for several important variations of quantum error correction codes and physical technologies. Through exhaustive simulations it quantifies exact tradeoff regions where different design points become advantageous.

Finally, Chapter 7 offers conclusions and possible future directions of research.

1.6 Relation to Prior Work

In this section, we present a broad overview of major efforts in quantum system design up to this point. The intention is to touch upon major themes and progresses made in quantum software research, and further establish the frontiers of novelty in this thesis. More detailed discussions of related work pertaining to specific topics can be found in the relevant chapters.

Svore et. al. [175] proposed one of the earliest designs of a full stack quantum software architecture, focusing mainly on the representation of the code at each stage: program-level, circuit-level and machine-level. Jones et. al. [84] have formalized this layered approach with more explicit definitions for the roles of each layer, and the protocols for inter-layer communication. Each of these approaches target a specific form of error correction or a specific technology. We have sought to unify these: by making suitable abstractions, we have created application, QEC and technology layers that can accommodate other variations within their own family. This allows comparisons of different design points, a central aim of this thesis. We further emphasize communication as an important source of overhead, and seek to minimize it at both the logical and the physical levels.

A number of previous works have looked at the cost of implementing one particular type of error correction on one particular technology [60,84,108]. The first comprehensive comparison of the resource overhead associated with different choices of quantum applications, QEC code, and quantum technology was performed by Suchara et. al. [174]. Their
study confirms the theoretical predictions for better scalability of surface codes. Building
on this work, the current thesis performs a much more rigorous study and makes major
new observations, especially pertaining to the sensitivity of good QEC choice to various
application and device characteristics. These observations augment the understanding of
the research community (for example with regards to the suitability of concatenated codes
in parallel applications). Contrary to [174], this thesis studies various problem sizes of a
particular application to quantify the QEC boundary of favorability, which forms the basis
for our most novel observations.

In terms of methodology, modeling communication overhead is a major part of this
thesis. By having a detailed view of the network in both braiding and teleportation com-
munications and considering conflicts in data motion, we propose methods to both reduce
logical communication requirements (i.e. LPFS scheduling, tile placement optimization),
and to reduce the physical communication overheads (i.e. EPR smoothing, braid conflict
resolutions). In [174], this is simplified to only account for parallelism factors, and it is
assumed that all logically parallel operations may be performed simultaneously.

Compilation and mapping of quantum programs is a central part of this thesis. The case
for an integrated compilation toolflow was made in [175] and made more mature with ex-
plicitly designed interfaces in [84]. Our abstraction layers closely mimic those proposed in
these works, with particular emphasis on the intermediary layer for communication reduc-
tion, data-flow analysis to exploit parallelism, and pipelining to reduce spatial and temporal
overheads. More extensive high-level programming language designs have been investi-
gated in [68, 189]. Lin et. al. [114, 115] proposes gate libraries suitable for substitutions at
the circuit-level of compilation.

An important part of this thesis is the reduction of total logical communication distances
by employing multiple architectural techniques. Previous architectures have similarly been
proposed for parallelism and compaction [124,182]. We further employ Longest-Path-First
Scheduling and local scratchpad memories to create an architecture with reduced commu-
nications requirements.

Pipelining ancilla preparation and teleportation latencies in the cycles of computation
has had a major impact in reducing overall time overheads in our study. This was first
proposed in [79]. In it the authors evaluate the area requirements to prepare ancilla and do
error correction “at the speed of data.” We employ the same ideas of ancilla factory pipelin-
ing, but do not assume that the pre-distribution of ancilla can occur without penalty; instead
we simulate the conflicts inherent in this distribution and propose methods to smooth the
communication requirements in the computation cycles.

Our toolflow ends with a simulated model of the technology based on their abstract
physical parameters such as speed or error rate. It is, however, beneficial to augment
the backend with actual pulse sequence generations for a realistic quantum machine. The
IBM Quantum Experience [2], based on superconducting technology, is currently the only
openly available quantum computer on which to run experiments. [172] has recently in-
troduced a toolflow that plugs into this backend, lowering high-level quantum algorithm
specifications (in Python) to circuits executable by the IBM machine.
Chapter 2

Scaffold: Objectives and Challenges in Quantum Programming Language Design

2.1 Introduction

The first step in gaining insight into a quantum algorithm’s implementation cost and possible optimizations is to express that algorithm in a programming language, to be interpreted in an automated manner. This chapter describes a proposed quantum programming language, Scaffold. We explain our design philosophy, and through multiple examples illustrate how this language can be used to express a host of quantum algorithms. We further discuss some of the most important issues that designers of quantum programming languages must consider.

A distinction must be drawn between high-level programming languages and low-level compilation targets. Scaffold is designed as a high-level language to intuitively describe quantum programs, and constructs that naturally arise when describing a quantum circuit. On the other hand, after compilation, Scaffold is converted into a lower-level language called quantum assembly or QASM, as proposed in [132, 175]. The QASM language describes quantum programs using a set of low-level standard gates. QASM is independent of the underlying quantum technology — any technology can in principle implement the low-level gates described by QASM, albeit in different forms and with different latencies. Grammars for Scaffold and QASM may be found in Appendices B and C respectively.

2.2 Scaffold Design Overview

Scaffold is a programming language for expressing quantum algorithms. A quantum algorithm can consist of a wide variety of components (including classical and quantum routines) which must be defined using different coding techniques. As a quantum programming language (QPL), Scaffold was formulated to make it easy to express an algorithm

---

1 The work in this chapter was performed in part in collaboration with Arvin Faruque and others [80].
with so many disparate components in a clean and efficient manner. It is from this notion of “putting things together” that Scaffold derives its name.

2.2.1 Language Design Goals

As mentioned, Scaffold is designed to facilitate expressing quantum algorithms in terms of operations and data types that can be compiled down to a QASM target representation, and for which resource estimation and other analyses can be expeditiously applied.

Towards this goal, the expressiveness of the language and its ease of use are primary factors. Surveying various quantum algorithms, we find that they often express arbitrary angles of rotations, without regard to the target technological capabilities. Many express computation in terms of quantum gates, while some include a *quantum oracle* — a black box transformation of quantum state for answering certain queries — which are usually expressed as classical math functions. Some use repeat-until constructs to achieve desired degrees of answer accuracy, and a few use recursion. In all of these cases, the language must be expressive enough to allow programmers to easily achieve the implementation task.

On the other hand, a quantum programming language is likely to be used primarily by theoreticians, who would want to test the performance of a particular algorithm, or by physicists, who would like to understand the resource consumption of an algorithm. Users familiar with basic quantum computing and classical computer programming concepts should be able to learn and use the language smoothly. This will be facilitated if the syntax is close to ubiquitous languages in the science and engineering communities, such as C. This has another important advantage, which is the ability to leverage existing and mature compilers for classical languages, and being able to extend them to accommodate the new domain.

2.2.2 Scaffold Design Decisions

Towards the realization of the design goals listed above, the Scaffold language is designed around an imperative programming model using C as a host language, with additional data types, instructions, and the ability to mix classical and quantum code.

The imperative design is explicitly preferred to increase ease of use. The applications Scaffold targets often have very natural expression in terms of loop nests, iterative calculations, and function calls with argument passing. Around any quantum calculation lies a framework of classical computation control, and for these it is often preferred to have familiar variable and control structures. Languages that influenced the design of Scaffold included popular classical high-level imperative programming languages (C/C++) [90, 173], hardware description languages (Verilog) [77], C-to-hardware languages (System-C) [78] and existing quantum programming languages (QCL) [135].

While Scaffold is a variant of C for familiarity’s sake, we recognize that some of its features—e.g., pointers—are of modest use in our application scenarios, and yet may be used in ways that introduce bugs. To mitigate these issues, Scaffold minimizes the legal use of pointers (no pointer arithmetic or arbitrary dereferencing) and provides aggressive
module main ( ) {
  int i=0;
  qbit control[4];
  qbit target[4];
  for(i=0; i<4; i++) {
    CNOT(control[i],target[i]);
  }
}

Figure 2.1: Example: A very simple Scaffold program.

syntax and out-of-bounds checking at compile time. In this way, we benefit from C’s straightforward familiarity, without paying for it in pointer and addressing errors.

A key feature of Scaffold is a Classical code to Quantum Gates sequence (CTQG) module. CTQG modules allow a programmer to express a desired quantum functionality using classical programming. Rather than viewing everything in terms of quantum gates, CTQG modules allow programmers to describe the functionality of some parts of the algorithm from a higher perspective. This is somewhat like hardware description languages, where this classical code is then converted into a sequence of gates. The inclusion of this feature greatly increases the ease of expression, and is used in several of our application implementations. Furthermore, CTQG modules come into play when allowing the expression of whole quantum applications, since many existing algorithms in the literature specify their code in terms of oracles that are typically consisting of imperative code.

Because Scaffold bears such similarity to C, it has the benefit that a high-level Scaffold compiler can use a high-level C compiler as its starting point. In particular, Chapter 3 shows how the LLVM compiler infrastructure [110] is used for much of the parsing, intermediate representation, and back-end. The language design decisions greatly facilitate leveraging existing tools, since we can build off of previous efforts in parsing, optimization and specially reversible logic synthesis. Because of its resemblance to C we can take advantage of existing front-ends such as Clang, a front-end to LLVM for parsing the C language family. We modify parts of Clang’s libraries to accommodate the new language features of Scaffold in addition to C.

2.2.3 Scaffold Program Example

Figure 2.1 begins our discussion by illustrating a very simple Scaffold program. Similar to the familiar “hello world” examples, this program has very little functionality, but is intended to show the simple beginnings of Scaffold programming and to illustrate a few key concepts.

Line 1 begins the definition of the main module. Every free-standing program must contain a module called main, which tells the compiler the entry point of compilation. Lines 2-4 define several program variables of different types. Scaffold programs support both classical and quantum variables. In this case, lines 3 and 4 each define a register comprised of 4 quantum bits. Lines 5-7 are a very simple loop construct. The loop body—
in line 6—computes a CNOT in which the i-th qubit in control is CNOT-ed with the i-th qubit in target. Because each iteration of this loop is independent (there are no loop-carried dependencies) the iterations could be executed in parallel (analyzed by the compiler dependency checks).

Scaffold lets the user specify when and where quantum operations get applied (sometimes based on classical conditions). It does not, however, force the user to dictate what happens inside a quantum operation (i.e. the actual unitary matrix of the gate). Such functionality is applied in a later technology-mapping stage. This is akin to the idea that a classical computer has a known instruction set at the assembly language level, and does not require each programmer to describe the specific functionality of an add or multiply instruction set. The Scaffold compiler does not check the functionality of each requested gate, but it does enforce rules that ensure the gates are applied in a valid quantum mechanical manner. For example, it will not allow the user to feed the same qubit into both the control and target ports of a CNOT gate.

2.3 Scaffold Language Features

2.3.1 Data Types

Qubits and Quantum Registers

The main data type in Scaffold is the qubit, represented as qbit. Quantum registers are defined in Scaffold for grouping individual qubits together. They can be defined and initialized as follows, where \( n \) is the size of the quantum register, and example_reg is the name.

```c
qbit example_reg[n];
```

Note that \( n \) is a constant number. This is an important feature of quantum programs—they specify a fixed circuit with fixed size and known inputs. We shall see later how this aids optimization. The important thing is that all inputs must be known at compile time, either by being declared as const, or by being determinable through constant propagation from other constants.

After the quantum register has been declared, a particular \( m \)th qubit within this qreg can be accessed using

```c
example_reg[m]
```

Similar to C, the qubits are numbered starting from 0. Since quantum registers store qubits which must abide by the no-cloning theorem, call-by-reference is used when passing them as arguments to either gates or modules.

Scaffold also defines a new data type cbit, which is a classical bit (C by default does not have boolean variables). A cbit is typically used when a qbit is measured, and thus collapses into a classical state.
Classical Variables

It is possible for a Scaffold programmer to define classical variables and pass them as parameters in the code. A classical variable is defined using the following notation:

```
datatype variablename;
```

where `datatype` is the type of the variable and `variablename` is the name of the variable. Valid primitive datatypes include the same datatypes commonly used in C: char, int, long, float, double, signed, unsigned.

Valid casts between datatypes can be done using the C casting notation. These types, however, must be used only when dealing with classical data. That is, in operations where the data will not be eventually mapped to qubits by the compiler — for example when computing an angle of rotation. If the classical computation is to be implemented using qubits, then the CTQG modules are more suitable, as they explicitly optimize such classical-to-quantum conversions.

2.3.2 Quantum Gates

Having laid out the data types available in Scaffold, we now turn to computation. In particular, this subsection discusses the use of individual quantum gates, and the following subsection covers program modules.

Scaffold, and its target compiled language of QASM, are technology-independent. While QASM contains gates that are more primitive and can be directly implemented by the technology, Scaffold contains higher-level gates for more expressibility. The built-in gates in the Scaffold language are rotations about the X, Y and Z basis, Hadamard, rotations around Z by $\pm \frac{\pi}{2}$, and $\pm \frac{\pi}{4}$, controlled-NOT, Toffoli, arbitrary rotations by angle $\theta$, initializing to the $|0\rangle/|1\rangle$ state and measurement in the Z basis. They are represented in the language as $X(q), Y(q), Z(q), H(q), S(q), S_{\text{dag}}(q), T(q), T_{\text{-dag}}(q), \text{CNOT}(ctrl,targ), \text{Toffoli}(ctrl1,ctrl2,targ), R_z(q,\theta), \text{PrepZ}(q,0), \text{MeasZ}(q)$.

The gates that are not covered by QASM, such as $R_z$ and Toffoli, will be automatically converted by the compiler through specialized decomposition routines. The gates that are readily available in QASM, however, will be directly transferred. In that regard, this is similar to assembly programming in C.

2.3.3 Modules

To improve the organization, readability, compilability, and error-checking for Scaffold programs, computations are encapsulated into function-like structures called modules. Each module defines a piece of the quantum algorithm, and a program typically consists of several modules, with the main module serving as the program’s entry point. Modules take quantum registers and classical variables as input and are defined using the C function syntax. While some are written with quantum variables and gates, along with classical or quantum control structures, others are written via CTQG approaches. In CTQG, modules
are written using classical operations (mainly mathematical functions like addition or cosine), with the intent of synthesizing to reversible logical circuits that are compatible with a quantum circuit. These variants will be explained in detail in the sections that follow.

Modularity is a central property of the language. In many quantum algorithms, the overall quantum circuit is synthesized using various standard sub-circuits. Some are well-known methods for quantum algorithm design, such as the Quantum Fourier Transform or Quantum Phase Estimation. Modularity allows for such modules to be designed externally, optimized, and later used in larger programs as libraries. Additionally, different instantiations of a module may be used in different parts of the circuit. For example, a 3-qubit QFT and a 4-qubit QFT may be used, but defined as only one module. The compiler then has the option of keeping them unified to save code space, or to expand and specialize them, increasing performance.

The body of a module may be a sequence of primitive gates or calls to other modules. Ultimately, this hierarchy of modules will culminate in the main module, which represents the entire circuit.

2.4 CTQG Modules: Using Classical Code in Scaffold

2.4.1 Motivation

Many quantum algorithms describe part of their functionality using classical code. Mathematical functions are a common example, where the behavior of the code may be described much more compactly by a few mathematical expressions than with a list of quantum operations. While these parts of the program are expressed as classical code, they are intended to execute as a sequence of quantum gates that operate on qubits. Scaffold must therefore support such modules, and convert them to executable quantum code. This are done through CTQG.

Figure 2.2 illustrates the concept of CTQG modules using a quantum circuit diagram. During the compilation process, a reversible logic synthesis tool generates reversible logic (i.e. quantum gates) out of these modules even if classically, they are not reversible. The programmer must be aware that using large irreversible modules may result in large ancilla overheads at this stage, because the compiler needs to pad enough qubits to create a valid reversible p-qubit to p-qubit mapping, from a possibly smaller m-qubit to n-qubit mapping.

As an example, we can look at the case of the Toffoli gate, also known as the controlled-controlled-NOT gate. The programmer has the option of specifying this 3-qubit gate either in a normal module with purely quantum code, using its decomposition into individual known 1-qubit or 2-qubit gates such as \( CNOT, T, T^\dagger \) and \( H \), or in a CTQG module using the description of the behavior. These two approaches are depicted in Figure 2.4 and their Scaffold code is presented in Figure 2.3. The succinct simplicity of the latter choice highlights the advantage of using CTQG modules.

Other instances of the convenience of CTQG use can be seen in oracle modules, which often play an integral part in the development of quantum algorithms. These oracles offer an abstract functionality, which can best be described using high level, classical functions. In some cases the circuit level synthesis of such module is also known and straightforward,
Figure 2.2: Classical code to quantum gate (CTQG) module shown inside a quantum circuit.

albeit long. In other cases however, the decomposition into gates might not be readily available.

2.4.2 Programming a CTQG Module

A simple Scaffold program is shown in Figure 2.5. This program contains a CTQG module, mult2, that functions as a multiplier by two, as described by the body of the code on line 2. The body is a classical function, which will be synthesized into quantum gates. In the main module, on line 7, we define a 4-qubit quantum register, which will then be passed to our mult2 module. Every CTQG module takes as input some qubits, and alters the state of the same qubits. This is in accordance with the nature of these modules which are to be synthesized to reversible circuits. As a result, there are no return types associated with these modules.

The quantum-mapped variables that will be allowed as arguments in CTQG modules are variable-bit. The reason for this is that classical datatype conventions (e.g. 8 or 32 bits) have been found suitable for binary computation and have historically been implemented in classical computers, as they align well with their microarchitecture (e.g. register sizes, memory systems). However, when the classical code within CTQG modules get converted to quantum bits and gates, the inclusion of unnecessary overhead might severely degrade the performance of the quantum computer. Therefore, Scaffold uses the variable-bit integer type to allow the programmer to define integers with the precise sizes needed, which will later be mapped to quantum registers.
Figure 2.3: Comparison between the usage of classical CTQG modules versus regular quantum modules. CTQG is able to express the same functionality more succinctly.

2.5 Applications Written in Scaffold

Since Scaffold’s inception, a variety of quantum applications have been written in this language. These applications cover many common themes in quantum algorithm design: Quantum Fourier Transform, Classical Oracles, State Distillation, Random Walk, and Amplitude Amplification among others. Below is a brief description of each application. Table 3.4 contains a summary of important characteristics.

- **Square Root (SQ):** A special application of Grover’s algorithm. Uses quantum amplitude amplification to search a database of $2^n$ entries. It is parameterized by $n$ (log of the number of entries) [69].

- **Binary Welded Tree (BWT):** Uses quantum random walk to find a path between an entry and exit node of a binary welded tree. The benchmark is parameterized by height of the tree ($n$) and a time parameter ($s$) within which to find a solution [39].
Figure 2.4: Decomposition of a Toffoli gate into fault-tolerant gates

```plaintext
1 ctqg mult2 (qint[4] example) {
2     example = example * 2;
3 }
4
5 module main()
6     qreg example[4];
7     mult2(example);
8 }
```

Figure 2.5: Example: A very simple program containing a CTQG module

- **Ground State Estimation (GSE)**: Uses quantum phase estimation to estimate the ground state energy of a molecule. The benchmark is parameterized here by the molecular weight \( M \), but could also be parameterized by precision [191].

- **Triangle Finding Problem (TFP)**: A quantum algorithm that uses discrete-time quantum random walk to find a triangle within a dense, undirected graph using quantum random walk. The program is parameterized by the number of nodes \( n \) in the graph [118].

- **Boolean Formula (BF)**: Computes a winning strategy for the game of Hex with quantum random walk. The benchmark is parameterized by the Hex board size \((x, y)\) [13].

- **Class Number (CN)**: A problem from computational algebraic number theory that uses quantum Fourier transform to compute the class group of a real quadratic number field. The program is parameterized by \( p \), the number of digits after the radix point for floating point numbers used in computation [70].

- Secure Hash Algorithm 1 (SHA-1): An implementation of the reverse cryptographic hash function. The message is decrypted by using the SHA-1 function as the oracle in a Grover’s search algorithm. The benchmark is parameterized by the size of the message in bits \( n \) [133].

- Shor’s Factoring Algorithm (Shors): Performs factorization using the quantum Fourier transform. The benchmark is parameterized by \( n \), the size in bits of the number to factor [165].
• **Ising Model (IM):** Finds ground state for ising model on n-qubit spin chain with error $\epsilon$. Parametrized by the qubit number ($n$) [19].

<table>
<thead>
<tr>
<th></th>
<th>Classical Time Complexity</th>
<th>Quantum Time Complexity</th>
<th>Number of Logical Operations</th>
<th>Lines of Scaffold Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>$O(n)$</td>
<td>$O(\sqrt{n})$</td>
<td>$3.1e8$</td>
<td>690</td>
</tr>
<tr>
<td>BWT</td>
<td>$O(\frac{1}{4} 2^n)$</td>
<td>$O(\frac{1}{4} k^4 n^9)$</td>
<td>$2.9e9$</td>
<td>454</td>
</tr>
<tr>
<td>CN</td>
<td>$O((n \ln n)^{0.5})$</td>
<td>$O(\log n \log *n)$</td>
<td>$3.1e8$</td>
<td>380</td>
</tr>
<tr>
<td>GSE</td>
<td>$O(2^n)$</td>
<td>$O(n^5)$</td>
<td>$9.7e7$</td>
<td>640</td>
</tr>
<tr>
<td>IM</td>
<td>$O(n, 2^n)$</td>
<td>$O(\frac{n^2}{\epsilon} \exp (a.n^b))$</td>
<td>$6.3e9$</td>
<td>100</td>
</tr>
<tr>
<td>SHA-1</td>
<td>$O(n)$</td>
<td>$O(\sqrt{n})$</td>
<td>$1.02e11$</td>
<td>430</td>
</tr>
<tr>
<td>Shors</td>
<td>$O(\exp(\sqrt[3]{\frac{9}{5} b(\log b)}))$</td>
<td>$O(b^3)$</td>
<td>$9.8e10$</td>
<td>1050</td>
</tr>
<tr>
<td>SQ</td>
<td>$O(n)$</td>
<td>$O(\sqrt{n})$</td>
<td>$2.4e9$</td>
<td>200</td>
</tr>
<tr>
<td>TFP</td>
<td>$O(n^2)$</td>
<td>$O(n^{1.3})$</td>
<td>$5.24e9$</td>
<td>3800</td>
</tr>
</tbody>
</table>

Table 2.1: Quantum applications written in the Scaffold language, and their characteristics.

The rest of this thesis uses the aforementioned applications as benchmarks to evaluate many of our proposed contributions. We will offer insight into the specific characteristics of each application, their performance and their bottlenecks. This constitutes one of the first studies in compiling such non-trivial and diverse set of quantum programs.

### 2.6 Related Work

This section offers a brief overview of the history and recent developments in QPL design, emphasizing their design objectives and points of difference.

Broadly, two categories of quantum programming languages have been under development: imperative and functional. The difference stems from the choice of host language. While imperative languages use languages like C (Scaffold [80], QCL [134]) or C++ (Q language [22]), functional languages have chosen to extend programming languages such as Haskell (Quipper [68]) or F# (LiQu [189]). The rationale behind a functional paradigm is that it makes program verification such as type checking easier, and that the application of quantum sub-circuits and recursive structures become immediate. Recently, programming languages based on scripting languages such as Python (ProjectQ [172]) have been introduced that allow for faster prototyping.

The circuit model of describing quantum algorithms is closely related to an imperative model of computation: the circuit width implies the array size to be operated on, and the
circuit depth contains timesteps of operations to be applied to those qubit arrays. One of the first efforts in designing a quantum programming language was the QCL [134] language, a C-like language which can be regarded as a primitive version of Scaffold. QCL’s supports user-defined operators, which closely resemble Scaffold modules. Its integration with a backend simulator allows for stepping through (small) quantum algorithms and observing intermediate states.

Functional languages, on the other hand, are better suited for reasoning about programs. Furthermore, these languages provide natural high-level (circuit-level) programming advantages. For example, generating the inverse of a quantum sub-circuit (which occur often in quantum programs due to the need to un-compute) or recursion (which occur in cases like the quantum Fourier transform (QFT)) are high-level operations that are addressed more easily by combining functions, as allowed under the functional paradigm. For example, consider this code for the QFT, which uses recursion to find \((n + 1)\)-qubit QFT from \(n\)-qubit QFT.

Program 2.6 shows an example 5-qubit QFT circuit implemented using the Quipper language, which is embedded in Haskell, implementing new monads. One can see that recursion is easily implementable using the functional paradigm, while the absence of array slicing (each recursion level operates on a subset of the array) makes recursion harder to implement in C-based programs (Program 2.7 implements QFT iteratively in Scaffold for comparison). Programs 2.6 and 2.7 also illustrate how Quipper is higher level than Scaffold, as the controlled rotation gate (Rz) must be separately implemented in Scaffold to be recognized (it is not part of the core universal set of gates). Scaffold’s future library extensions are aimed at remedying this.

```haskell
qft1 :: [Qubit] -> Circ [Qubit]
qft1 [] = return []
qft1 [x] = do
    hadamard x
    return [x]
qft1 (x:xs) = do
    xs1 <- qft1 xs
    xs2 <- rotations x xs1 (length xs1)
    x1 <- hadamard x
    return (x1:xs2)

where
    rotations :: Qubit -> [Qubit] -> Int -> Circ [Qubit]
    rotations _ [] _ = return []
    rotations c (q:qs) n = do
        qs1 <- rotations c qs n
        let m = ((n + 1) - length qs)
        q1 <- rGate m q 'controlled' c
        return (q1:qs1)
```

Figure 2.6: Quipper program to recursively implement a 5-qubit QFT.
Figure 2.7: Scaffold program to implement the same 5-qubit QFT as Program 2.7. Recursion is difficult in Scaffold in the absence of array slicing, so iterative methods must be used.

An interesting future course of action would be to combine different programming languages to gain numerous advantages. For example, using Quipper as a frontend to Scaffold, it would be possible to access all the useful high-level programming features of the functional language as well as the backend analysis and optimization capabilities provided by the imperative language.

2.7 Future Extensions

Scaffold has been an early attempt at providing an expressive, easy-to-use, and easily compilable quantum programming language. Since its inception, and after being put to heavy use by us and various other researchers, our understanding of quantum programming language design and some of the major challenges that must be met have also matured. In this section we first examine some of these issues, and then provide some immediate directions for Scaffold.

2.7.1 Major Issues for Future QPL Design

Any modern and versatile quantum programming language must have the following characteristics, which can be implemented through syntactic provisions or compiler support.
• **Verifiable**: An important factor in writing correct quantum code is for a compiler or IDE to catch obvious errors, so that the program can be verified. For example, a hand-coded subroutine must be reversible and unitary in order for it to be counted as a valid quantum transformation. Scaffold currently checks rudimentary errors, such as the passing of data types to a module that does not meet its definition. However, using a type system that can track certain properties of data, such as no-cloning or error probabilities, a much larger amount of useful information would become available to the programmer for writing correct code.

• **Debuggable**: Debugging a quantum program is not as easy as a classical program. Running or simulating a program is out of the question, since a useful quantum algorithm is by definition outside the scope of classical computation, and at any rate, placing checkpoints on quantum states amounts to collapsing them. Although it is possible to simulate certain aspects of Scaffold (in particular the classical modules), a more intelligent method must be devised for the fully-quantum subroutines. A possible direction is by using techniques of partial simulation. In this method, a particular subset of the large Hilbert space would be sampled and those vectors would be passed through the circuit transformation. By reasoning about coverage properties of such transformations, the output subset could be checked for correctness.

• **Efficient**: Due to cost being a primary concern in designing quantum algorithms, a programming language/compiler must be able to reason about the cost of computation. In particular, this reasoning must be optimal, meaning redundant operations or matrices must be collapsed so that the final cost is an accurate estimate of actual gates to be implemented.

• **Intuitive**: Clearly, a more intuitive, higher-level language will be easier to use especially by a community as multi-disciplinary as quantum computing. Certain subroutines recur in many quantum algorithms, such as quantum phase estimation or quantum Fourier transformation. In addition, programming paradigms such as recursion often make programming of certain algorithms easier. Moving the language to a higher level while maintaining many features of a fast, reliable compilation remain challenging.

• **Scalable**: Any quantum language and its implementation must admit large-scale analysis and/or small-scale simulations, to be viable for near- and long-term deployment in quantum computer development. Analyzing on very large scales, or simulating even on small scales, remains a challenge.

### 2.7.2 Possible Extensions to the Scaffold Language

Although efforts have been made to ensure the comprehensiveness of Scaffold, we identify the following features for near-term improvements to the language. Some of these are under development for future releases of Scaffold, as of this writing.
Reversal and Uncomputation

One area of possible expansion is the provision of *reverse* primitives. Very often in the specification of a high-level algorithm, a function is specified, followed by an "uncompute" instruction. Currently the user must manually program this uncompute sub-circuit in Scaffold, and ensure that it correctly implements the inverse of the original compute sub-circuit. It is preferable that this task be automated by the compiler.

One of the reasons for the prevalence of uncompute sub-circuits is that quantum circuits are *always* reversible. That means when programmers want to design the quantum equivalent of a classical function with $m$-qubit inputs and $n$-qubit outputs, they must add ancilla qubits to create an equal number of inputs and outputs. These ancilla qubits are unintentionally entangled with the data qubits, and part of the job of the uncompute circuitry is to undo this entanglement. After this point the ancilla qubits are reusable for other parts of the overall program. This is very similar to garbage collection in classical compilation.

Automated reversal faces several restrictions. One strategy is to forbid the destruction of quantum data (measurements, creation of local ancilla qubits) inside the modules that are called using this primitive. Another issue that needs to be addressed is the reversibility of modules that include dynamically generated gate sequences. One possible solution to this problem is to provide a primitive to store the sequence of gates generated during a single call of the module and to provide a mechanism to apply this sequence in normal or reversed order later on. The code snippet in Figure 2.8 illustrates the basic idea behind this concept:

```plaintext
// Module instance
instance m1;

// Call module capturing gate sequence
module1(qubit q1, qubit q2) {
  ...
  m1;
  ...
}

// Call the instance of gates
call m1;
// Call the instance backwards
_reverse_m1;
```

Figure 2.8: Pseudocode representing the idea behind the quantum reverse primitive

Note that this would require runtime support for capturing gate sequences, some of which may be largely dependent on the algorithm.

27
Better Libraries

Another possible extension to the language is the expansion of libraries. For example, simple math functions can be implemented to act upon quantum registers. While general implementations of such functions would likely be very resource-intensive, specialized and optimized versions could be offered as library functions for use when needed. Another example is the controlled-Rz gate. This gate has a simple decomposition in terms of CNOTs and uncontrolled gates, which can be leveraged in libraries for easy access. Common techniques such as Quantum Fourier Transform, Quantum Random Walk, and Quantum Phase Estimation recur frequently in quantum programs, and well-designed libraries can save time and effort in development of future programs.

Array Slicing

As Scaffold is implemented in C, it has no native capability for array slicing. C allows indexing into individual elements of an array, but has no support for selecting a range of elements from index $i$ to index $j$. A Verilog-like mechanism for addressing individual qubits is highly desirable, since unlike classical computing, quantum registers need not be uniform (in fact, this is undesirable as it might incur extra overhead on qubits, currently the most precious resource). Individual addressing capabilities, or the ability to address a subset of a qubit register, will greatly enhance the flexibility of the language.

Controlled-U Constructs

Controlled versions of a unitary matrix are often used when an already-implemented sub-circuit’s execution needs to be conditioned on the state of a different ”control” qubit (or several controls) in the overall circuit (the sub-circuit would act only when the qubit is in state $|1\rangle$; a subset of the Hilbert space during superposition).

For the case of one-qubit control, this is simple to implement in terms of CNOTs and uncontrolled versions of the unitary (Figure 2.9 illustrates an example). For doubly-controlled sub-circuits (depending on whether two “control” qubits are in state $|11\rangle$), this would require more thought. Scaffold currently does not support this type of construct, requiring the programmer to specify exactly how the controlled-U will be implemented. An interesting research project is to identify ways to automate this procedure for 2 or more number of controls.

```plaintext
module cRz (qubit ctrl, qubit target, const double theta) {
    Rz(target, -1*theta/2);
    CNOT(target, ctrl);
    Rz(target, theta/2);
    CNOT(target, ctrl);
}
```

Figure 2.9: Pseudocode representing the idea behind the quantum reverse primitive
2.8 Chapter Summary

This chapter presented Scaffold, a quantum programming language based in C that allows the expression of quantum algorithms. In addition to describing the language features of Scaffold, we also highlighted broader design choices arising in quantum programming languages — choices that must advance the expressibility of high-level constructs, and the verifiability of quantum programs. Several recurring themes, such as specialization of program inputs, mixing of classical and quantum code, and the use of well-known methods such as Quantum Fourier Transform should serve as guides to language design.

The next chapter concerns the implementation of compiler support for Scaffold. We show how existing compiler frontends can be leveraged to robustly lower Scaffold to a circuit-level language, and the issues that arise as a result of compiling large quantum programs.
Chapter 3

ScaffCC: A Framework for Compilation and Analysis of Quantum Applications

3.1 Introduction

This chapter is centered around the issue of compilation and automatic analysis of quantum programs. A well-designed compiler must take into account quantum-specific aspects of the code, such as qubit entanglements or uncomputation and garbage collection. It must be able to analyze the code quickly, and estimate qubit count and latency — currently the biggest obstacles to quantum computing. Finally it must generate high-quality code. Deep optimizations are necessary to bring the cost of implementation down.

Another important indicator of a good compiler for quantum applications is its scalability. Such compilers must operate on very large codes: high-speed classical computation will only be surpassed in the context of large-scale computations (e.g. factoring a 2048-bit number), however long it would take for quantum technology to catch up. It is vital for researchers to have a good understanding of future algorithmic requirements as the technologies and architectures mature.

This chapter introduces ScaffCC, an open-source tool created for the purpose of compiling and analyzing Scaffold programs. We describe how traditional and novel compiler passes may be used to compile quantum programs, and how the issue of scalability can be addressed.

In this chapter, we describe the unique requirements of quantum program compilation and design ScaffCC, a compiler framework with an extensible quantum program analysis toolbox, for quantum programs that contain trillions of operations.

Most previous work has focused on designing, mapping, and scheduling hand-optimized quantum circuits for implementing small-scale quantum algorithms. By introducing ScaffCC, we focus on automated tools which can support a broad range of large-scale quantum benchmarks.

This chapter makes the following contributions:

\footnote{The work in this chapter was performed in collaboration with Shruti Patil, Alexey Lvov and others \cite{81, 82}.}

\footnote{ScaffCC is available at the following link: https://github.com/epiqc/ScaffCC.git.}
First, we make key observations regarding the differences between classical and quantum compilation. For example, quantum programs typically specify a fixed circuit, and therefore contain one execution trace. They are also commonly compiled for specific input sizes or values. As a result, they yield deeply, statically analyzable code, mitigating the need for optimizations such as branch prediction and emphasizing other optimizations such as parallelization of operations (instructions). This creates opportunities for aggressive constant propagation and deep optimization, while simultaneously putting greater pressure on the scalability of the compiler algorithms employed.

Second, we present compiler algorithms and compiler output formats that can accommodate the large scale and deep optimization found in our quantum benchmarks. In particular, we find that output modularity and a dynamic, instrumentation-driven compilation technique are important to managing scale.

Third, despite the differences inherent in quantum compilation as opposed to the classical case, we show the applicability of known classical compiler algorithms, such as loop unrolling and procedure cloning, to the domain of quantum computing. Our compiler leverages mature compiler technologies through the LLVM framework.

Fourth, we present data-flow analysis as an example of classical techniques employed in the quantum domain. In particular, we propose the use of data-flow analysis techniques, both for important program checks such as “no-cloning” and “entanglements,” and also for obtaining circuit estimates such as the critical circuit path or its usage of qubits and operations. These metrics help focus further optimizations.

Finally, recognizing that quantum programs often use classical reversible logic to describe sub-circuits of a quantum circuit, we present a novel technique for their compilation and simulation.

The rest of this chapter is organized as follows: Sections 3.2 offers an overview of the compiler we have developed to translate from high-level quantum algorithms to lower-level quantum assembly operations. Sections 3.3, 3.4 and 3.5 describe the research challenges in different parts of the compiler toolflow, including techniques to manage large scale and to synthesize from classical reversible logic. Section 3.6 discusses analysis passes enabled by the ScaffCC functionality. Section 3.7 surveys related work. Finally, Section 3.8 discusses future directions for quantum compilers, and Section 3.9 offers conclusions.

3.2 Internal Structure of ScaffCC

ScaffCC compiles a program written in the Scaffold programming language, and outputs a quantum assembly (QASM) representation. It targets logical quantum computation, that is, compilation, analysis and optimizations before synthesis into machine-dependent physical-level operations. This section gives a broad overview of the input and output languages, and the design of the ScaffCC compiler.

Figure 3.1 depicts a block diagram of ScaffCC’s internal structure. We have implemented ScaffCC in LLVM [110], a rich, open-source library of compiler technologies, by adding intrinsic functions representative of quantum gates and a new datatype representative of qubits. Furthermore, we have extended Clang, a C-family frontend to LLVM, to accommodate parsing of our language.
Figure 3.1: Internal structure of the ScaffCC compiler: The top, middle and bottom parts respectively show translation of CTQG modules (Section 3.5), QASM code generation (Section 3.4), and quantum program analysis (Section 3.6).
3.2.1 Compilation of Quantum vs. Classical Modules

The first step of compilation is to separate the modules in the program which are marked as CTQG. This separation is shown in the red (top) and blue (middle) tracks of compilation in Figure 3.1. CTQG modules have been defined by the programmer using classical gates (for example describing a classical math function), and are handled by the separate CTQG sub-compiler as described in Section 3.5. CTQG’s output is translated directly to QASM without going to LLVM’s intermediate format, and is linked with the output of the quantum modules after they have been converted to QASM. Although this approach yields fast output code generation, it is not suitable for whole program analysis since a part of the code will bypass the LLVM-IR representation. Thus, we have implemented a QASM-to-IR translator which we use to convert the entire program once it has been compiled. This provides correct input for quantum program analysis.

A critical code generation issue lies in the degree to which output code can or should be linearized (or flattened). We refer to this as “classical control resolution”. Our goal is to establish a judicious balance—we wish to flatten as much as possible in order to support efficient synthesis of quantum circuits, while also keeping enough abstraction to ensure circuit generation remains tractable. During the compilation of non-CTQG modules, it becomes necessary to process some of the classical instructions within them, in order to remove high-level abstractions and obtain sub-circuits that clearly specify the sequence of gate operations and qubits which are acted upon. This amounts to flattening the program on a per-module basis, and is required for correct scheduling and mapping during later stages of the toolchain. Unfortunately, performing code linearization in a way that scales well and does not result in a time and space explosion is non-trivial. Section 3.4 has a detailed description of this step and explores ways to make it execute faster.

3.2.2 Generating QASM-Language Output

The last phase of the compiler performs a decomposition of unitary operations into supported gates in QASM, which is a subset of those allowed in Scaffold. This is a key step in the translation of a high-level program into a standard assembly language, and is similar to instruction selection in classical compilers. For some gates, this is a straight-forward process. For example, the output of CTQG contains many “Toffoli” operations, which in order to be compatible with QASM, would each be substituted by a fixed 16-gate sub-circuit. Other gates, such as rotations by arbitrary angles, may be more complex. We employ a state-of-the-art method, as proposed in [105], to approximate these gates.

QASM has been used to implement and study quantum circuits for small problems using a flat circuit format [55, 123, 150]. However, realistic quantum circuits that we examined contain between $10^7$ and $10^{12}$ logical gates, rendering full flattening infeasible. In Section 3.3, we introduce modifications to the original flat format that enable more manageable code sizes.
3.2.3 Program Analysis

Finally, as Section 3.6 discusses in detail, ScaffCC can perform a range of useful analyses on its input programs, both for program correctness checks and for circuit estimates. The LLVM toolkit represents computations as graphs, which facilitates program analysis. This is the green track at the bottom of Figure 3.1).

3.3 Managing Scalability Through Choice of QASM Format

In compiling large-scale programs of trillions of operations, it is important to design a target language that can accurately relay the circuit information to physical controllers, yet itself remain manageable in size. This is especially important when considering the type of hardware on which quantum computers are likely to be implemented. For example, dilution refrigerators are the method of choice in superconducting quantum computers, which cool the chip to near zero Kelvin temperatures. In such an environment, power dissipation and size of memory (of both instruction and data) are extremely limited. We cannot afford to load huge numbers of instructions onto the controller.

In this section, we consider QASM format adjustments over previous flat-code proposals, and study their impact on code generation feasibility.

3.3.1 Hierarchical QASM format (QASM-H)

QASM programs can be represented by a space-consuming flat description, or by a denser hierarchical description which takes advantage of sub-circuit duplications to reduce the output code size. Some modularity is also desirable for program analysis of large codes. Analysis techniques when applied hierarchically reduce analysis time and memory usage, thus scaling better to large program sizes. We demonstrate this through the example of timing analysis in Section 3.6.3.

3.3.2 Hierarchical QASM with Loops (QASM-HL)

Further information about repeating quantum operations can be retained within the QASM format, in the form of loops. Quantum circuits show two prominent types of quantum operations: The first type are operations that are applied to a large set of qubits. These are used, for example, when transforming qubits prepared in the ground state into initial superposition states. Due to the absence of qubit dependencies, these operations are highly parallel and are implemented simultaneously when the hardware technology allows it. (For example one can use control technologies such as microwave traps that affect a large number of qubits at the same time.) We denote these as forall loops.

The second type of operations are serially repeated transformations, typically used in quantum algorithms to converge to a more precise solution. For example, Grover’s Search Algorithm makes use of a repeated invert-and-reflect operation that gradually increases the likelihood of measuring the correct answer. In the physical implementation, the control
#define n 1000
module foo(qbit q[n]) {
    for(int i=0;i<n;i++)
        H(q[i]);
        CNOT(q[n−1],q[0]);
} module main() {
    qbit b[n];
    foo(b);
}

module foo(qbit * q) {
    H(q[0]);
    H(q[1]);
    ...
    H(q[999]);
    CNOT(q[999], q[0]);
} module main() {
    qbit b[1000];
    foo(b);
}

module foo(qbit * q) {
    H(q[0:999]);
    CNOT(q[999], q[0]);
} module main() {
    qbit b[1000];
    foo(b);
}

Figure 3.2: Code Snippets for QASM-F, QASM-H and QASM-HL: Progressively more classical control is retained. Note that Scaffold does not contain pointers or allow their manipulation, but QASM address representation for accessing memory resembles C syntax for ease of use with LLVM.

exercised for the sequence of operations within the loop body can be synthesized once, and then reused. We denote these as repeat loops.

In order to identify quantum forall and repeat loops in high-level programs, we define a pure quantum block as a basic block that conforms to the following criteria: 1. It does not contain classical computation instructions such as arithmetic or compare instructions; 2. It does not contain function calls which have non-quantum data types as arguments; 3. The qubit array variables depend directly on the loop induction variable. Through static analysis of the loops around the purely quantum blocks, we can obtain trip counts to provide the number of repetitions for the repeat loops, and loop values to provide the range of qubits that are simultaneously operated upon in the forall loops. This allows for efficient optimizations and analyses.

3.3.3 QASM Code Size Comparison

Figure 3.3 shows the reduction in code size when using QASM-HL over QASM-H. A great advantage in code size is already obtained across all benchmarks when using QASM-H as opposed to flat QASM.

Referring to this figure, QASM-HL output format greatly helps code size for the SQ and BWT algorithms, making an exponential growth with problem parameters into a linear one. The reason is that these algorithms make use of repeat blocks with high iteration
count, in a manner that converges the quantum states to the correct results. As programs scale, the increased number of quantum operations is captured within the repeat loop of QASM, keeping the code sizes small. On the other hand, the TFP algorithm has numerous forall blocks, but a relatively low number of repeat blocks. As the problem size for this algorithm scales, the trip counts of forall loops capture the increased number of qubits being operated upon, resulting in some code improvement. For three of the benchmarks, not much advantage is gained when using QASM-HL over QASM-H. In the GSE program, very few pure quantum loops and with low trip counts exist, impeding the effectiveness of loop retention. In addition, a major part of the BF and CN circuits are compiled using the CTQG sub-compiler, which outputs a flat circuit format. Quantum loops constitute a very small percentage of the non-CTQG part, resulting in only slight code size improvements. Overall, QASM-HL’s advantage is in making compilation tractable for more programs.

![Figure 3.3: Reduction in code size of QASM-HL compared to QASM-H output, due to retention of quantum loops.](image)

### 3.4 Managing Scalability through Instrumentation-Based Compilation and Analysis

Another important goal of ScaffCC is for the compilation passes to scale well with increasing circuit sizes. As previously defined, QASM-HL supports this by allowing modularity and repetitions in the output code, which mitigates the size explosion that results from flattening the whole circuit (recall operation counts of more than $10^{11}$ reported in Table ). However, with the exception of some loops, QASM-HL still requires per-module flat code to enable effective circuit synthesis. Therefore, many classical control constructs, such as
if-then-else conditionals, non-quantum loops, parameterized modules, etc. must be pro-
cessed in the compiler. Scaffold programs contain the description of a quantum circuit and
are thus specialized for a particular set of input parameters (or problem sizes), yielding
deeply analyzable programs. This fixed-trace nature of program control-flow and its inde-
pendence from qubit states means that all classical control-flow constructs can be resolved
in the compiler.

This section begins with a motivating example regarding the need for classical control
resolution, and then describes methods for compiler implementations of it. The speed and
tractability advantages of our second method over the first are discussed at the end.

3.4.1 Dual Nature of Quantum Programs

Consider Figure 3.4 which shows a segment of a Scaffold program where the module main
contains calls to module Oracle located inside two nested loops. For each different value
of \( j \), a different version of Oracle is called, since the rotation angle in the \( R_z \) rotation
gate changes. In order to correctly decompose this gate, the compiler needs to disambiguate
these different module versions, and obtain the correct rotation angle for each one to ar-
rive at its equivalent set of gates. This is why, for example, QASM-HL does not contain
parameterized modules. We investigate how the compiler can automate this resolution of
classical control.

This illustrates that a program written in a quantum programming language inherently
has two interleaved portions: quantum data and classical control. We can resolve classi-
cal control structures precisely because they only control the program flow, and they do
not depend on the states of data qubits (which probabilistically change at runtime). The
computation is what occurs on quantum data, once the classical controls are followed.

In principal, all of the classical control can be brushed off and resolved at compile time,
leaving a trace of only primitive quantum operations. This is equivalent to full flattening
of the program. However, we must choose which parts of the classical control we want
to resolve. For example, in the above example, rotation angles would not be known if the
constants at call points are not propagated to their relevant functions. This would prohibit
the decomposition of those rotations, putting an extra burden on runtime to decompose
them. However, certain other structures such as loops are more favorable to preserve, since
they can be easily interpreted by the controller at runtime.

3.4.2 Pass-Driven Approach

Our first approach, pass-driven, relies on static usage of transformation and analysis passes
such as heavy constant propagation and constant folding. It processes the modules in the
call graph of the program in depth-first, pre-order and unrolls all loops that have not been
marked as quantum loops. It further clones those modules that are called with different pa-
rameters in multiple call-sites, and uses inter-procedural constant propagation to specialize
those modules. These steps are repeated until there is no further action to be taken. Since
ScaffCC uses the LLVM infrastructure, several pre-written passes are available for these
transformations; we adopt these and expand on them.
```c
#define s_ 3000 // iteration count

module Oracle (qbit a[1], qbit b[1], int j) {
   double theta = (-1)*pow(2.0, j)/100;
   X(a[0]);
   Rz(b[0], theta);
}

module main () {
   qbit a[1], b[1];
   int i, j;
   for (i=1; i<=s_; i++) {
      for (j=0; j<=3; j++) {
         Oracle(a, b, j);
      }
   }
}
```

Figure 3.4: Example Scaffold program showing the need for classical control resolution. Different versions of the same module with different gate sets are created, but can be discovered either statically using compiler passes such as loop unrolling and procedure cloning, or dynamically using instrumentation and execution.

In its most naive form, the pass-driven approach consists of flattening the quantum circuit. Using this method, the usage of all qubits and gates (the quantum portions of the code) become explicit. This process would result in a single module that declares all qubits (including data and ancilla) in the circuit and all the operations applied to those qubits from start to finish. Obviously, the lack of abstractions in the output file causes this method to quickly become infeasible for large circuits. However, we can introduce increasing levels of abstraction into the analysis which would mitigate this explosion in space, and yet still provide the ability to fully propagate constants and resolve classical dependencies in the code. Therefore, we aim to preserve the modularity of programs and retain some analyzable loop structures.

Referring back to Figure 3.4, we begin by unrolling the inner loop in module `main` by a factor of 4, which causes all call sites to module `Oracle` to have constant call parameters. We then use procedure cloning to create a module clone from each call site that has a unique set of input parameters. We then use inter-procedural constant propagation to propagate the input parameter constants of the call sites into each corresponding module. Repetitive application of these transformation passes (loop unrolling, function cloning and constant propagation) yields a program that preserves modularity but is flattened on a per-module basis. These steps are illustrated in Figure 3.5 This process is equivalent to a partial execution of the code. This example also illustrates a further optimization: simple loops, as discussed in Section 3.3, may be kept since their loop bodies are all quantum and their resource usage can be multiplied by the loop trip count. The outer loop in module `main` is an example of this kind of loop, where the input parameter “s” is a timestep variable that
indicates the number of required iterations over a circuit segment in order to converge to the answer. This is the major source of computation in this benchmark; avoiding unrolling the loop will mitigate the space complexity.

Figure 3.5: Pass-driven compilation using standard compiler passes of loop unrolling, function cloning and constant propagation. This process can easily lead to explosion in space.

### 3.4.3 Instrumentation-Driven Approach

The pass-driven approach can quickly become cumbersome for algorithms that have many large modules. The transformation passes can create progressively larger intermediate code sizes (as illustrated in Figure 3.5), which will incur unacceptable space and time costs. Here, compilation time is a tangible metric that will suffer as a result of the compiler being forced to handle very large intermediate formats. To address this, the *instrumentation-driven* approach shifts from static code transformation to an execution-based transformation. This shifts the job of resolving classical dependencies to the classical processor, recognizing that fast classical processors can be used to execute through classical portions of the code and collect information regarding the quantum part. The instrumentation-driven approach will produce identical QASM output as the pass-driven approach, yet in a more scalable manner.

In order to execute a Scaffold code, we need to instrument it. The classical parts of Scaffold are identical to C, and therefore the only parts standing between Scaffold and regular C code are the qubit declarations and gate operations. In order to make the quantum aspects of the code compatible with classical processors, we first use ScaffCC to compile the Scaffold program into its LLVM intermediate representation, and then use a custom
pass that will instrument this bytecode such that quantum data declarations and operations are changed into external calls to a library for this purpose. In this case, we want those operations to be converted to QASM representations. Therefore every gate operation would be converted to a call to a classical instruction that prints the relevant gate.

For such source-to-source rewriting, a naive instrumentation approach would be to instrument the quantum instructions to print themselves textually as the classical component executes. However this will result in a flat output. For QASM-HL output, the instrumentation approach must preserve modularity during execution. For this purpose, the program is modified to execute in two modes, the quantum mode and the classical mode. In the quantum mode, the instructions in a module are rewritten into the quantum assembly format, while in the classical mode, the call paths are followed to determine the next set of modules to be translated. In particular, we use “procedure cloning” to create the quantum version of the module from the original version (denoted as the classical version).

In the quantum mode, each module resolves the sequence of its quantum operations and quantum data references, by executing the classical control instructions within it. Once the quantum operations and their operands are extracted, they are converted into QASM-HL format and written to the output file. To achieve this, each quantum operation is instrumented with a print function that prints the operation type and the resolved data operand references in the QASM-HL syntax. The function calls to other modules are also instrumented, but removed in order to prevent them from executing. Once the instrumentation pass is performed, a dead code elimination pass is used to remove the dead instructions in the quantum version.

The classical version of a module is instrumented to invoke the quantum version of the module, before executing the function calls contained within it. In this module, the quantum instructions are removed, leaving only the function calls intact. To prevent repeated execution of the module, runtime decision instructions are added at the beginning of the classical version. We use the technique of memoization to determine if the module was executed previously, by inserting it into a look-up table. As further optimization, loop iterations that have exactly identical call sequences, including their call parameters, are removed so that the call sequence is executed only once.

### 3.4.4 Compilation Speed Comparison

Figure 3.6 shows the improvements of the instrumentation-driven approach over the pass-driven approach in overall compilation time across the range of all quantum benchmarks. Results were collected using a 2.27 GHz, Intel Xeon CPU with 24 MB of shared cache and 126 GB of RAM. For each benchmark, the compilation time is normalized to the pass-driven time of the smallest problem size. As problem sizes increase, the instrumentation-driven approach scales better than the pass-driven approach. This amounts to significant improvements in compilation time for large benchmarks. For example, for the Triangle Finding Problem with problem size \( n = 15 \), the instrumentation-driven approach generates QASM-HL code within ~20 hours, while compilation using the pass-driven approach takes several days.
Figure 3.6: Improvement in compilation time with the instrumentation-driven technique over the pass-driven, for different problem sizes. Figure is scaled to the pass-driven time for the smallest problem size (the left-most blue bar in each benchmark cluster). Pass-driven compilation can be faster for small benchmarks, while instrumentation-driven compilation supports larger benchmarks. On average, instrumentation-driven technique is 3X faster.

3.5 Compiling Classical CTQG Modules

In many important quantum algorithms a large portion of modules use only classical reversible logic operations—operations which can be decomposed into the universal set of NOT, CNOT and Toffoli gates.

Typically, reversible logic submodules (often called ‘classical oracles’) are at the bottom of the dependency hierarchy and by gate count constitute more than 90% of the circuit. CTQG is a source of inefficiency in the sense that it creates Flat QASM. Also, unlike general quantum circuits, classical oracles can be simulated on a conventional computer allowing a continuous development cycle: 1. write code, 2. test by simulation, 3. correct bugs (if any). Compiling classical oracles separately gives an advantage of being able to verify a significant part of the quantum circuit by simulation.

At the first step of compilation ScaffCC detects all purely classical reversible logic modules and compiles them using CTQG, a sub-compiler converting to flat QASM format. Later during compilation, these precompiled classical oracles are inserted verbatim into the final code every time a call to an oracle is encountered. Also CTQG allows code developers to simulate any oracle on any set of input signals for verification and debugging purposes.

Many important basic operations such as integer arithmetic, fixed-point arithmetic, manipulations with bit strings, allocation of ancilla signals, if-then-else statements and loops with non-quantum bodies can be expressed solely by means of classical reversible logic. CTQG uses state-of-the-art built-in algorithms to compile these operations and pass them as QASM code to ScaffCC.
The basic integer arithmetic operations in reversible logic are \( a = a + \text{const} \), \( a = a - b \) and \( a = a + bc \) where the variables are integer numbers in standard binary \( n \)-bit representation. For reversible adder and subtracter, CTQG uses a recently developed algorithm by Cuccaro et al. [46] which uses \( 6n - 3 \) CNOT gates, \( 2n - 2 \) Toffoli gates and does not require any ancilla signals at all. That is, CTQG adder and subtracter have size linear in the bit width of the arguments. The CTQG integer multiplier uses similar ideas (see [46]); it has size \( O(n^2) \) and uses no ancilla signals either. Using a constant integer expression inevitably requires ancilla signals because reversible logic does not allow constant '0' or '1' gates. However CTQG automatically recycles ancillas used for representation of constants that are no longer needed. For example only 8 ancilla signals (not 24 as with a brute force approach) will be allocated by CTQG for the module that computes \( \{ a = a + 231[11100111]; b = b + 219[11011011]; c = c + 189[10111101]; \} \).

Fixed-point arithmetic analytic functions such as \( \frac{1}{x} \), \( e^x \), \( \sin x \), \( \cos x \) and \( \ln x \) are much harder to implement in reversible logic. To the best of our knowledge, there exist no purely reversible circuits for these functions. CTQG has a built-in implementation of these functions which uses fewer ancillas than a brute force Taylor series approach. For example for \( \frac{1}{x} \) we use infinite product representation:

\[
\frac{1}{x} = (2 - x) \cdot \left(1 + (1 - x)^2\right) \cdot \left(1 + (1 - x)^4\right) \cdot \left(1 + (1 - x)^8\right) \cdot \ldots
\]

which has doubly exponential convergence \( \forall x \in [1/2, 1] \), and produces \( O(n^2 \ln n) \) gates and \( O(n \ln n) \) ancillas. For \( e^x \), \( \sin x \), \( \cos x \) and \( \ln x \) our built-in functions produce \( O(n^3 \ln n) \) gates and \( O(n^2 \ln n) \) ancillas.

In order to produce \( \text{if (bit) \{body\}} \) circuits, we add \text{bit} as an extra control signal to every gate of \{\text{body}\}. This transforms NOTs to CNOTs, CNOTs to Toffolis, Toffolis to 3-control Toffolis, etc. Any \( n \)-control Toffoli then decomposes into a number of regular Toffolis. Arbitrary depth embedded if-then-else decomposes into elementary reversible gates by applying the above procedure several times.

Generally neither conventional nor reversible circuits can have loops. However if the maximum number of loop iterations can be predetermined, then the loop can be “unrolled” producing an amount of gates approximately equal to this maximum iteration count multiplied by the number of gates in the loop body. Figure 3.7 is an example of a circuit written in CTQG that computes \( 1 + 2 + 3 + \ldots + n \) for a given iteration count in a brute force fashion.

Although CTQG has a large variety of highly optimized built-in reversible logic functions and methods for generating new reversible functions from the existing ones, they are not sufficient to generate all reversible functions just by their conventional (non-reversible) description optimally with respect to the usage of ancilla signals. As it is shown in [164], the set of gates \{NOT, CNOT, TOFFOLI\} is universal for all reversible boolean functions which represent even permutations on the set of all possible input bit strings. Only one reusable ancilla signal is required to remove the “even permutations” constraint. [164] gives an explicit algorithm for representing any reversible boolean function given by a table of values as a sequence of \{NOT, CNOT, TOFFOLI\} with only one extra ancilla signal. Providing that its table of values fits in memory, any small reversible circuit can be gener-
#define M 100

    int control_i;
    $ i := 1;
    $ sum := 0;

    for (control_i = 1; control_i <= M; control_i++) {
        $if (i <= n)
            $ sum += i;
        $endif
        $ i += 1;
    }
}

Figure 3.7: Sample CTQG code, showing the usage of loops.

ated this way optimally and then be programmed in CTQG by directly listing the sequence of \{\text{NOT, CNOT, TOFFOLI}\} gates.

CTQG is a one-pass compiler and is able to produce QASM output gate by gate “on the fly” without remembering any of the previously produced gates. Thus, it can work on circuits as large as $10^{12} - 10^{13}$ gates, with the limiting factor being only the runtime but not the memory size.

3.6 Analyzing Scaffold Programs

One of the most important uses of a quantum compilation framework is to obtain information about quantum algorithms and their implementation. Programming for quantum devices can be error-prone—one must have good reason to believe that the intent of the algorithm is reflected correctly, and that the implementation does not violate the laws of quantum mechanics. An example is the no-cloning theorem, which requires that the state of one qubit cannot be copied into the state of another while maintaining the first state [132]. This is a necessary, albeit not sufficient, condition on the soundness of code. As a result, ScaffCC uses aliasing analysis to emit error messages when a programmer tries to use a multi-qubit gate on the same qubit, since that quantum state cannot be mapped onto two distinct qubits.

The next sections describe ScaffCC analyses that not only help in program validity checks, but also give timing or resource estimates for the algorithm’s circuit, which are very useful for later application-aware optimizations.

3.6.1 Entanglement Analysis

Entanglement is a fundamental phenomenon in quantum mechanics, denoting a logical relation between measured states of qubits. An example wave function of two entangled
qubits is $|\psi^+\rangle = (1/\sqrt{2})|00\rangle + (1/\sqrt{2})|11\rangle$. It shows that the measurement states of the two qubits are logically related to each other. For example if one is measured and collapsed to state $|0\rangle$, the other qubit will also collapse to the same state. This phenomenon is extensively used for logical transformations of quantum states and for fast communication using quantum teleportation. Further, it is the key reason behind exponential speed-up possible with certain forms of quantum computation [85].

Since entanglements affect the final outcome of qubit states, a view of entanglements occurring within a quantum program is useful to the programmer for both designing algorithms and debugging. To analyze the large number of qubits in a quantum program, we use data flow analysis techniques to automate the process of tracking entanglements. The entanglement analysis pass in ScaffCC performs a conservative analysis, adding annotations in the output QASM-HL program to denote entangled qubits. Figure 3.8 shows an example of a module annotated with its entanglements.

```plaintext
module EQxMark_1_1 ( qbit* b, qbit* t ) {
    ...
    Toffoli ( x[0], b[1], b[0] );
    // x0, b1, b0
    Toffoli ( x[1], x[0], b[2] );
    // x1, x0, b2, b1, b0
    Toffoli ( x[2], x[1], b[3] );
    // x2, x1, b3, x0, b2, b1, b0
    Toffoli ( x[3], x[2], b[4] );
    // x3, x2, b4, x1, b3, x0, b2, b1, b0
    CNOT ( t[0], x[3] );
    // t0, x3, x2, b4, x1, b3, x0, b2, b1, b0
    Toffoli ( x[3], x[2], b[4] ); // x3
    Toffoli ( x[2], x[1], b[3] ); // x2
    Toffoli ( x[1], x[0], b[2] ); // x1
    Toffoli ( x[0], b[1], b[0] ); // x0
    ...
} // Final entanglements:
// (t0, b4, b3, b2, b1, b0);
```

Figure 3.8: Entanglement annotations in the $EQxMark$ module of the Square Root (SQ) application. Entanglements are added as a comment to every instruction that creates them.

Two qubits are entangled when their individual wave functions are inseparable, such that only the joint wave function and not individual wave functions can be described. In reality, determination of entanglement would require precise tracking of quantum states and transformations of qubits, however a conservative analysis without knowledge of actual states is possible by tracking simply the interactions with other qubits. It is based on the observation that if two qubits interact, they are likely to have become entangled with each other. Such interactions occur when multi-qubit operations are performed. In particular, of the primitive gates allowed by Scaffold, the CNOT and Toffoli operations potentially
create entanglement among their operand qubits. ScaffCC performs this analysis for qubits in every module: control and target qubits from multi-qubit operations are stored in a table as they are encountered within each module, and the instructions are annotated with these pairs. Since the entanglement property is symmetric, reflexive and transitive, the previous entanglements of the control and target qubits are also added to the annotations.

In addition to compute instructions, quantum programs also contain uncompute instructions to reverse state changes of ancilla qubits. The CNOT and Toffoli operations are inverse functions of themselves; therefore they create disentanglements when reapplied to the same set of control and target qubits. Thus, entanglement analysis also involves tracking of uncompute portions in a module.

To be able to identify disentanglements, sets of control and target qubits are stored along with a timestamp for each gate. When the same gate with the same set of (target, control) qubits is re-encountered, the control qubits are examined for state changes since the timestamp of the original instruction. Changes are determined by whether the qubits served as target qubits in other instructions. If any changes were determined since the entangled instruction, the (target, control) pair is retained in the table, along with the new pair and timestamp. Otherwise, the instruction is marked as a reverse operation, and a disentanglement is recorded. This removes the (target, control) entries from the table. As a consequence, if the set of control qubits for a target qubit becomes empty, it is assumed to have been restored to its original state, and is removed from the set of entangled qubits. Figure 3.9 illustrates this process to determine the entanglements and disentanglements for a small example circuit with two data qubits and two ancilla qubits.

### 3.6.2 Resource Analysis

The high implementation cost of qubits and operations underscores the importance of a program analysis which can quickly calculate the number of resources consumed by that program. This number can serve as an early comparison of the resource requirements of different algorithms before implementation on a physical device, as well as a form of feedback to other parts of the compiler (as discussed in 3.6.4). Qubits remain the most expensive resources in quantum computing, but the number of gate operations also matters – more gates increase the likelihood of error, thus requiring more error correction which in turn requires more qubits.

As previously mentioned, the compiler operates at the logical level and the number of resources reported are fewer than those in the final implementation of the circuit due to the overhead associated with error correction. Chapter 6 will evaluate the cost of added error correction.

Resource analysis as a form of whole-program analysis can also be carried out using pass-driven and instrumentation-driven approaches, similar to what was discussed in Section 3.4. In this case, either an additional compiler pass would count the number of qubits and operations on the LLVM-IR code, or instrumentation would yield a program which upon execution collects its own resources. The instrumentation-driven approach again performs better for larger problems.

In order to obtain the best of these two approaches, we combine them into a hybrid solution. The hybrid approach, while employing instrumentation to execute the classical
portions of the code dynamically and thus avoid space costs, also analyzes the trip count of loops statically using LLVM passes. An early loop rollup pass would mark the program bytecode with information about the trip counts of loops, and then change the loop conditions to only execute once. The marked information is used by the instrumentation pass to make the hash table entries be multiplied by certain numbers and not just incremented.

For our various quantum algorithm benchmarks, this hybrid instrumentation-driven approach is on average 63% faster than the pass-driven approach. It may also be observed that as problem sizes increase, the advantage margin generally increases, showing the scaling properties of the instrumentation-driven approach.

### 3.6.3 Timing Analysis

Even if the compiler has no knowledge about a hardware implementation’s resource constraints, high-level timing analysis can estimate the circuit’s critical path length by reordering instructions in order to optimize the logical circuit’s length. For a given sequence of quantum instructions, ScaffCC performs a hierarchical critical path estimation, which involves the scheduling of instructions with the assumption of unbounded quantum resources. The no-cloning theorem enforces a data dependency between quantum instructions when
they share one or more operands (there is no difference between reads or writes, contrary to classical computing.) Adhering to these dependencies, the critical path timing analysis schedules operations by reordering instructions in *as-soon-as-possible (ASAP)* order.

Since the quantum program traces can be exceedingly large, we take advantage of modularity to arrive at a critical time estimate. The algorithm proceeds in postorder of the call graph of the program, processing leaf modules before the non-leaf ones. Algorithm 1 describes the analysis. It uses a `last_timestep` table to keep track of the latest timestep in which a qubit was scheduled in an operation. Traversing instructions of a leaf module in program order, a `last_timestep` table lookup is performed for all operands of an instruction, since each operand may represent a data dependency. This instruction is then scheduled in the earliest timestep possible, resulting in an update in the `last_timestep` data for its operands. Once all instructions in a module are processed, its `last_timestep` data is stored, referencing each operand by its argument number for modular analysis.

For a non-leaf module, the algorithm proceeds in a similar manner, except that when a module invocation instruction parameterized with qubit arguments is encountered, the arguments are treated as its operands. The `last_timestep` table is examined to determine the earliest timestep the module can be scheduled to start. The values from the `last_timestep` table of invoked module are added to that of the invoking module, increasing the critical path length for the instruction’s operands by the pre-computed value. Once all modules are processed, the schedule length obtained is the estimated critical path length.

```plaintext
for each module M in post-order of program’s callgraph do
    for each instruction I do
        // Determine earliest timestep in which
        // the operation can be performed
        for each operand o; do
            get last_timestep[M][o]
        end
        lt = max(last_timestep[M][o])
    end
    if I is a quantum primitive then
        Schedule I in timestep lt+1
        //Update last_timestep table
        for each operand o; do
            last_timestep[M][o] = lt+1
        end
    else
        //I is a module invocation
        Schedule module to start execution in timestep lt
        for each operand o; do
            last_timestep[M][o] = last_timestep[I][o] + lt
        end
    end
end
```

**Algorithm 1:** Modular estimation of critical time.
### 3.6.4 Remodularization

Timing analysis of large-scale quantum programs requires scalable algorithms that make use of the modularity of a program to avoid repetitive analysis and improve analysis time.

However, this comes at the cost of decreased schedule optimality. For example, parallelism between module boundaries can be overlooked in non-flattened sequences of instructions.

Figure 3.10 depicts this loss of parallelism with an example. To strike a balance between modularity and optimizability, we perform remodularization of the input quantum program. The process involves inlining modules that are too small for optimization, into their respective call sites, and obtaining larger flattened modules.

![Module Toffoli(a,b,c) and Modular/Flattened Analysis](image)

**Figure 3.10:** The effect of a modular (hierarchical) critical path analysis. Lost parallelism in module boundaries causes longer reported critical paths, but the time to analyze is significantly improved.

We define a *Flattening Threshold (FTh)* for module size in terms of the number of quantum gates it contains. Informed about module sizes from resource estimation passes, a remodularization pass in ScaffCC flattens the modules that are smaller than the threshold; i.e. all the function calls contained within it are inlined. This results in leaf modules that...
consist of at most $FTh$ operations. Figure 3.11 shows how the estimated critical path gets better with more flattening, due to more inter-modular parallelism being discovered.

![Graph](image)

Figure 3.11: The effect of modularity on circuit critical time estimates for the case of the BWT($n=300$, $s=1000$) benchmark. More circuit flattening (modular inlining) causes shorter (closer to real) reported critical paths, at the cost of longer analysis time.

An appropriate flattening threshold determined empirically, such that it reveals the maximum amount of parallelism without being a major burden on scalability. We perform a characterization of our available applications for this purpose. Figure 3.12 shows the percentage of modules with gate counts (module sizes) falling within specified ranges for each application. This reveals the proportion of modules that could be flattened for a certain $FTh$. Based on these and other experiments, for the remainder of our studies we use $FTh$ set to 2 million operations. This flattens 80% or more of the modules contained within all applications except SHA-1. For SHA-1, we used a flattening threshold of 3 million which flattens the entire benchmark.

### 3.7 Related Work

Several compilers for quantum programming languages based on C [134] and Haskell [12, 68] have been previously proposed. ScaffCC differentiates itself through two objectives: generation of tractable quantum assembly code that is also amenable to aggressive low-level optimizations; and logical program analyses at full program scale. Similar to Quipper as proposed by Green et. al. [68], the ScaffCC compiler handles program scale by making heavy use of modularization. Additionally, ScaffCC recognizes the implications of the
Figure 3.12: Histogram of gate counts represented as percentage of total modules in benchmarks. Using a flattening threshold of 2M operations, 80% or more modules are flattened for all benchmarks except SHA-1. For SHA-1, a flattening threshold of 3M is used to flatten the entire benchmark.

degree of modularity on both efficiency and quality of compilation, and presents scalable techniques to achieve both.

Moreover, this chapter presents a first study in the trade-off between modular inlining and critical-time estimation accuracy. Although other papers envision the prospect of the QASM language being an extension of conventional classical assembly languages extended with a quantum instruction set [55, 136], to the best of our knowledge none have implemented large circuits using this format and studied the trade-offs between manageability and optimizability.

Previous work has enabled resource analysis as part of algorithm development [68, 134]. ScaffCC expands its analysis toolbox with other useful analyses such as entanglement and timing analysis. The analysis framework can be easily extended further. For example, Metodi et al. [123] propose a useful reliability analysis in circuits, whose results can be compared with the reliability goal of the hardware and used to determine circuit locations in need of error correction. Techniques for exact entanglement analysis have been previously proposed in [147, 149]. Perdrix [147] has developed typed language extensions for abstract interpretation of entanglements in quantum data arrays, while Prost and Zerrari [149] have proposed formal semantics for identifying entanglements in higher order functions. ScaffCC performs a conservative and modular entanglement analysis at a purely logical level. This is intended to aid both design and debugging of quantum algorithms, which benefit from an understanding of where entanglements are potentially created and removed in a program. Furthermore, Schuchman and Vijaykumar [161] identify a program transformation which exploits parallelism between computation and uncomputation portions of a program, albeit at the cost of increased qubits. This transformation can easily be added to ScaffCC due to its tracking of uncompute regions.
3.8 Future Extensions

Although ScaffCC performs many optimizations on the classical portion of the code, it lacks aggressive quantum-specific optimization and circuit shrinking passes. Below we describe several possible directions for future improvements to the ScaffCC (or any other) compiler. Several of these proposals are under active investigation as of this writing.

3.8.1 Circuit-Level Optimizations Using Gate Identities

Many quantum circuits can be shrunk if the compiler is aware of certain gate identities. For example, using two CNOT gates in succession is akin to an identity — the compiler can safely remove them from the circuit. Unfortunately there are countless possibilities of gate combination, and exhaustive approaches may be very expensive to check for each identity. A good approach is to do this offline, in the form of “gate libraries.” The compiler may then draw upon known identities from this library for circuit-level optimizations. Lin et. al. [115] have built such a library.

3.8.2 Decomposition in Terms of Other Gate Sets

Currently Scaffold decomposes every circuit into the Clifford+T standard gate set. While this is the most common choice, other gate sets are also typically used to express circuits, such as Toffoli gates instead of T, or the V basis. Having a choice of output allows comparison of results between different researchers.

3.8.3 Recycling Qubits and Reducing Circuit Footprints

An important distinguishing factor between classical and quantum compilation is the scope of variables. In classical computation, statically allocated registers can be freed up when the program exits a function’s scope. However, this is not necessarily true for quantum computation. Due to the phenomenon of entanglement, a qubit may get entangled, and in the absence of proper uncomputation, that qubit may not be freed. This significantly adds to the qubit usage of a program: accumulated garbage cannot be freed with guaranteed safety.

In Section 2.7 we proposed a language-level solution to this: using a “reverse” primitive in Scaffold, it is possible to allow the programmer to guarantee to the compiler that a qubit has been properly disentangled, giving permission to the compiler to reclaim that qubit and reuse it for other allocations.

A more automated mechanism greatly simplifies the task of programming, although giving correct guarantees must be sufficiently explored in future research. ScaffCC’s current entanglement analysis is conservative, meaning it will mark as entangled qubits that interact at any point, even if they have later been disentangled. A more thorough data path analysis may discover disentanglements as well.
3.8.4 More Intelligent Code Flattening

We have discussed the importance of code flattening in striking a balance between deep optimization (necessary for managing quantum resources) and manageability of compilation. However, our current flattening approach is a simplistic one which solely relies on the size of modules. Among other things, it fails to take into account the frequency of calls to a module (similar to hot code optimization), or its relation to the critical path of computation. The current procedure is also all-or-nothing: if a module is found to have less than $F_{Th}$ gates, it is completely flattened; whereas no flattening may occur if it is slightly larger than the threshold. To better see the inefficiencies that may arise, consider a very large module consisting of calls to many small modules — if the sum of all the children’s sizes is more than $F_{Th}$, no flattening is done, and we are left with many prohibiting small-module boundaries.

Better flattening heuristics can be explored. The goal of any heuristic approach must be to maximally dissolve module boundaries while respecting a certain budget for the size of each module after flattening. Given a module size budget of $B$ (determined by how much processing power is available to compiler and analyze large modules), it is favorable to achieve near-uniform $B$-size modules after flattening.

Therefore, an improved flattening heuristic may combine top-down flattening with bottom-up inlining: modules that are smaller than $F_{Th}$ in size would be completely flattened (all their children recursively inlined), while outstanding modules that are still small (below an Inlining Threshold ($I_{Th}$)) would be inlined into their parents’ call points.

However, module size is not the only metric for effective flattening. In particular, when faced with multiple options for flattening up to a certain budget, the option that yields the greatest benefit must be chosen. Module call frequency is one important indicator of module “importance.” Since analysis and scheduling on flattened modules yields better results, it is better to flatten high-frequency modules rather than low-frequency ones.

Another metric that may be taken into account for improved flattening is the parallelism factor of individual modules. Given multiple choices for flattening, it is favorable to flatten more parallel modules since they enjoy greater benefits when flattened. Of course, it is not possible to precisely know the parallelism factor of a flattened module before actually flattening it; therefore the flattening decision must be made using the parallelism factor of all of its children.

Finally, since the goal of flattening is to increase parallelism of operations, we must aim to flatten across the “width” of a circuit rather than its “depth.” Flattening across depth only increases the module size without adding any parallelism benefits. Therefore, a possible heuristic may select modules to be inlined using a round-robin algorithm on circuit qubits, whereby in each iteration, the next inlining priority is given to a module that covers the least recently covered qubit.

3.9 Chapter Summary

This chapter has examined the issues concerning the compilation and analysis of quantum circuits. Our study shows the applicability of many mature techniques from classical com-
pilation, while also highlighting the specific differences of the quantum domain to be considered. In particular, we recognize that quantum programs are almost always specialized to their certain inputs. This allows for aggressive optimization, at the cost of scalability. We showed that scalability must be managed at two levels: the code generation phase and the output format. We explored the dual classical-quantum nature of programs and how they can be exploited to help mitigate the problem of scalability. Finally, we illustrated several important use cases for a quantum compiler, which can help algorithm development, correctness checking, and resource estimation.
Chapter 4

Case Study: Characterizing the Performance Effect of Trials and Rotations in Applications that use Quantum Phase Estimation

4.1 Introduction

This chapter showcases how the ScaffCC compiler may be used to compile, analyze, and ultimately gain insight into the behavior of various quantum algorithms. We use the Quantum Phase Estimation (QPE) module as a point of focus in this case study.

QPE is a key technique in designing quantum algorithms for exponential speedup. Intuitively, QPE allows quantum algorithms to find the hidden structure in certain kinds of problems. In particular, Shor’s algorithm for factoring the product of two primes uses QPE. Simulation algorithms, such as Ground State Estimation (GSE) for quantum chemistry, also use QPE.

Unfortunately, QPE can be computationally expensive, either requiring many trials of the computation (repetitions) or many small rotation operations on quantum bits. Selecting an efficient QPE approach requires detailed characterizations of the tradeoffs and overheads of these options.

In this chapter, we explore three different algorithms that trade off trials versus rotations. We perform a detailed characterization of their behavior on two important quantum algorithms (Shor’s and GSE). We also develop an analytical model that characterizes the behavior of a range of algorithms in this tradeoff space.

This chapter characterizes the performance impact of QPE implementation choices on full-scale quantum applications. To accomplish this, we work with full QC apps and QPE methods written in Scaffold. Our evaluations of resource requirements and program runtimes are based on analysis performed as part of the ScaffCC compiler framework. ScaffCC synthesizes logical quantum circuits from the high-level language description. In this work we use the circuit model to show various ways of performing QPE. This is a shorthand

\footnote{The work in this chapter was performed in collaboration with Shruti Patil and others [144].}
method. In fact, resource estimates and algorithm runtimes are entirely generated by the ScaffCC compiler. When algorithms are drawn as circuits, the timesteps needed to complete the calculation is referred to as the circuit depth.

In our comparison of QPE schemes, we focus on logical qubits and logical timesteps. That is, we estimate costs before error correction is added. We do this because error correction overheads are heavily dependent upon technology and coding choices. We attempt to remain independent of those choices. Our logical comparison is quantitatively valid as long as all the alternatives used end up using the same amount of error correction per logical bit and logical operation. Our comparisons, however, are still qualitatively valid when trying to determine the QPE scheme with the lowest runtime. Adding error correction will maintain runtime relationships, since longer logical runtimes will experience equal or greater error correction overhead than smaller runtimes. Furthermore, although the bulk of resources in any quantum circuit implementation is due to added error correction redundancies, the algorithmic level (also referred to as the logical level) has very high leverage in controlling this. Algorithms with fast runtimes are less prone to the accumulation of errors and can tolerate fewer error correction steps. Thus, finding better high-level QC algorithms and optimizing their implementation efficiency pays off multiplicatively because of the further benefits that accrue in reducing quantum error correction required.

QPE is a core building block computation for various quantum applications such as solving systems of linear equations [71], prime factorization [165], and finding answers to quantum many-body problems [181, 191]. In fact, there are really only three core building blocks for useful quantum algorithms: QPE, quantum random walks [39] (used for some graph search algorithms), and Grover’s function [69] (used for Grover’s search algorithm).

Because of QPE’s central importance in many QC applications, characterizing its performance and identifying efficient implementation approaches is important. As one application example, consider Ground State Estimation (GSE). Finding the exact ground-state energy of a system of many particles is an intractable task for classical computers due to the exponential growth in computational cost as a function of the particles involved. However, using quantum systems themselves to store and process data, as is the case in quantum computers, this problem can be efficiently solved. Molecular energies are represented as eigenvalues of an associated Hamiltonian and can be obtained by encoding a molecular wave function into a set of quantum bits (qubits), simulating their evolution using quantum operations on those qubits, and then extracting the energy using quantum phase estimation. Experimental realizations of QPE for calculating chemical system energies have already been reported using linear optics [109].

The overall goal of QPE is to compute the eigenvalue of a quantum unitary circuit. This circuit is also referred to as the Oracle, since it can be regarded as a black box of computation, whose eigenvalue will reveal important information for solving the overall problem. Different applications have different oracle circuits of varying complexity, and this is a key factor in the tradeoff space of QPE techniques discussed in this chapter.

QPE approaches for obtaining phase estimations have commonly rested on two underlying techniques:

- **Quantum Fourier Transform (QFT)**: In this approach, qubits are first computed on via the oracle function, and then a QFT is performed to extract an eigenvalue. A
full implementation of QFT requires potentially many, exponentially small quantum rotation operations, which may be costly to realize with good precision.

- **Repetitive Trials**: Using repetitive trials involves applying the oracle several times, measuring samples from the probability distribution of the eigenvectors, and finding the final answer via classical post-processing of the answers; this reconstructs an estimate based on the collected measurements. This technique reduces the number of quantum rotation operations required (compared to QFT-based approaches) but may involve many trials in order to obtain an accurate estimate.

Given these two general implementation styles, specific QPE algorithms vary in their use of them. Table 4.1 shows a coarse-level comparison of three distinct approaches for Quantum Phase Estimation which cover a spectrum of possible methods:

- **Kitaev Hadamard Tests (KHT)**: The approach originally proposed by Kitaev [98] relies on a pre-determined number of trials to achieve a desired target for the error-rate and precision of estimation.

- **Approximate Quantum Fourier Transform (AQFT)**: This approach is based on QFT, but uses Approximate QFT [17] instead of the full QFT in order to curb the number of rotations.

- **Arbitrary Constant Precision Algorithm (ACPA)**: ACPA is an approach proposed by Ahmadi and Chiang [10] which offers a method between KTH and AQFT in terms of the number of rotations and trials involved.

This work provides the first detailed performance comparison of these QPE techniques within real, full-scale QC applications. Overall this work makes the following contributions:

- Through implementing different QPE methods, we perform an empirical workload characterization of the tradeoff between performing higher-precision rotation operations versus performing more trials for obtaining accurate phase estimates.

- We characterize two important quantum applications that use quantum phase estimation as a fundamental building block: Shor’s algorithm for prime factorization, and the Ground State Estimation algorithm for calculating molecule energy levels. We analyze the effect of different methods of QPE in each of these algorithms.

- We find that practical resource constraints limit parallel execution within our QPE methods, heavily influencing which methods have the lowest runtime.

- We also find that the application also heavily influences the appropriate QPE method, since some methods require more redundant execution of the application than others.

The rest of this chapter is organized as follows: Section 4.2 describes QPE in detail, and Section 4.3 describes the different implementation approaches we consider. Section 4.4 analyzes the resulting design space tradeoffs. Section 4.5 derives key application characteristics from Shor’s algorithm and Ground State Estimation. Section 4.6 discusses related work, and Section 4.7 summarizes the chapter.
<table>
<thead>
<tr>
<th>Number of Rotations</th>
<th>Number of Trials</th>
</tr>
</thead>
<tbody>
<tr>
<td>KHT</td>
<td>None</td>
</tr>
<tr>
<td>AQFT</td>
<td>Many</td>
</tr>
<tr>
<td>ACPA</td>
<td>Some</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of three QPE Methods, with the idealized assumption of perfect rotation precision. (With more realistic imperfect rotation precisions, AQFT will have a non-zero number of trials since it uses repeated trials to regain precision.) ACPA is a middle ground between KHT and AQFT in terms of the number of rotations and trials involved.

4.2 Quantum Phase Estimation

If a quantum unitary operator $U$ acting on an $m$-qubit input vector yields:

$$U |u\rangle = e^{i2\pi\varphi} |u\rangle,$$

then $|u\rangle$ is said to be an eigenstate (or eigenvector) of the corresponding unitary matrix of $U$. In this case, the eigenvalue is a phase shift $e^{i2\pi\varphi}$ that is introduced in the state vector. The goal of QPE is to estimate the value of phase $\varphi$ in the eigenvalue for a given unitary quantum circuit $U$. This phase value reveals important information about the hidden structure of the quantum oracle function in algorithms such as order finding, the core computation of Shor’s algorithm.

Generally, estimating $\varphi$ is a challenging task. Two factors affect the adequacy of estimation: The precision $\delta$ determines how close to the actual eigenvalue the estimation outcome is. The error rate $\epsilon$ determines how likely it is, given the probabilistic nature of quantum states, for the final estimate to have the desired precision (equivalent to $1 - \text{Prob(success)}$).

We use $\tilde{\varphi}$ to denote an estimate of $\varphi$. Since the overall phase shift lies in the $[0, 2\pi)$ interval, an estimate of $0 \leq \varphi < 1$ with $n$ bits of precision is written as:

$$\tilde{\varphi} = 0.x_1x_2x_3...x_n$$

(4.2)

Where $0.x_1x_2x_3...x_n$ is a notation for $\sum_{i=1}^{n} x_i \times \frac{1}{2^i}$ and $x_i \in \{0, 1\}$.

4.2.1 Quantum Rotation Decomposition

There are an infinite number of single-qubit rotations that can be done on a quantum state, but we can implement a finite subset on a physical computer. A major cost in some of the QPE approaches arises when algorithms must decompose arbitrary rotations—especially precise small rotations—into a sequence of physical operations.

Decomposition is possible due to the universality of a small set of quantum gates for representing any single-qubit operation. For example, Hadamard (H), Phase (S), Controlled-NOT (CNOT) and the $\frac{\pi}{8}$-gates (T) form a universal set [30]. If we compare applying the decomposed set of gates instead of the actual (perfectly precise) rotation to a
particular qubit state, the decomposition precision indicates how closely the approximate outcome state would resemble the exact state (i.e., the likelihood that measuring it would yield the same answer.)

For an arbitrary angle, we need an approach for determining the correct decomposed sequence. In this chapter we employ the Single Qubit Circuit Toolkit (SQCT) proposed by Kliuchnikov et al. [99], which offers practical decompositions of up to accuracy \(10^{-15}\), based on the results of [100].

For the purpose of quantum phase estimation, we are interested in rotation angles used by the quantum Fourier transform, which are of the form \(R(\frac{2\pi}{2^k})\), and the tradeoffs that are present in limiting their precision. Figure 4.1 shows the length of decomposition obtained from SQCT [169] for these rotation operations. Higher precision decompositions can be directly linked to a longer gate sequence, although at a particular precision level, there is no significant difference between different angles.

Using smaller angles in an algorithm means that the precision of decomposition must be increased, in order to have a faithful approximation of those rotations. Our model (Section 4.4) assumes that the presence of angle \(\theta = \frac{2\pi}{2^k}\) means that decomposition should take place at least with precision:

\[
|\theta - \bar{\theta}| < \frac{1}{2^k}.
\]  

(4.3)

The lower dotted line shows this trend, and guides the actual precision that we should pick in different scenarios, depending on the degree of the smallest angle present. This increase in precision would affect the approximation of all gates, thus being costly for circuit size.

### 4.3 Implementations of Quantum Phase Estimation

This section describes the three QPE methods and their relative strengths and weaknesses. Overall, they span a design space that presents tradeoffs of runtime (circuit depth), operation-level parallelism (circuit width) and resource usage (number of qubits and gates). They also implicitly differ in the accuracy with which they estimate the phase of the input eigenvector. In order to make the approaches comparable for our characterization, we set each of their success probabilities \((1 - \text{error rate})\) to 0.8 in our experiments, i.e. an 80% total probability of obtaining the correct outcome.

#### 4.3.1 Kitaev’s Hadamard Test with S gate (KHT)

Kitaev [98] proposed an algorithm that avoids arbitrary quantum rotations, which expand into long sequences of physical operations as shown in Figure 4.1. Unfortunately, the algorithm uses a large number of independent trials to perform QPE to required precision. These trials can be parallelized, but can require an impractical amount of quantum hardware to do so. Our results in Section 4.4.3 will elaborate upon this.
Kitaev uses a series of Hadamard tests to perform the phase estimation, each of which requires a single phase-shift gate $S$:

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$$

(4.4)

This matrix is a transformation on a qubit vector $|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$ as previously described in Section 4.2. The circuit that accomplishes Kitaev’s Hadamard Test is shown in Figure 4.2. When applied to the $|0\rangle$ qubit (no probability of measuring “1”), the Hadamard gate ($H$) results in a qubit that has equal probability of $|0\rangle$ and $|1\rangle$. The Hadamard gate is the inverse of itself and is often applied again before measurement, as in the Hadamard test. Each Hadamard test estimates one bit of the phase $\varphi$ with a precision of $\frac{1}{16}$ [95]. The accuracy at which we are computing the estimated phase $\tilde{\varphi}$ can be expressed as:

$$Pr(|\varphi - \tilde{\varphi}| < \frac{1}{2^n}) > 1 - \epsilon$$

(4.5)
To achieve a desired success rate, that is to succeed with $1 - \epsilon$ probability, we need $m_{KHT}$ trials of Kitaev’s Hadamard test for each bit [10], where

$$m_{KHT} = 55 \ln \frac{4n}{\epsilon} \sim 55 \ln(20n) \quad (4.6)$$

In the above equation, we set $1/\epsilon$ such that the success probability $1 - \epsilon = 0.8$. This gives us the value of $m_{KHT}$ that achieves an overall success probability equivalent to that of QFT. The estimation of each bit in Equation 4.2 occurs independently of the other bits. Therefore the circuit requires no controlled phase operations. This also allows many (or all) bits to be estimated simultaneously, subject to the resource constraint that enough qubits are available for the individual Hadamard tests. Thus, KHT has two levels of parallelism, fine-grained parallelism (analogous to instruction-level parallelism) within the computation in Figure 4.2, and coarse-grained parallelism (analogous to task-level parallelism) between trials. Indeed, all the approaches have coarse-grained parallelism but only KHT has fine-grained parallelism across bits.

![Kitaev’s Hadamard Test with S Gate](image)

Figure 4.2: Kitaev’s Hadamard Test with S Gate (time flows from left to right in this circuit diagram)

### 4.3.2 QFT and Approximate QFT (AQFT)

Because KHT requires a large number of trials, a more common implementation of the QPE algorithm is based on Quantum Fourier Transform (QFT), shown in Figure A.4. QFT (and Approximate QFT) trade the expense of arbitrary quantum rotations for fewer trials in achieving desired QPE accuracy.

In the QFT approach, $n$ ancilla qubits are used to form the upper register in the circuit, while $n_{LR}$ qubits form the eigenstate in the lower register, whose periodicity (phase) is to be estimated. The last step of QPE is implemented using the inverse quantum Fourier transform ($QFT^\dagger$) on the ancilla states. The inverse QFT module will cause the probability distribution of the state of the upper register to be clustered around the correct phase value. With a Hadamard gate and a measurement, an estimated bit is revealed. A typical circuit for an $n$-qubit $QFT^\dagger$ is shown in Figure 4.4. The estimated phase is computed from its least to its most significant place. Proceeding one qubit at a time, the estimation accuracy of higher significance qubits increases by taking into account the estimates of all previously determined qubits. This approach provides a success probability of at least $\left(\frac{4}{\pi^2}\right)$ [17]. When $\phi$ is an exact multiple of $1/2^n$, the success probability is 1.

The QFT circuit requires $O(n^2)$ rotations with degree up to $e^{i2\pi/2^n}$. In reducing this complexity, Barenco [17] showed that the lower bound of success probability of QFT can
be achieved with fewer rotations per qubit, by considering the estimates of only the last \(k\) (also referred to as \(k_{AQFT}\)) qubits instead of all previous qubits. This approach achieves an asymptotic probability of \((4/\pi^2)\) when \(k > \log_2 n + 2\) is selected [35], thereby reaching the lower bounds of QFT success guarantees. This bounds the number of rotations to \(O(n \log_2 n)\) limiting their degrees to \(\exp^{2\pi i/2k}\). In practice, due to the logarithmic reduction in circuit length, AQFT provides a viable alternative that performs just as well (and sometimes better) than QFT in the presence of decoherence [17]. Therefore, we choose to use the AQFT circuit for our comparison.

A better lower bound on the success probability of AQFT has also been derived as \((4/\pi^2 - 1/16)\) [35]. When estimating an exact n-bit phase, the success probability of AQFT also approaches 1. However, these analyses assume that the rotations are precisely applied. In reality, rotations can only be achieved up to a precision factor (Section 4.2.1 discusses this approximation), which reduces the overall success probability. We will discuss this impact in the following section, in the context of both the AQFT method and the ACPA method which is described next.

![Figure 4.3: n-qubit QPE circuit based on QFT to estimate the phase \(\varphi = 0.x_1x_2x_n\).](image)

![Figure 4.4: n-qubit \(QFT^\dagger\) circuit. This circuit requires rotations of exponentially increasing precision with the number of qubits.](image)
Figure 4.5: Limiting the precision of rotations to achieve the \( n \)-qubit \( AQFT^\dagger \) circuit to estimate the phase \( \varphi = 0.x_1x_2...x_n \), where \( k \geq 2 + \log_2(n) \) is the highest degree of rotation applied. This is the same circuit used in ACPA, but with different requirements for the value of \( k \).

### 4.3.3 Arbitrary Constant Precision Algorithm (ACPA)

The Arbitrary Constant Precision Algorithm (ACPA) [37] bridges the KHT and AQFT approaches, providing a design space where the number of arbitrary rotations can be reduced at the expense of more trials. Intuitively, AQFT performs a series of smaller and smaller (higher-precision) rotations to reach a desired accuracy. ACPA omits some of the smallest rotations at the expense of accuracy. This loss in accuracy is then recovered by performing more trials and then performing a majority vote on the classical bits resulting from measurements (the result is binary).

If the degree of rotation is limited to \( e^{2\pi i/2^k} \) where \( k \geq 3 \) (\( k \) also referred to as \( k_{ACPA} \)), the number of trials required is:

\[
m_{ACPA} = \frac{2\ln(1/e')}{\pi^2(1 - \frac{n}{2k-1})^2}
\]

Here, \( 1 - e' \) is the success probability of estimating each bit. The overall success probability obtained is \( 1 - ne' \). However, the above expression also assumes that the rotations are perfectly implemented. Since practical rotations are imperfect, they increase the number of required trials to reach the desired accuracy. [37] derives the effect of imperfect rotations on the number of trials in ACPA. In particular, if the rotations can be achieved with an accuracy of at least \( \eta = \frac{1}{(k-1)2^k} \), the number of trials is given by:

\[
m_{ACPA} = \frac{2\ln(1/e')}{\pi^2(1 - \frac{n}{2k-3})^2} \sim \frac{2\ln(5n)}{(1 - \frac{n}{2k-3})^2}
\]
We use this equation to derive the number of required trials. Here, we set $1 - n\epsilon' \geq 0.8$, giving us $\epsilon' \sim 1/5n$ to accomplish an 80% success probability.

One can consider the AQFT approach as the limiting case of ACPA, i.e. when $k_{ACPA}$ is chosen to be $> 2 + \log_2 n$, the two approaches implement the same circuit. Therefore, we can compute the number of trials required for the AQFT circuit as:

$$m_{AQFT} = \frac{2\ln(5n)}{\pi^2} \cdot \left(1 - \frac{1}{2^{k-3}}\right)^2$$

4.3.4 Phase Kickback ($U_f$)

Finally, looking at the circuits for all the QPE methods (Figures 4.2-4.5), we see that for any type of quantum phase estimation with accuracy of $n$ bits, the range of quantum operations $U^{2k}$ with $0 \leq k < n - 1$ must be applied. The modules of $U$, $U^2$, $U^4$, ..., $U^{2n-1}$ are collectively called the phase kickback function and we refer to it as $U_f$. Figure 4.6 depicts the phase kickback for the case of the AQFT circuit. (The first stage of AQFT and ACPA are indeed equivalent to one application of $U_f$; KHT splits $U_f$ across its different trials.) $U_f$ has in many cases its own interpretation—for example, in Shor’s algorithm it is equivalent to a modular exponentiation module, raising its input to the power of a fixed number, modulo the number to be factored.

4.4 Analysis of the Design Space

This section lays out the design space options in detail, and evaluates the tradeoff between QPE techniques, providing runtime results for different precision requirements and resource constraints.
Our analysis is divided into two categories. First, we have factors that constrain the number of trials that can be executed in parallel (task-level parallelism). These include the number of qubits as well as the number of available eigenstates in each application with QPE approach. These resource requirements affect how many parallel trials can be supported by the hardware. Second, we have a factor that affects the execution time of each trial. This is the size of the phase kickback function ($U_f$) that must be called across the many trials, thus contributing to the circuit size generated by the implementation.

### 4.4.1 Factors Affecting the Parallelization of Trials

These factors actually can also affect parallelization within trials, but we can assume that resources are sufficient for supporting parallel execution within at least one trial.

#### Number of logical qubits

Parallel execution of multiple trials will require multiple copies of quantum state, which we measure in logical qubit count. (The number of physical qubits is larger due to QEC.) Since arbitrary quantum data cannot be copied (the quantum no-cloning theorem), independent trials can only occur if they start from the beginning of the entire application or from an intermediate computation that starts with only freshly initialized qubits.

The maximally-parallel implementations require the most qubits since they require a copy of the entire computation for each trial. In general, if $p$ is the parallelization factor achieved during an execution, the number of qubits required by the three methods is given by: $NQ = p \times (n_{LR} + 1)$. Since the maximum parallelism available in the algorithms is across the trials and the bits, the number of qubits required can be as high as $mn(n_{LR} + 1)$ in maximally parallel implementations. In practical implementations, $p$ ranges from 1 to $mn$. For example, KHT may be applied with $p = m$ such that the bits are processed serially while the $m$ trials are performed in parallel. This reduces the requirement to $m_{KHT}(n_{LR} + 1)$ by reusing expensive qubit resources between successive bit estimations.

#### Number of eigenstates available

To estimate the eigenvalues through multiple trials, the phase estimation algorithms must be applied to the same eigenstate a number of times. For the trials to be parallelized, multiple copies of this eigenstate must be available. In general, the parallelization of trials is constrained by the number of such copies available. While this can be bounded by the number of available qubits in the system, in some cases it may simply be too expensive to prepare multiple copies of the states. However, for many applications including Shor’s and GSE, these eigenstates can be generated by applying $U_f$ to suitable initial states which are straightforward to prepare. Therefore, our study assumes that with sufficient resource availability, the eigenstates can be prepared in parallel.
### 4.4.2 Factors Affecting Circuit Size

In addition to the specific computations described for each QPE approach in Section 4.3, the size and implementation cost of the $U_f$ function is an important factor. Section 4.5 considers the cost of varying the size of $U_f$. Here we examine the implementation cost by studying the number of invocations of the $U$ modules. The higher the invocation count of $U$, the greater the execution time of each trial and the greater the resources needed to support parallelism across trials.

The $U_f$ function performs a central transformation in the QPE algorithm in creating the eigenstate whose periodicity is to be estimated. For many algorithms, the $U_f$ function is also the most expensive part of the circuit. Therefore the cost of its implementation is an important consideration and provides an insight into its complexity. In general, the function $U^k$ can be implemented using $k$ copies of $U$. Hence, the number of invocations of $U$ is given by:

$$N_U = m \times (1 + 2 + 4 + \ldots + 2^{n-1}) = m \times (2^n - 1) \quad (4.10)$$

In our characterizations, the GSE algorithm uses this scheme to implement $U_f$. However, for large $n$, this can result in an undesirably high number of invocations of $U$. For example, in Shor’s algorithm, $n$ is expected to be about 1000. Therefore, a more efficient implementation of $U^k$ where the cost stays constant regardless of the value of $k$ is desirable, and has been proposed for Shor’s algorithm [145]. If such implementations are available for $U$, the number of invocations of $U$ are $N_U = m \times n$. For both types of implementations, the smaller the number of trials, the fewer the invocations of $U$, consequently, one can expect better overall runtime.

Table 4.2 shows a simplified comparison of the resources and runtime for the three methods, assuming that only the trials are parallelized. Choosing between Kitaev’s algorithm and others depends on the relative cost of $U_f$ and rotations, while choosing a suitable technique between that of ACPA and AQFT depends on the number of trials, number of rotations and the cost of the rotations. In Section 4.5, we determine these costs empirically.

<table>
<thead>
<tr>
<th></th>
<th>KHT</th>
<th>ACPA</th>
<th>AQFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gates</td>
<td>$m_{KHT}(U_{f_{gates}} + 3n)$</td>
<td>$m_{ACPA}(U_{f_{gates}} + n.(k_{ACPA} - 1) .Avg_{R_{gates}})$</td>
<td>$m_{AQFT}(U_{f_{gates}} + n.(k_{AQFT} - 1) .Avg_{R_{gates}})$</td>
</tr>
<tr>
<td>Time</td>
<td>$U_{t_{steps}} + 3n$</td>
<td>$U_{t_{steps}} + n.(k_{ACPA} - 1) .Avg_{R_{t_{steps}}}$</td>
<td>$U_{t_{steps}} + n.(k_{AQFT} - 1) .Avg_{R_{t_{steps}}}$</td>
</tr>
<tr>
<td>Qubits</td>
<td>$m_{KHT}(1 + n_{LR})$</td>
<td>$m_{ACPA}(1 + n_{LR})$</td>
<td>$m_{AQFT}(1 + n_{LR})$</td>
</tr>
</tbody>
</table>

Table 4.2: Analytical comparison of the three techniques with respect to number of gates, circuit depth and number of qubits, assuming that the trials are performed in parallel while bits are processed serially.

---

65
4.4.3 Empirical Evaluation

To evaluate the different design metrics for the QPE implementations, we first limit the circuit to a micro-benchmark whose oracle is substituted by a proxy circuit consisting of simple CNOT gates. The intention is to explore the tradeoff space of QPE methods independent of oracle costs, which can vary significantly across applications. In particular, we show that resource constraints can limit parallelism and heavily influence the choice of QPE implementation. We later generalize our evaluation by plugging in realistic oracle costs and QPE precision targets based on parameters derived in Section 4.5.

Results were obtained using the ScaffCC compiler framework, which can measure resource usage in terms of logical qubits and logical gates, as well as compute runtime in terms of logical timesteps. The compiler can generate these measurements through static analysis because quantum applications are generally compiled with all inputs known.

The Effect of Desired Precision on Trials and Rotations

As described previously, the number of trials varies for the different methods for a given \( n \), the desired QPE precision in bits. Figure 4.7 shows how estimating more bits of QPE precision causes a sharper increase in the required number of trials in KHT as opposed to AQFT. The graph also depicts the impact of \( k_{ACPA} \) (the highest-precision rotation) on the ACPA method. The value of \( k_{ACPA} = 3 \) is insufficient to generate a high enough success probability in each trial demanding considerably large number of trials. For \( k_{ACPA} \) of 4 or higher, its behavior more closely tracks AQFT. On the other hand, using even a few controlled rotations helps the estimation process more than KHT, which does not take into account the values of the previously estimated bits.

![Figure 4.7: Number of trials for KHT, ACPA-k and AQFT. Trials for ACPA-k are computed from Equation (4.8) assuming imperfect rotation gates.](image)

66
Tradeoff between Resources and Runtime

A key tradeoff in our QPE schemes arises when parallelism is constrained by practical constraints in the number of logical qubits that can be implemented in a quantum machine. Specifically, the runtime of each QPE scheme is heavily dependent upon how much concurrency can be physically supported in executing independent trials.

Figure 4.8 compares the runtime of algorithms as we bound the number of logical qubits for the QPE implementations requiring 64-bit precision of phase estimates. Low availability of qubits forces a serial execution of the algorithms, driving up their runtimes. With more resources, the runtime of each algorithm improves as their inherent parallelism in the algorithms gets exploited. For KHT, the trials as well as the bits can be processed in parallel; however exploiting this parallelism requires $\sim 10^6$ logical qubits. (With error correction, this could easily be $\sim 10^8$ physical qubits, an impractical number for the foreseeable future.) On the other hand, for ACPA and AQFT, the truly serial parts are the rotations that require bit-wise processing, thus their parallelism can be exploited with fewer available qubits. When a low number of qubits is available, ACPA executes the fastest among the three. When available logical qubits becomes greater than $10^6$, all algorithms can be fully parallelized, with KHT offering the best runtime.

![Figure 4.8: Space-time tradeoffs presented by the three methods with bounded resource availability. With fewer available resources, ACPA executes fastest, while KHT executes fastest when available resources are as high as $\sim 10^6$.](image)

Resource Constraints with a Larger Oracle Function

We can extend our microbenchmark by increasing the cost of the oracle function from a simple CNOT gate to the oracle for Shor’s algorithm and GSE. Figures 4.9 and 4.10 show the runtimes for the three methods when the runtime of $U_f$ function is $\sim 10^7$ and $\sim 10^9$ timesteps (order of the size of $U_f$ for Shor’s algorithm and GSE, respectively). Desired
Figure 4.9: Space-time tradeoffs with $U_f$ and desired precision set for Shor’s algorithm. With fewer available resources, AQFT executes fastest due to fewer invocations of expensive serialized $U_f$.

Figure 4.10: Space-time tradeoffs with $U_f$ and desired precision set for GSE.
4.4.4 Summarizing the Analyzed Tradeoffs

This section characterized the tradeoffs of different QPE techniques and the design space that they create. While the KHT method is beneficial in terms of parallelism and hence runtime performance with ample resources, it incurs a large cost in terms of qubits and gates. In all metrics, ACPA is a middle solution to the extreme requirements of KHT and AQFT. Ultimately, the amount of resources at hand (especially logical qubits), the level of parallelism supported by the architecture, and the target runtime will determine the method of choice. QPE is a fundamental component of two very important QC applications: Shor’s factoring algorithm and GSE. Using the specific characteristics and parameters from these algorithms, our results here have also detailed the tradeoffs regarding how best to implement QPE for these full applications.

4.5 Case Studies

In this section, we derive the QPE target precision and oracle size for two algorithms that have quantum phase estimation as a fundamental component: Shor’s algorithm for prime
factorization, and the Ground State Estimation (GSE) algorithm for finding the ground energy level of a molecule. The fundamental difference between these algorithms (and generally all algorithms involving QPE) is the oracle $U$ function that is used for their implementation.

4.5.1 Case Study 1: Shor’s Algorithm

Factoring a large number into its prime constituents is a hard problem for classical computers, and forms the basis of many modern cryptography techniques [52, 153]. A seminal quantum algorithm by Shor [165] offers exponential speedup over the current best known classical algorithm [113]. Quantum phase estimation lies at the heart of Shor’s algorithm. We have implemented an optimized phase kickback function $U_f$ for this algorithm, as proposed by Pavlidis and Gizopoulos [145].

In this algorithm, the overall phase kickback module $U_f$ implements a modular exponentiation function:

$$|x\rangle |1\rangle \xrightarrow{U_f} |a^x \mod N\rangle$$

(4.11)

Referring to our discussion of $U$ and $U_f$ modules in Section 4.3.4, here the $U$ modules are equivalent to modular multiplication units, while the overall phase kickback module $U_f$ implements a modular exponentiation function. From modular arithmetic it is known that
4.5.2 Case Study 2: The Ground State Estimation Algorithm

The ground state estimation algorithm is used for estimating the ground state energy $E_0$ of a molecule [191]. Knowing the energy to lie in an interval $[E_{\text{min}}, E_{\text{max}}]$, this can be reduced to estimating the phase $\varphi$ in the equation

$$ U\ket{\psi_0} = e^{i2\pi\varphi} \ket{\psi_0}, $$

Figure 4.13: Comparing the total number of gates of GSE using KHT, AQFT and ACPA methods when the required precision of estimated phase is varied.

A repeated application of modular multiplication yields modular exponentiation. That is:

$$ a^x \mod N = (a^{2^0 \mod N})^{x_0} + \ldots + (a^{2^{n-1} \mod N})^{x_{n-1}}, \tag{4.12} $$

where $x_0, \ldots, x_{n-1}$ are the bits of $x$, the upper register of phase estimation.

Our implementation of this algorithm is efficient in the sense that there is a polynomial cost associated with the phase kickback module $U_f$ as a function of $n$, because each of the $U^{2^k}$ modules is customized separately and does not result from exponentially many calls to the $U$ module. Analytical analysis yields $n \times (796n_{LR}^2 + 692n_{LR})$ number of gates and $n \times (1045n_{LR} - 38)$ number of timesteps for the phase kickback circuit [145], where $n_{LR} = n/2$ is the size of the lower register of phase estimation. These values can be used in the analytical model of Table 4.2 to yield the overall cost of implementing Shor’s algorithm with the three QPE methods.

Figure 4.11 shows this cost as a function of the size of the phase kickback module $U_f$. A particular problem size (factoring a 32-bit number with 64 bits of precision) has been marked on the graph.
with $U$ being dependent on the system Hamiltonian $H$, among other things:

$$U = e^{iE_{\max} \tau} e^{iH \tau}. \quad (4.14)$$

Due to the complexity of implementing the $U^{2^k}$ subcircuits, the subcircuit $U$ is approximated by an $\text{ApproxU}$ module. We implement the algorithm for a molecule of molecular weight 10, with up to 12 bits of precision for the energy estimate, and with simplified assumptions for $\phi$. It is difficult to obtain a closed-form analytical expression for the number of gates and timesteps in the phase kickback module. However, from our algorithm implementation we can obtain empirical values for these numbers with varying precision. These values are plotted in Figure 4.12. This implementation, unlike that of Shor’s, does not result in a polynomial function for the size of the phase kickback module. The lack of customized and efficient specifications of $U^{2^k}$ circuits has resulted in exponential growth from repetitive application of the $U$ module, as mentioned in Eqn. 4.10.

Figure 4.13 shows the total number of gates as a result of varying the required precision in the overall GSE algorithm, showing the exponential growth effect.

### 4.6 Related Work

Efficient decomposition of single-qubit unitaries has been widely studied. The Solovay-Kitaev algorithm [96] is traditionally well-known for this purpose, and an early implementation by Dawson and Nielsen [47] outputs a sequence of order $\log^{3.97}(\frac{1}{\epsilon})$ gates for every rotation approximated with $\epsilon$ precision. Selinger [162] proposes an improvement which decomposes $Z$-axis rotations into a sequence of length $4 \log_2(\frac{1}{\epsilon})$ in the worst case. Paetznick and Svore [139] include non-determinism in their algorithm to achieve gains in circuit cost, while Bocharov et al. [27] improve this by moving away from the traditional decomposition bases of H, S, CNOT, and T gates.

Abrams and Lloyd [6] were the first to notice quantum phase estimation can be used in estimating the ground state energy of a molecule. Improvements on making this method faster have since been proposed [143].

Finally, Svore et al. [176] propose an improvement over the KHT method which makes asymptotic improvements on its circuit width and depth, by modifying the Hadamard tests to infer multiple bits of the phase simultaneously instead of one bit at a time. However, this requires extra invocations of the $U$ function, which is a significant cost factor in QPE implementations as per our findings.

### 4.7 Chapter Summary

This chapter has examined the tradeoff of three different methods for quantum phase estimation. These methods on the surface differ in the extent to which they use rotation gates and trials to accomplish the goal of phase estimation, but resource-wise they create a design space which requires different amounts of algorithm runtime and quantum space (or qubit usage). We performed an analytical as well as empirical study of the tradeoffs present in this design space, and discussed the effect of both hardware constraints and the type of
algorithm that uses the phase estimation in guiding the choice of a suitable QPE method for different quantum applications. In particular, we find that ACPA provides a good tradeoff between the number of trials and the number of rotations, often achieving a sweet spot in execution time when parallelism is limited by available physical resources.
Chapter 5

Compiler Management of Communication and Parallelism

5.1 Introduction

While quantum error correction codes can protect qubits against decoherence, fast execution time is still the best defense against decoherence, so efficient architectures and effective scheduling algorithms are necessary.

This chapter is concerned with ways to leverage the maximum amount of parallelism possible in quantum applications. It introduces architecture designs and scheduling algorithms that leverage data- and instruction-level parallelism, focusing especially on qubit communication as a major source of delay and error.

Although the QC algorithms we use as benchmarks offer dramatic reductions in computational complexity, it is also important to optimize their actual runtimes. Quantum bits (qubits) are highly error-prone, susceptible to decoherence over time, and require expensive Quantum Error Correcting (QEC) operations for reliable computation. Accelerating quantum computation is important because it allows the computation to “stay ahead of the errors.”

In this chapter, we propose the Multi-SIMD(k,d) architecture for quantum computation. Multi-SIMD approaches combine operation-level parallelism (with \( k \) different gates or operating regions) along with data-level parallelism (up to \( d \) qubits can be operated on within a single gate). We show that QC machines of this type are well-suited to the parallelism and computational needs of large-scale QC benchmarks. In addition, we explore parallelism and locality scheduling techniques as a means to optimize the performance of quantum computations onto Multi-SIMD hardware.

A natural source of state degradation in this architecture is communication delay, due to the fact that qubits need to be moved between regions in order to undergo an operation or to interact with other qubits (to support data and instruction parallelism). QEC requires many physical qubits to represent a single logical qubit. This makes compute regions and memories large in area, which results in large communication distances. When we add support for operation- and data-level parallelism, distances become even larger.

\(^1\)The work in this chapter was performed in collaboration with Jeff Heckey, Shruti Patil and others [7].
These high communication costs push us towards a communication-centric scheduling model. Exploiting the “mostly-serial” nature (at the operation level) of many of our applications, we adopt a Longest-Path-First scheduling (LPFS) strategy that keeps operands in-place for computations. Small scratchpad memories can further avoid global communication by capturing some temporal locality. In fact, the scratchpad memories are critical to leveraging the locality generated by LPFS, since it is often the case that one operand stays in a region for the next operation, but other operands must temporarily be moved aside. To maintain scheduling quality while reducing analysis times, we have developed a hierarchical approach that performs fine-grained scheduling in leaf modules with higher-level coarse-grained scheduling to stitch them together. Our results show an increase in speedup when incorporating communication from 3% to 308% and when incorporating local memories from 3% to 64%. With both modifications, total speedups can reach 9.83X.

We make the following contributions:

- To exploit parallelism while mitigating control complexity, we propose a novel Multi-SIMD(k,d) quantum architectural model. Ion trap technology with microwave control provides a practical basis for our model.

- We propose a hierarchical scheme to apply the scheduling algorithms to large-scale QC benchmarks. Through remodularization and application of the algorithms at coarse-grained and fine-grained levels, we show that our algorithms can effectively analyze large-scale QC benchmarks and generate schedules that are comparable to their estimated critical path lengths.

- We propose and evaluate communication-aware scheduling algorithms that reduce communication overheads while maximizing parallelism. We compare a traditional scheduling algorithm to the designed algorithm that considers the structure of quantum programs, and evaluate the speedup of both algorithms over sequential execution and naive communication models.

- We analyze the benefits of scheduling for local, scratchpad memories next to each compute region in reducing long communication delays to the global memory and overall execution speedup.

The rest of the chapter is organized as follows: Section 5.2 discusses the technology and architectural model proposed. Section 5.3 discusses our scheduling algorithms in depth. Section 5.4 discusses the schedules found. Sections 5.5 and 5.6 discuss related work, future work, and final conclusions.

### 5.2 Multi-SIMD Architectural Model

We propose Multi-SIMD(k,d), a family of quantum architectures motivated by practical constraints in controlling quantum technologies. Multi-SIMD architectures support parallelism at both the operation and data levels. In this section, we describe the concept, feasibility and challenges of this architecture.
5.2.1 Motivation Behind SIMD-Style Computation

Our work is motivated by trapped-ion quantum computing which is a leading candidate in viable quantum computing. However, our logic also applies to other technologies as discussed in 5.2.2.

In trapped-ion systems, operations are typically performed by focusing lasers on each ion, which represents a physical qubit. Because these lasers must have precisely tuned relative phase, they can be extremely large in size, currently the size of a small room, limiting current visions for QCs to roughly 6-12 lasers.

However, recent experiments with microwaves demonstrated multi-qubit control that could scale from 10 to 100 ions for a single microwave emitter [163]. Another level of fanout could create a small number of SIMD regions each capable of performing a different operation (“gate”) on roughly 100 to 10,000 qubits. Microwaves are a natural choice for driving single qubit gates and coupled with gradients they can also drive two-qubit gates [83, 137]. The signals for different regions can be tuned to cancel microwaves for some regions, effectively shielding some ions from neighboring signals [45].

Figure 6.4 shows the block diagram for our proposed architecture. Based on ion trap technology with microwave signaling mechanisms, we propose to build a scalable system with multiple independent operation regions, each offering gate-level parallelism. This greatly mitigates the complexity of control. Within each SIMD region, a large amount of data-level parallelism is available—100 - 10,000 qubits. In this manner, the architecture can support $k$ independent regions each allowing $d$ bits of SIMD computation, where $k$ is limited by the number of microwave signals and the complexity of generating interference to shield neighboring regions. The value of $d$ is limited by the ability to fan out the microwaves signal to all $d$ qubits without disrupting the rest of the system.

The $k$ SIMD regions serve two purposes. First, they allow active computation on $d$ qubits per cycle. Second, when idle, they can be used as passive short-term storage for qubits between compute operations. In later variations we provision each SIMD region with a small amount of scratchpad memory (or local memory) for temporary storage of qubits during intermediate computations on other qubits in the same region, similar to a traditional hierarchical memory system.

The global memory is distributed next to each region to provide better locality. A first touch initial placement scheme of qubits maps all qubits in the system to the global memory segment closest to where they will be used first. It is also assumed that qubits can “cool” in global memory if needed, i.e. lose entropy and return to the $|0\rangle$ state.

During a scheduled run of the benchmark, active qubits must be moved between SIMD regions and idle qubits must be moved to the global memory. Storing some qubits in the local memories can potentially reduce overall communication cost to and from the global memory. Such data motion is described in Section 5.2.3.

Overall, efficient use of Multi-SIMD requires orchestration of qubits to maximize parallelism and to reduce qubit motion. Devising and evaluating effective scheduling techniques for Multi-SIMD, at both the logical and physical level, is the focus of this chapter.
Figure 5.1: Block diagram of Multi-SIMD quantum architecture based on ion traps controlled by microwave technology. $k$ operating regions each support quantum operations on $d$ qubits simultaneously. Each operating region has a local memory for storing qubits. Dedicated SIMD regions supply a steady stream of purified $|0\rangle$ and $|A\rangle$ ancilla states. EPR ancillas are prepared at the global memories and distributed among regions via quantum teleportation. Physically, the architecture is comprised of many small ion traps, which can be used for both storing and transporting qubits. In this figure, $k = 4$ SIMD regions exist, and the capacity of each is an unspecified $d$.

5.2.2 Adaptability to Different Technologies

Our Multi-SIMD model is similarly applicable to any physical implementation technology and can be applied so long as the scaling properties hold. For example, superconducting qubits could be electronically tuned in and out of resonance in parallel, allowing SIMD operation in regions [23]. Recent work with quantum dots using rotating frames and microwaves also support Multi-SIMD in a manner very similar to trapped ions [121]. A global operation can also be applied to a lattice of neutral atoms. Further, work by Chi et. al. utilizes a SIMD control approach towards electron spins on liquid helium technology [36].

5.2.3 Handling Data Movement and Teleportation

During the execution of a series of quantum operations, many qubits have to move between different regions (i.e. storage or operation regions). In the quantum regime, there are three main means of transporting qubits:
Figure 5.2: Methods of communicating qubits in the chip: (a) Shuttling of ions between ion trap channels. Altering the voltage applied to traps allows ions to move in a straight line or to turn. (b) Swapping in stationary qubits such as superconductors or quantum dots. Using nearest neighbor operations, it is possible to transport the source ($q_1$) to the destination ($q_3$), using intermediary qubits if necessary ($q_2$). (c) Quantum teleportation: The EPR pair of $q_2/q_3$ is distributed prior to teleportation, keeping $q_2$ near the communication source $q_1$, and using $q_3$ as the (arbitrarily distant) communication destination. By measuring the states of $q_1$ and $q_2$, and classically transmitting those measurement results (classical 0 and 1 bits), the state of $q_3$ takes on that of $q_1$ using none, one, or both of the X and Z operations at the target side. This completes the transmission of $q_1$ to $q_3$, while the state of $q_1$ is destroyed in the process.
1. **Shuttling**: In a trapped-ion technology, the voltage of electrodes that create a potential well for trapping ions can be manipulated by precise pulses. This can be done such that the qubit moves between different electrodes, thereby physically moving from one place to another (Figure 5.2a). This mode of transportation is *distance-sensitive*, and the delay associated with it is directly proportional to the traveled distance, and the number of turns on the path. All shuttling transportations are done through the channels between regions in the Multi-SIMD architecture (dotted areas in Figure 6.4).

2. **Swapping**: In superconducting and quantum dot technologies, qubits are not mobile, and are thus incapable of being shuttled. These qubits can only interact with their neighboring qubits, through two-qubit operations such as the CNOT. Fortunately, it is possible to combine CNOTs in order to gain the effect of qubit swaps. Doing so repeatedly causes the qubit to move arbitrarily. The downside of this operation is that it is expensive in both time and space — it can take considerable time to do sequential swaps over a long distance, and it requires the space between the source and destination be filled with “intermediary” qubits.

3. **Teleportation**: Teleportation is a technique in quantum mechanics that makes transmission of exact qubit states possible. It works by *entanglement* of states, which causes operations on one qubit to affect the state of the other entangled qubit, even if the two are far apart from each other. Figure 5.2c shows the steps required. Since repeated movement of data qubits on the physical fabric is error-prone, teleportation reduces quantum decoherence by communicating information through a classical channel. However, teleportation does require a pre-distribution of entangled *Einstein-Podolsky-Rosen (EPR)* pairs of qubits (q2/q3) to the source and destination of each communication, so that the sender and receiver each have one half of the pair. The EPR distribution must be done via shuttling or swaps.

Leaving aside the pre-distribution stage, we note that contrary to its name, teleportation is not instantaneous, since the classical results of measurement must still be transmitted. However, quantum teleportation is *distance-insensitive*: the communication delay is just the number of cycles it takes to perform the operations shown in Figure 5.2c, regardless of the distance between source and destination. In the Multi-SIMD architecture, *teleportation units* (grey areas in Figure 6.4) in each region act as buffers for EPR pairs during teleportation.

Whenever we want to communicate data between two distinct parts of the architecture, there are two delays associated with the needed teleport operation: the delay associated with generating and distributing entangled EPR pairs, and the constant delay associated with the series of operations in Figure 5.2c, which is four times longer than a single quantum gate. This constant communication cost favors parallelism, since the serial version of our computations pay just as much to access global memory as our parallel versions do. In fact, parallelism breaks the computation into finer-grain chunks on separate regions. This reduces global memory accesses by leveraging local storage in regions and additional
local memories, which is analogous to spatial computing approaches in classical comput-
ing [64, 177, 180].

Our compiler and schedulers can mask teleportation latencies in two ways: First, pre-
distribution of EPR pairs in cycles prior to when the information transmission is needed
can mitigate the first latency. However, still in a naive movement model, communication
can quintuple the actual computation cost as a result of repeatedly moving qubits between
SIMD regions and global memory. In many cases our compiler can schedule teleportation
operations in parallel with the computation steps, such that the latency of moving qubits
around is masked.

While teleportation has constant latency with communication distance, longer distances
do imply higher EPR bandwidth requirements (larger communication channels to move
enough EPR pairs throughout the architecture). To minimize EPR bandwidth requirements,
the next chapter will investigate smoothing algorithms for mapping to such a non-uniform
memory architecture. Additionally, the issue of correctly dealing with the problem of de-
coherence at the physical level when moving EPR pairs, which may arise in long commu-
nication channels, is studied in [43].

5.2.4 Optimizing Communication using Local Memory

The communication overhead of four cycles per qubit movement in teleportation can be re-
duced even further by adding a local memory storage to each SIMD region. A local mem-
ory region bordering an operating region acts as a temporary store for a qubit that must
be moved out of the SIMD region for one or more cycles, and moved back into the same
region for the next computation on it. Due to the proximity of the local memory region,
qubits can be moved back and forth physically using shuttling or swapping. This eliminates
the need for computational cycles required for teleportation to global memory reducing the
communication overhead considerably. While the latency of shuttling or swapping trans-
port of a qubit depends on the size of the operating region (or the distance to be traversed),
note that even teleportation operation involves a similar shuttling or swapping latency for
accomplishing the interaction of data qubits with an EPR qubit. In our experiments, we
assume a one-cycle latency overhead for local communication.

5.3 Scheduling Operations on Data

Efficient scheduling of large-scale benchmarks requires a scalable approach and a delicate
balance between maximizing parallelism and minimizing communication. We implement
and study two different scheduling algorithms with different approaches to scheduling for
parallelism with communication awareness. We also use a hierarchical, coarse-grained
scheduler to limit final program size.

The primary means of communication within the Multi-SIMD architecture is telepor-
tation, which potentially requires an additional overhead of 4 timesteps per operation. If each
operation timestep requires data teleportation to receive its operands, then the full runtime
could be 5X longer than the computation critical path. By adapting our schedulers to ac-
count for data movement costs and to reduce unnecessary movements, our runtimes remain much closer to the computation-only ideal.

Schedules are stored as a list of sequential timesteps. Each timestep consists of an array of \( k + 1 \) SIMD regions. The 0th region contains a list of the qubits that will be moved and their sources and destinations timestep. The remaining SIMD regions contain an unsorted list of operations to be performed in that region. Operations consist of an operation type (X, Z, CNOT, etc.), a list of qubit operands and a list of pointers to each operand’s next operation.

### 5.3.1 Communication-Aware Scheduling with RCP

The Ready Critical Path (RCP) algorithm is a traditional parallel computing algorithm described in [195, 196]. The primary distinguishing feature is that it maintains a ready list instead of a free list for all tasks, only queuing the tasks that have all of their dependencies met. This prevents tasks from waiting for data during execution and reduces the overall run time. The ready list is a simple, unsorted list of operations.

Here RCP is extended to support the Multi-SIMD execution model by providing a priority scheduling mechanism that evaluates the operation type, movement cost, and slack. The operation type is used for grouping qubits to expose data parallelism. The movement cost indicates whether a bit needs to move (0) or is already in the present SIMD region (1); minimized movement is preferred. Slack is the graph distance between uses of a particular qubit, so if many operations need to occur before the next use that qubit does not need to be scheduled as soon; this is negatively correlated with the priority. The metrics can be multiplied by weights, \( w_{op} \), \( w_{dist} \), and \( w_{slack} \) respectively, though in this chapter all weights are set to 1.

As shown in Algorithm 2, at each timestep the priority for each operation type in each SIMD region is calculated. A maximum priority is maintained along with the SIMD region and operation type. As the priority for each configuration is calculated, this maximum and its corresponding values are updated. When all configuration priorities have been computed, the SIMD region and operation type with the highest priority is scheduled, the ready list is updated, and the SIMD region is removed from the list of available regions. This priority computation is repeated until all operations in the ready list are scheduled or all SIMD regions are occupied. The scheduler then updates the ready list with the operations that will be ready after the current timestep completes. The scheduler completes once the ready queue is empty.

### 5.3.2 Communication-Aware Scheduling with LPFS

Many of our benchmarks are highly serial; Figure 5.3 shows an average critical path speedup of around 1.5X. This serialization prevents the data dependencies from being accelerated significantly through parallelization, but it does provide ample opportunity for reducing expensive communication by optimizing the location of qubits throughout the execution. To this end, the Longest Path First Scheduling (LPFS) algorithm was developed (see Algorithm 3).
Algorithm 2: Ready Critical Path (RCP) algorithm. At each timestep, it computes the relative weight for scheduling an operation type to each SIMD region based on the prevalence of that operation, the distance of each qubit from that region, and the graph distance to the next use of that qubit. The highest weighted operation type is scheduled to its preferred region and the calculation is repeated until no more operations can be scheduled.

LPFS assigns \( l \) SIMD regions, where \( l < k \), to be dedicated to computing \( l \) longest paths. These longest paths are statically assigned to those regions and so any qubits in their operational paths will have relatively few movements. This is especially useful when arbitrary rotations are decomposed, where a single qubit may have up to several thousand operations performed sequentially with no movement of the target qubit.

The longest path algorithm initially finds the critical path in the program DAG generated by LLVM. It starts at the top of the DAG and sets a tag at each node with the furthest
Function $lpfs(DAG G, list of int simds, int l)$: Schedule $S$ is

for $i$ in 0 to $l$-1 do
  // Get longest paths for allocated SIMD regions
  $simd[i]$ = getNextLongestPath(G.top);
end

$ready = G.top();$

while (! $ready.empty()$ && ! $simd forall().empty()$ ) do
  // Schedule each time
  for $i$ in 0 to $l$-1 do
    // Schedule allocated SIMD regions
    if (refill && $simd[i].empty()$) then
      // Reuse SIMD region if it is out of operations
      $simd[i]$ = getNextLongestPath($ready$);
    end
    $op = simd[i].pop()$;
    $S[time][i].push(op)$;
    if (opportunistic_simd) then
      // Schedule ready operations of the same type
      $S[time][i].push(ready.getOps(op.op_type))$;
    end
  end
  for $i$ in $l$ to $k$-1 do
    // Schedule unallocated SIMD regions
    $optype = ready.top().op_type$;
    $S[time][i].push(ready.getOps(op.op_type))$;
  end
  for $op$ in $S[time].forall().getOps()$ do
    // Update ready list
    $ready.push(op.getReadyChildren())$;
  end
  $ready.uniq();$
  $time++$
end
return $S$;
end

Algorithm 3: Longest Path First Scheduling (LPFS) algorithm. First, the longest paths for $l$ allocated SIMD regions. At each timestep first schedule the allocated regions, then any unallocated regions are assigned from the free list. If the SIMD option is set, add data parallel operations to all regions. If refill is set, anytime a region completes its path find a new one from the current free list.

distance from the top. After finding the largest depth at the bottom, the path is traced back and recorded. Multiple longest paths can be found sequentially, and at arbitrary positions in the execution based on the current free list. By being able to find multiple longest paths, further movement can potentially be reduced.

All longest paths are returned as an ordered list of the operations to be performed in that path. These longest paths are stored in an array of size $l$ for each statically scheduled region.
All operations that are not on any of the active longest paths are added to a free list. These are scheduled in any of the SIMD regions in the range \([l+1, k]\) based on their order in the free list. This order is not prioritized, but is instead populated based on the order of operations as they are scheduled, and their descendants.

Two additional options can be used to control LPFS, SIMD and Refill. SIMD scheduling allows any SIMD region dedicated to a longest path execution to execute other free list operations of the same type; this setting also allows any timesteps where that SIMD region would have to stall for dependencies to execute arbitrary operations from the free list as well, thus providing SIMD parallelism. The Refill option is for the case when multiple longest paths are used and they are of unequal lengths. Once the shorter path completes, the next longest un-executed path will be found and scheduled in the region that completed. Our experiments were run with \(l = 1\), and both SIMD and Refill enabled.

### 5.3.3 Hierarchical Scheduling

To allow benchmarks to scale beyond tractable sizes, we employ a coarse-grained schedule which stitches together optimized schedules for leaf modules scheduled by RCP and LPFS in our call tree. The scheduler uses a simple, list-based approach to schedule leaf modules and operations in non-leaf modules with the goals of improving parallelism and/or reducing communication overheads. Algorithm 4 illustrates the overall approach.

Given a set of pre-scheduled leaf nodes, the coarse-grained scheduler completes the program scheduling by using a list scheduler to compose together a full schedule. Operations are assigned priorities based on criticality and scheduled in priority order. The quantum gate operations in non-leaf modules are scheduled along with invocations to other modules. The invoked modules have been previously scheduled by one of the fine-grained schedulers discussed next and are now treated as blackbox functions. Once the schedule for a module is determined, it is characterized as a blackbox with a length dimension equal to schedule length, and a width dimension equal to highest degree of parallelism found in the schedule.

To allow the coarse-grained scheduler to effectively parallelize the invoked blackboxes within the width (\(k\)) constraint, we consider flexible rectangular dimensions for each blackbox. During fine-grained scheduling of each module, multiple schedules are determined to find schedule lengths with widths between 1 to \(k\). The coarse-grained scheduler is presented these blackboxes with multiple dimensions. When parallelizable modules are encountered, the combination of blackboxes that yields the minimal length subject to the width constraint is chosen. Algorithm 4 shows the pseudo-code for coarse-grained scheduling with flexible blackbox dimensions.

In a \(k\)-resource constrained schedule, the width for any invoked module is at most \(k\). Any operations (other than invoked modules) encountered by the coarse-grained scheduler have an operation execution cost of 1 and a movement cost of 4.

### 5.3.4 Scheduling Communication using Local Memory

To reduce communication overheads, we study further optimizations using local memories attached to the SIMD regions. Once computations are mapped to SIMD regions and the
for each non-flat module do
    // Track schedule in terms of blackbox dimensions
    totalL = 0; totalW = 0; // total length and width
    currL = 0; currW = 0; // current length and width

    for each op Fi in a priority-ordered set of ops \{Fi: Prioritiy(Fi) \geq Prioritiy(Fj) \text{ if } (i < j)\} do
        Check predecessors to find the earliest timestep \(t_e\) in which \(Fi\) can be scheduled
        Get width \(W\) and length \(L\) for \(Fi\)
        if \((t_e \leq totalL + currL)\) then
            // dependencies show that \(Fi\) can be parallelized with previous schedule
            if \((currW + W \leq K)\) then
                // parallelize the operation \(Fi\)
                timestep\((Fi)\) = max\((totalL+1, t_e)\)
                currW = currW+W, currL = max\((currL, timestep(Fi)+L)\)
                \(F_p = \{F_p, F_i\} \) // Add to set of parallel functions in current schedule
            else
                // k-constraint would be violated if parallelized
                for set of functions \(\{F_p, F_i\}\) do
                    Try all combinations of possible widths, and compute length.
                end
                if one or more combinations found with combined width \(\leq K\) then
                    Choose combination with smallest length.
                    \(currW = \text{Width of combination, } currL = \text{Length of combination}\)
                    \(F_p = \{F_p, F_i\} \) // Add to set of parallel functions in current schedule
                else
                    // serialize \(Fi\) due to k-constraint
                    totalW = max\((totalW, currW)\), totalL = totalL + currL
                    timestep\((Fi)\) = totalL+1
                    currW = W, currL = L
                    \(F_p = \{Fi\} \) // set of parallel functions in current schedule
                end
            else
                // serialize \(Fi\) due to data dependency
                totalW = max\((totalW, currW)\), totalL = totalL + currL
                timestep\((Fi)\) = totalL+1
                currW = W, currL = L
                \(F_p = \{Fi\} \) // set of parallel functions in current schedule
            end
        end
    end

    totalW = max\((totalW, currW)\), totalL = totalL + currL
    Store totalW and totalL in data structure
end

Algorithm 4: Hierarchical scheduling algorithm for \(k\) SIMD regions. Operations are scheduled in priority order, like list scheduling. Contains flexible blackbox dimensions.

necessary communication schedules are determined, some qubits can be moved to local memories using a lower overhead shuttling or swapping movement instead of teleportation to the global memory. Distinguishing a move as a local move or teleportation primarily depends on where its next computation is scheduled, but local memory regions may also be constrained by capacity. With prior information about the next computation and available space in the local memory, a move can be directed to a local or global memory by the scheduler.
When a qubit in a SIMD region (source) must be stored for a few cycles before being used in a different SIMD region (destination), there are multiple options for storage: the global memory, the source local memory, the destination local memory, or if idle, either of the two SIMD regions. With limited local memory capacity, holding qubits for other regions or holding additional qubits before they are required can both reduce the space available for qubits being currently operated upon. For simplicity, unless the source SIMD region is idle, we move such qubits to the global memory for storage.

After a computation or communication cycle, qubits which are not scheduled for the next operation are moved to global or scratchpad memory, while those which undergo further operations in the same region will stay in place. Physically, any qubits left behind from the teleportation of state can be reinitialized and reused as ancilla or EPR pairs.

The local moves are derived from the RCP or LPFS schedules on leaf modules. The schedules are updated to reflect the single cycle local move if its next operation location is in the same region and there is space in the local memory, otherwise the original teleportation movement is kept. If any SIMD regions in a timestep have a global move, the full four cycle move time is retained to avoid disrupting the schedule.

5.4 Results

We present three sets of results. First, we evaluate the amount of parallelism our schedulers can expose when communication is assumed to be zero-cost. Second, we demonstrate the benefits of communication-aware schedulers when accounting for communication costs to global memory. Finally, we evaluate the benefits of local scratchpad memories associated with each SIMD region.

In each set of results we look at all eight benchmarks under the LPFS and RCP scheduling algorithms. Both algorithms are run with \( k = 2 \) and \( 4 \) (the number of regions). For simplicity, we assume infinite \( d \) (number of qubits in a region). LPFS also uses Refill and SIMD options. Refill lets a region with a longest path schedule a new path if the current one completes, and SIMD allows non-path operations to be SIMD scheduled in a longest path region. All experiments were run at the logical level, omitting physical qubit sub-operations and error-correction, as these are constant multipliers to the overall runtimes within the range of runtimes involved.

5.4.1 Characterizing Logical Parallelism

In our first set of results, looking solely at the parallelism in the benchmarks, we are able to compare the speedups provided by a Multi-SIMD architecture against the theoretical maximum based on the estimated critical path. As seen in Figure 5.3, all of the benchmarks except Shor’s were able to achieve near-theoretical speedup on the Multi-SIMD architecture, at either \( k = 2 \) or \( 4 \). This is assisted by using \( d = \infty \), which allows for clustering as many qubits as possible into SIMD regions. In Section 5.4.4 we explain the difference in trend for Shor’s.

RCP speedups are lower than or equal to LPFS in every benchmark except TFP, in some cases tying at \( k = 4 \) with LPFS at \( k = 2 \). This is due to RCP being better suited to coarse-
Figure 5.3: The speedup over sequential execution of each benchmark with each scheduling algorithm, compared to the estimated critical path. Most benchmarks do not need many SIMD regions in order to achieve near critical path speedup, due to modest parallelism, and the fact that most of this parallelism comes from data- and not instruction-parallelism. Almost all benchmarks, except Shor’s, achieve near-complete speedup by $k = 4$.

5.4.2 Runtime Speedup with Data Movement Analysis

Figure 5.4 shows all scheduling algorithm speedups over a naive movement model where data is moved between SIMD regions and global memory every timestep, effectively increasing the overall runtime by 5X. All benchmarks show some speedup over communication-unaware runtime models due to reduced movement. An average increase in speedup of 46% is seen across all benchmarks. The largest gain is seen in GSE (308%). The critical path was not used for a theoretical bound in these or the next results because we did not have a viable critical path model for incorporating movement.

Some benchmarks, such as BF, CN, SQ, SHA-1 and Shor’s, have a large number of highly dependent, serial operations. BF, CN and SHA-1 are composed of several CTQG modules, which produces unoptimized code that is highly locally serialized. This results in benchmarks that have a low degree of parallelism and cannot be well optimized. The in-
interactions between data dependencies also result in many small (1-2 qubit) moves between global memory and various SIMD regions that cannot be removed to improve performance. GSE shows the largest gains due to its distinctive structure. The two key qubit registers containing the primary active qubits are rarely moved out of an SIMD region once they are in place and typically have long sequences of operations on the same qubits. This results in very few moves either between SIMD regions or memory.

5.4.3 Runtime Speedup with Local Memories

Figure 5.5 illustrates the performance benefits of adding local memory to SIMD regions. To study the effectiveness of a small amount of local memory in each SIMD region, we experiment with varying memory sizes. For each benchmark, we limit the capacity of local memory to one-half and one-quarter of the minimum qubits \( Q \) required by the benchmark. \( Q \) is computed by scheduling the benchmark run sequentially, with maximal possible reuse of ancilla qubits across functions. Table 5.1 shows the value of \( Q \) determined for each benchmark. A parallel schedule of the benchmark may require more than \( Q \) qubits, thus the global memory may be much larger than the size of \( Q \). The maximum benefit of local memory is demonstrated with an infinite capacity memory region.

Algorithms that are effective at reducing communication overheads do so by scheduling maximally interacting qubits in a single region. Adding local memory to SIMD regions raises the effectiveness of these algorithms since it allows the frequently interacting qubits
Figure 5.5: Speedups due to the addition of local memory to operating regions, computed over a sequential, naive movement model. Results are shown for a Multi-SIMD(4,∞) architecture.

to stay close. Thus, LPFS naturally sees higher benefits than RCP with local memory for most benchmarks.

These data movement based speedups show that, at least in some cases, it is possible to minimize the communications within the system to achieve a reasonable speedup (up to 9.82X in the case of SHA-1) in a practical system with physical limits on EPR distribution for teleportation and limited parallelism.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF $x=2, y=2$</td>
<td>1895</td>
</tr>
<tr>
<td>BWT $n=300, s=3000$</td>
<td>2719</td>
</tr>
<tr>
<td>CN $p=6$</td>
<td>60126</td>
</tr>
<tr>
<td>SQ $n=40$</td>
<td>120</td>
</tr>
<tr>
<td>GSE $M=10$</td>
<td>13</td>
</tr>
<tr>
<td>SHA-1 $n=448$</td>
<td>472746</td>
</tr>
<tr>
<td>Shors $n=512$</td>
<td>5634</td>
</tr>
<tr>
<td>TFP $n=5$</td>
<td>176</td>
</tr>
</tbody>
</table>

Table 5.1: The minimum number of qubits $Q$ required by the benchmarks, computed with sequential execution and maximum reuse of ancilla qubits.
5.4.4 Sensitivity to $d$ and $k$ parameters:

The preceding results show that in most quantum benchmarks, data-level parallelism ($d$) achieved through microwave signaling is able to capture most of the parallelism, even with modest numbers of distinct SIMD regions. In fact, even though we practically assumed infinite amount of data-parallelism available in our SIMD regions, our other experiments have shown that decreasing this to below 32 qubits only causes marginal changes. This is important, since we have solely focused on the logical level, but we can see that even with added qubits for error correction, the architecture is able to cope and perform well.

Shor’s showed a greater sensitivity to the number of SIMD regions available ($k$). The reason for this can be traced to the large number of rotation operations that exists in this code. These rotation operations can theoretically execute at the same time because they are on distinct qubits, except for the fact that practically they need to be decomposed into primitive, standard operations (as described in Chapter 3). This can prohibit the parallelization of operations unless more SIMD regions are created to accommodate them, as illustrated in Table 5.2. Since many of these rotations were not inlined into the code, to keep the size manageable, they remain as blackboxes in the course-grained schedule. That causes the scheduler to allocate a separate region to each, effectively increasing the need for these regions. In Figure 5.6, we show how the speedups vary with $k$ for this benchmark.

<table>
<thead>
<tr>
<th>Rotation Operation</th>
<th>Primitive Operations Approximating Rotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_z(q_1, \theta_1)$</td>
<td>$T(q_1) - S^\dagger(q_1) - H(q_1) - Z(q_1) - ...$</td>
</tr>
<tr>
<td>$R_z(q_2, \theta_2)$</td>
<td>$H(q_2) - Y(q_2) - X(q_2) - H(q_2) - ...$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$R_z(q_n, \theta_n)$</td>
<td>$S(q_n) - X(q_n) - T(q_n) - T^\dagger(q_n) - ...$</td>
</tr>
</tbody>
</table>

Table 5.2: Parallel rotations cannot be executed simultaneously on a hardware with primitive operations, unless there are enough SIMD regions to accommodate them.

5.5 Related Work

This chapter builds on several important previous studies relating to SIMD parallelism [36,94,161], ancilla preparation [79,84], and quantum architecture [36,124,182]. Our work is the first to use a complete compiler infrastructure to discover this parallelism, allowing us to evaluate a non-trivial set of benchmarks (previous work focused almost exclusively on Shor’s and Grover’s, or other small quantum circuits). It is also the first to incorporate data movement analysis and optimizations within the compiler framework established.

Some prior work has explored optimization of execution latencies with SIMD architectures, but in a more limited context. Chi et al. [36] proposed a SIMD architecture based on the technology of electron spins on liquid helium. For a quantum carry-lookahead adder circuit, they evaluated pipelining of ancilla preparation for CNOT and Toffoli gates to reduce latency, and optimization of width of SIMD regions to reduce area requirements. Our work builds on this model, with the implementation of a complete compiler and the study of a much larger and more diverse benchmark suite.
Figure 5.6: Shor’s speedups as scheduled with a communication-aware scheduler on a Multi-SIMD architecture with local memories. High numbers of rotations cause long serial threads of operations to each execute on a separate SIMD region, thus getting better gains with higher $k$.

Schuchman et al. [161] identify a high-level parallelism pertaining to specific quantum tasks of uncomputation (analogous to garbage collection for qubits) and propose a multi-core architecture to minimize latency and expensive inter-core communication during their execution. This kind of parallelism fits well into our Multi-SIMD model; it can be easily extended to support the proposed multiple cores. Some degree of uncomputation already exists in the compiled code of our benchmarks and is naturally parallelized by our model, and more can be added in the future to reclaim unused qubits.

## 5.6 Chapter Summary

We have proposed a Multi-SIMD architectural target for this compiler that incorporates practical physical constraints in quantum computing technology. To fully exploit its computational power, our work has also developed a scalable compiler that uses deep analysis to create logical schedules that enable parallelism and minimize communication. Largely as a result of SIMD parallelism, efficient communication and fine-grained data locality, we achieve speedups over our sequential baseline of 2.3X to 9.8X. We propose a new Longest Path First Scheduling (LPFS) algorithm, and find that it outperforms a more traditional parallel computing algorithm such as the Ready Critical Path (RCP), largely due to the high cost of communication. Explicit prioritizing of communication reduction proves to be a useful goal in the scheduling of quantum workloads. Since quantum error correction can have overhead exponential in program execution time [11, 136, 170], these speedups can be even more significant than they appear, because they offer important leverage in allowing complex QC programs to complete with manageable levels of QEC.
Chapter 6

Quantum Error Correction: A Comparative Study of the Topological Surface Code and Concatenated Steane Code

6.1 Introduction

Critical to making the most use of quantum machines is choosing the lowest-overhead type of error correction, as it is by far the most expensive component of computation. In this chapter, we evaluate three established quantum error correction codes — concatenated Steane, double-defect surface, and planar surface — using the full set of compilation, scheduling and network simulation tools that we have developed seeking to optimize mappings of quantum applications to physical technology. We quantify the cross-over boundary between the favorability of each code in terms of space-time overhead. We also measure the sensitivity of these boundaries to both application and technology characteristics. Contrary to previous expectations, we find that communication congestion can favor planar surface codes in both trapped-ion and superconducting technologies, especially in highly parallel applications.

In building the next generation of QCs, there is yet no firm consensus on a fundamental question: the exact choice of quantum error correction (QEC) code to protect fragile quantum states. This is the most resource-consuming aspect of QC. The reason is the large gap between reliability requirements of an application (i.e. the maximum acceptable error-per-operation that the application imposes, called logical error rate ($p_L$)), and reliability rates of a technology (i.e. the best achievable fidelity on a device, called physical error rate ($p_P$)). For the next ten years or more, qubits in quantum computers will number in the tens or hundreds, not millions. For this reason, we do not want to lose more to QEC than is necessary. Therefore it is imperative to have an accurate estimate of the time and space

---

1The work in this chapter was performed in part in collaboration with Adam Holmes, Pranav Gokhale and others.
In this chapter, we explore the tradeoffs between two main families of QEC, concatenated and surface codes. Theoretical results [48, 60, 170] provide us with some basic information regarding this comparison. For example, we know that the asymptotic resource efficiency of surface codes over concatenated codes ensures that they are more favorable when very strong encoding is required, i.e. when physical technology is very faulty, or when application size is very large (i.e. imposes stringently low error requirements). In addition, each code has a threshold error, above which it cannot operate. This is higher for surface codes than concatenated codes. Figure 6.1 depicts these rough regions of comparison.

However, beyond rough estimates of these extreme cases, it is unknown where the actual regions of favorability lie on this graph. Finding the cross-over boundary between the regions in Figure 6.1 necessitates the simulation of computation and communication steps for various configurations of the design space. By proposing a unified toolflow that facilitates iterative design and evaluation, we are able to accurately quantify this boundary for the first time.

We focus on three of the leading choices for quantum technology, namely trapped-ions, superconductors and quantum dots, in addition to the aforementioned QEC families. The crossproduct of these leads to six schemes for implementing an arbitrary quantum application, with vastly different computation and communication models. These are summarized in Table 6.1.
Table 6.1: The design space for implementing an error-corrected quantum application on physical hardware. Based on valid computation and communication models, for each technology there are three QEC choices: (1) concatenated, (2) planar surface (teleport-based), (3) double-defect surface (braid-based).

Table 6.1 shows that for all technologies, there are three methods by which to implement error correction: one concatenated method, and two variations — teleport-based (planar) and braid-based (double-defect) — of the surface method. Due to their similar nature (i.e. constraint of nearest-neighbor-only interactions), superconductors and quantum dots have the same behavior in this table.

In this work, we make some key observations:

- Previous work has shown teleport-based communication to be useful for trapped ions, as it decouples the hard part of moving physical qubits from the actual point of communication in a program [79, 122]. On the other hand, double-defect surface codes were deemed suitable for the nearest-neighbor constraint of superconducting and quantum dot qubits. But long braids cannot cross, causing their own form of communication contention when many logical qubits must interact in parallel. We evaluate planar surface codes as a hybrid scheme, taking tiles of surface codes and connecting them with teleportation instead of braids. Hence, we can once again decouple the hard part of the communication from the point it is needed in the application. Although planar codes were designed for trapped-ion machines which support free qubit mobility, we shall see that these codes can also outperform braid-based codes on superconducting and quantum dot machines as well, especially when applications exhibit high parallelism.

- We determine how sensitive Figure 6.1’s favorability regions are to variations to technology characteristics. This is important due to the evolving nature of the field. For example, we determine that improving 2-qubit gates relative to 1-qubit gates results in better concatenated codes on superconductors and quantum dots, but better surface codes on trapped ions.

- One of the main strengths of our work is explicit modeling of communication delays. Resource estimates based solely on metrics such as number of gates and parallelism factor [84, 174] risk missing crucial contributing elements. For example, arbitrary gates cannot be easily implemented, necessitating a compilation stage into a technology-supported ISA. Similarly, physical qubits can typically only interact...
when placed in the vicinity of one another, necessitating an architecture capable of communicating qubits to perform multi-qubit gates.

The rest of this chapter is organized as follows: Sections 6.2 and 6.3 outline the main properties of technologies and QEC families used in this study. In Section 6.4, we present our methodology and toolflow. Sections 6.5 and 6.6 detail optimizations on teleport- and braid-based communications. Section 6.7 shows results, Section 6.9 discusses related work, and Section 6.10 concludes the chapter.

## 6.2 Device Technology

Experimental physics research is ongoing to create a technology that can accommodate large numbers of stable qubits, gate operations with high fidelity, and the ability to easily address and control the system’s many qubits. In this work we focus on three leading candidates: Trapped ions, superconductors and quantum dots.

Our goal is to create a model that abstracts away physical details not relevant to calculations of space and time. Below our abstraction layer are detailed control sequences of physical systems, such as precise pulses, isolation from the outside world, and qubit cooling mechanisms. Above this layer are qubits and gates that need to be executed. In our model, we simplify this transition by identifying key attributes of a technology, and proceed to compare different technologies based on them. These attributes are: Qubit mobility, relative latency of gates (i.e. 2-qubit gate vs. 1-qubit gate vs. measurement speeds), and error rate (i.e. percentage of gates that are faulty on average).

Table 6.2 summarizes the most important technology parameters, based on recent experimental results. We use them to build a realistic model of execution and arrive at a more precise version of Figure 6.1. Section 6.7 contains simulation results with these parameters, plus a more optimistic set to account for future improvements.

## 6.3 Quantum Error Correction (QEC)

This section describes the two main families of QEC in our study, and the implications they have for space-time overheads in a particular application-technology configuration. As with any type of error resiliency, we use redundancy of information to protect the desired state. The encoded qubit is called the logical qubit, while its underlying qubits are called physical qubits.

### 6.3.1 Concatenated Codes

Concatenated codes [103] are a family of codes built around the idea that a logical encoded block may be recursively encoded, with each recursion—called concatenation level (l)—providing a more reliable code block. We explore a version of concatenated codes called the Steane code [170], which has one of the lowest overheads in this family. Our tools are easily extensible to similar codes [14, 108]. Proper implementation of the Steane code has several challenges, as outlined below.
Figure 6.2: The difference in scaling and resource usage for encoding a single logical qubit in (a) concatenated Steane and (b) planar surface codes. Surface codes do not suppress errors as fast as concatenated codes (slower slope increase), however their resource consumption grows much more modestly (slower darkening of shades). The lines converge to the relevant physical error threshold — $10^{-4}$ for (a) and $10^{-2}$ for (b) — beyond which errors cannot be corrected.
<table>
<thead>
<tr>
<th></th>
<th>Gate Latency</th>
<th>Movement Latency</th>
<th>Gate Error Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Trapped-Ion Qubits</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.5µs (1-qubit)</td>
<td></td>
<td>1µs / hop</td>
<td>10^{-6} (1-qubit)</td>
</tr>
<tr>
<td>100µs (2-qubit)</td>
<td></td>
<td>10µs / turn</td>
<td>10^{-3} (2-qubit)</td>
</tr>
<tr>
<td>25µs (measure)</td>
<td></td>
<td>[29,74,188]</td>
<td>10^{-4} (measure)</td>
</tr>
<tr>
<td></td>
<td>[16,112,127]</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Superconducting Qubits</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1ns (1-qubit)</td>
<td></td>
<td></td>
<td>10^{-3} (1-qubit)</td>
</tr>
<tr>
<td>40ns (2-qubit)</td>
<td></td>
<td></td>
<td>10^{-2} (2-qubit)</td>
</tr>
<tr>
<td>140ns (measure)</td>
<td>N/A</td>
<td></td>
<td>[18]</td>
</tr>
<tr>
<td></td>
<td>[18,157]</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Quantum Dot Qubits</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50ps (1-qubit)</td>
<td></td>
<td></td>
<td>10^{-2} (1-qubit)</td>
</tr>
<tr>
<td>30ns (2-qubit)</td>
<td></td>
<td></td>
<td>10^{-1} (2-qubit)</td>
</tr>
<tr>
<td>1ns (measure)</td>
<td>N/A</td>
<td></td>
<td>[131]</td>
</tr>
<tr>
<td></td>
<td>[84,106,167]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: The most important parameters affecting our models for space-time calculations, for different technologies. Trapped ions enjoy higher-fidelity gates, while superconductors and quantum dots have faster gates (ion gates speeds can in principle be increased [21,63]). Quantum dots are the fastest technology, albeit with lower reliability.

First, a challenge in quantum error detection is that reading a state collapses the state, so errors must be detected and corrected without knowing exact values. This necessitates the use of extra “helper” qubits, known as ancillas. Ancillas are associated with logical data for syndrome measurement—learning just enough about the encoded block to understand what kind of error might have occurred on it. The key idea is to divide the large state space into a valid code subspace and an orthogonal error subspace. Syndrome measurement then has the effect of projecting the state onto the basis vectors of the error subspace, quantizing errors. They can then be corrected by applying a corrective gate.

Second, Steane QEC has the important advantage of allowing bitwise operations; that is, applying the same operation to the \( i \)th physical qubit of every block has the effect of applying that operation to the whole logical qubit. However, no QEC allows a universal set of just bitwise gates [54, 198] (except with high-overhead code-switching techniques [104]). The T gate is our non-bitwise gate, and it requires a special ancilla be prepared and pre-distributed to the gate’s location [31].
Third, data communication occurs through teleportation, which is a quick method of communicating quantum information over arbitrary distances with constant latency. While low-overhead, teleportation is not without cost; to use it, we need to pre-distribute two physical ancillas, to the source and destination of the teleportation.

In summary, we can build a model of the concatenated Steane code based on bitwise computation and teleport communication; but we must also carefully consider the extensive physical movement of ancillas in this model. In trapped ions, this physical movement can be achieved efficiently by qubit shuttling. In superconductors, physical qubit movements are much less efficient, as they rely on a chain of swaps (nearest-neighbor interactions) to move the qubit from source to destination. This is more resource-intensive, using both more qubits (communication channels must be “filled” with unbroken chains of adjacent qubits), and more time (every swap consists of three CNOT gates).

6.3.2 Surface Codes

The second method of error correction we explore is the surface code [60], a special case of topological codes, where qubit information is encoded in the topology of a two-dimensional lattice of physical qubits [151]. Extending the lattice dimensions increases its reliability. The measure of coding strength is called code distance \((d)\), corresponding to the dimensions of the lattice.

A surface code lattice alternates “data” and “measure” qubits. The measure qubits are responsible for continuously measuring syndromes from surrounding data. In this regard, they are similar to ancilla in concatenated codes. However, the important difference is that the number of qubits on the lattice are fixed, whereas ancilla are continuously generated and recycled. Recording a history of syndrome measurements, errors can be detected using maximum likelihood reasonings.

We explore two surface code flavors, double-defect [60] and planar [33,48], which differ mainly in their models of computation and communication. Double-defect implementations consist of one monolithic lattice, where different regions correspond to different logical qubits. Aggregated syndrome measurements in time can be likened to a 3D space-time volume; the bottom area being the lattice, and the vertical height the progress in time. All operations on and interactions between logical qubits occur via a special operation known as braiding, similar to 3D pipe movements in this space-time volume. If one visualizes a 2D array of measurement cycles across the lattice, then the braid represents points where no such measurement occurs on a given cycle. The planar code, on the other hand, is a hybrid code: each qubit is encoded in its own lattice or “plane” as in double-defects, but they use bitwise computation and teleport-based communication, similar to concatenated codes.

6.3.3 Comparing QEC Codes

Figure 6.2 shows logical error rate \((P_L)\) as a function of physical error rate \((p_P)\) for both QEC schemes. Each line corresponds to a different level of concatenation \((l)\) or a different code distance \((d)\). The regions between the lines are shaded proportional to the physical
qubit overhead \((q)\) of encoding one logical qubit at that concatenation level or code distance.

Concatenated codes reduce the error doubly exponentially for an exponential cost in resources. Surface codes reduce the error exponentially for a quadratic cost in resources. Surface codes have better resource scaling for two main reasons. First, the code distance parameter increases lattice size in fine-grain quadratic resolutions until the desired reliability is achieved, whereas added concatenation levels expand code blocks in large exponential leaps. Second, surface codes have better decoders. Decoding a logical qubit in surface codes simultaneously aggregates information from all underlying physical qubits, whereas concatenated codes impose hard decoding at each level.

An important difference between concatenated and surface codes is the \textit{code threshold}, which is the minimum reliability required for the code to function properly. Quantum devices need to meet a threshold reliability requirement before any sort of error correction can further increase their reliability. Through error simulations, code thresholds are calculated to be around \(10^{-4}\) for the concatenated Steane [132] and \(10^{-2}\) for the surface code [60]. As expected, the lines in Figures 6.2a and 6.2b ultimately converge to their respective thresholds. This illustrates a major advantage of surface codes, namely the fact that they can operate on much faultier devices, which is highly desirable for current technological limitations.

### 6.4 Methodology and Toolflow

Figure 6.3 depicts our overall toolflow. We use the ScaffCC compiler already introduced in Chapter 3 as the frontend to our toolflow, lowering high-level descriptions of quantum algorithms to a standard logical-level QASM. The new addition is the backend, responsible for adding extra physical-level redundancies to ensure error-tolerance, as well as performing optimizations and simulations to calculate the final space-time overhead.

For a fair and comprehensive study, the tools must account for factors such as the application parallelism, technology characteristics (e.g. gate speeds, reliability rates, mobility of qubits) and communication characteristics (e.g. contention, teleportation vs. braiding models, straight moves vs. turns). The toolflow’s end-to-end integration allows for analysis of program graphs through multiple stages and application of various optimization heuristics. It also facilitates iterative designs through feedback capabilities. For example, the overhead of downstream error correction can often depend on the degree of code flattening in earlier compilation stages, which can be optimized iteratively (as discussed in Section 6.7).

Physical redundancies take multiple forms: extra data qubits are needed to encode one logical data, and extra ancillas must be generated to facilitate teleportation, syndrome measurement, or T gate implementation.

Optimizations occur at two levels: logical-level reduction of total communication needs through exploitation of data locality, and physical-level improvements in the overhead of those essential communications through optimizing the network load.

The next two sections (6.5 and 6.6) are dedicated to the details of concatenated and surface code optimizations. The final design space is explored in Section 6.7 through a complete simulation of concatenated and surface codes for all combinations of quantum...
applications, problem sizes, and physical error rates, on all three trapped-ion, superconducting and quantum dot technologies. Of particular interest is the parallelism potential of applications, which will affect the performance of our optimization protocols. We show sets of results for two parallel applications (IM and SHA-1) and two serial applications (SQ and GSE).

### 6.5 Optimized Teleportation

This section discusses optimizations in QEC schemes that use bitwise computation and teleportation-based communication, namely the concatenated code and the planar surface code. Section 6.5.1 describes a logical-level optimization to reduce total necessary teleportations, while 6.5.2 proposes physical-level optimizations of both time and qubit usage.

#### 6.5.1 Effectiveness of a Multi-SIMD Architecture

Bitwise computation behaves similarly in both concatenated and surface codes; identical gates are applied to many qubits comprising an encoded block in the former, or an encoded lattice in the latter. This suggests that a SIMD-style architecture is suitable.

We use the Multi-SIMD architecture proposed in Chapter 5, which combines qubit-level parallelism with operation-level parallelism: many qubits undergoing the same operation...
are clustered in one SIMD region, and multiple (reconfigurable) SIMD regions can accommodate heterogenous types of operations at any cycle. There is well-established technological support for this: trapped-ion systems can use microwaves to operate on hundred of qubits simultaneously. Superconducting and quantum dot systems can leverage voltage multiplexing to achieve the same. This massively reduces control complexity in large QCs.

A natural source of latency in the Multi-SIMD architecture is movement of data, due to the potential for numerous qubit movements between different regions. Therefore, optimizing communication is a central part of optimizing overall time. We use the logical-level scheduling approach of [7] which minimizes total teleportations by pinning the longest chain of operation dependencies to one region, and also using small local memories next to each region.

Although the reduction of total teleportations has a good impact on reducing overall program latency (up to $\sim 10X$), this is still a logical-level optimization that ignores physical overhead. Next, we discuss the sources of physical overhead and methods to mitigate it.

![Communication requirements on a SIMD-based architecture with 8 computation and 8 distributed memory regions. Solid (green) arrows are a sample of ancilla communications on the interconnection network, to support teleportation, T gate, or QEC operations (purple). Although heavy data dependencies may exist, ancillas enjoy high flexibility in scheduling.](image)

Figure 6.4: Communication requirements on a SIMD-based architecture with 8 computation and 8 distributed memory regions. Solid (green) arrows are a sample of ancilla communications on the interconnection network, to support teleportation, T gate, or QEC operations (purple). Although heavy data dependencies may exist, ancillas enjoy high flexibility in scheduling.

101
6.5.2 Smoothing Ancilla Distribution

Transportation of data by teleportation is a virtual operation that does not use the physical interconnection network to move qubits. Instead, the network links are used by underlying ancilla qubits that facilitate the task of teleportation. Ancillas are physically transported for other tasks too, such as performing T gates or doing rounds of error correction. Figure 6.4 shows how ancilla qubits must continuously be generated and distributed to support such higher-level tasks. We use dedicated factories [171] in the architecture for generating a steady supply of each of the three ancilla types [79, 84].

However, ancillas are temporary qubits by nature. For each higher-level task, fresh ancilla are used, which are then recycled after they fulfill their purpose. This is how teleportation is efficient. It allows decoupling of the hard part of a communication (physical movement on the mesh) from the point of use in the program. Ancillas are not bound by stringent data dependencies, and they can be prefetched at arbitrary points in time. If prefetched very close to the usage point during program execution, there will be the least amount of unnecessary qubits created and in flight at any given time, albeit at the risk of potential data stalls. On the other hand, prefetching much further in advance imposes no risk of stalls, at the cost of accumulating extra idle qubits for long periods of time.

Our goal is to achieve “just-in-time” distribution by smoothing the network load so as to avoid congestion. We do this by anticipating the usage point of an ancilla, and generating and communicating it some number of cycles before reaching that point. Given the specialized nature of quantum applications, we have perfect static knowledge of when each ancilla will be needed. Hence, walking the dependency graph, we use look-ahead windows to anticipate usage points.

We achieve good performance across all applications, with up to $\sim 24X$ savings in qubit cost and only a maximum of $\sim 4\%$ extra latency. The choice of window size is important: Figure 6.5 shows that a good window choice can reduce peak qubit usage by a factor of $\sim 11X$, at almost no cost to latency. However, small windows, although substantially capping qubit usage, may impose unacceptable latencies. Application seriality or parallelism does not affect these results, since ancillas do not follow regular data dependencies.

6.6 Optimized Braiding

In this section, we describe a scalable yet efficient solution to the braiding problem, which is the main method of computation and communication in braid-based surface codes. Section 6.6.1 lays out our general method, while Sections 6.6.2 and 6.6.3 propose logical- and physical-level optimizations.

6.6.1 Braid Simulation using Message Passing

Figure 6.6 shows how a simple logical CNOT operation may be performed between two double-defect logical qubits, each mapped to a specific location on the 2D lattice. Figure 6.6a shows the entire 3D space-time volume of such operation, where (white) data...
Figure 6.5: Smoothing ancilla distribution over the execution cycles using look-ahead windows. Magenta shows the non-smoothed case, while blue and green correspond to smoothing with windows of length 50 and 500 respectively. Smoothing with the right window size can greatly reduce qubit usage during execution cycles, while incurring less than 5% latency.

defects are stationary and extend upwards in time, and (black) ancilla defects are braided so as to cause the desired 2-qubit interaction.

We propose an automated braid synthesis and optimization approach that is much more scalable than traditional hand optimizations, and yet achieves good performance. We do this by tracing the state of the 2D lattice over various time slices. As such, the problem is reduced to simulating a mesh network with braids as messages and qubit tile corners as routers. Employing network optimization techniques to compress the overall time of communicating all program braids has the same effect as compressing the total 3D volume, but with much higher automation and scalability.

The cost to scalability is some reduction in efficiency of packing braids. A regular network simulation fails to discover all valid scenarios of a 3D topological transformation, such as stretching braids backwards in time. However, we discovered that in the majority of our surveyed applications, operations are highly serial and therefore immune to such inefficiencies (low parallelism reduces the need for interference optimization from the start). For those applications that do contain highly parallel operations, Sections 6.6.2 and 6.6.3 propose heuristics to address the inefficiencies arising from braid contention.

Several characteristics make braid networks distinct: (a) braids travel $n$ hops in one cycle, (b) some braids have to remain stable for $d$ (code distance) cycles to stabilize syndrome measurements, (c) routers cannot buffer braids and (d) virtual channels cannot be used as braids cannot physically use the same channel. For these reasons, we use a circuit-switched network. The $n$ hop/cycle property means that a braid claims the route’s resources (nodes
Figure 6.6: Logical CNOT gate between two qubits (light blue shade for clarity) of a six-qubit setup, arranged in a 2x3 lattice of vertical double-defect tiles (blue dotted lines). Underlying is the physical layer, consisting of a sea of data/measure physical qubits (shown in green/yellow). White defects are logical data and black defects are logical ancilla. (a) depicts the entire space-time volume (time goes up vertically). (b-f) each represent a slice in time, called an “event.” From a network perspective, the corners of the tiles are routers, the red-demarcated lines are channels, and black defects are messages routed in the mesh.

and links) at once when opened, and releases them when closed. We use a greedy approach of trying to place as many braids as possible, in order to reduce overall cycles.

In our braiding algorithm, we maintain a ready queue of operations whose dependencies have been met, and execute as much of them as possible in each cycle. Each operation is broken into a series of steps called events (e.g. Figure 6.6b-6.6f show the five events of a CNOT gate). Each event pertains to an open or close braid, and has a timer corresponding to how long the braid should remain in case syndromes need to be extracted. To ensure
forward progress in a busy network, we include route adaptivity of a dimension-ordered route and a drop/re-inject mechanism, both after certain timeouts.

6.6.2 Optimizing Qubit Arrangement

To reduce braid contention, we can optimize the placement of qubits on the 2D mesh before performing simulation, so that qubits which interact frequently are close to each other. Specifically, the optimized arrangement of qubits attempts to minimize the sum of $L_1$ norms between pairs of qubits involved in CNOT operations (the only non-local operations). This is done through iterative calls to a graph partitioning library, METIS [87], to separate the qubits (each represented as a vertex on a graph of qubit interactions) into two partitions, such that the weight of crossing edges is small. Relative to a naive arrangement of qubits, the optimized qubit arrangement reduces the lengths of braids, hence reducing the chance of braid collisions.

6.6.3 Optimized Braid Priorities

In our braiding algorithm, when multiple braids are eligible to be placed but not all of them fit on the mesh, the choice of priorities can have a significant impact on performance. We have devised heuristic policies to prioritize “more important” braids. It must be noted that such policies are only essential in highly parallel applications (SHA-1 and IM), where there is an initially large discrepancy between the schedule length and critical path length, due to communication conflicts.

The metrics we use to determine priority are the criticality of the braid (how many future operations depend on it), its length, and its type (whether it is a closing or opening braid). Our prioritization policies are summarized below:

- **Policy 0**: No optimization. All operations and events in program order.
- **Policy 1**: Interleave: Allow individual events to be interleaved, but keep operations in program order.
- **Policy 2**: Interleave + layout: Optimize initial qubit layout for interaction distances.
- **Policy 3**: Interleave + layout + criticality: Sort gates by highest criticality first.
- **Policy 4**: Interleave + layout + length: Sort braids by longest first.
- **Policy 5**: Interleave + layout + type: Sort by closing braids first, opening braids next.

Policy 1 (event interleaving) allows for multiple braids to progress concurrently, and at different rates. Policy 2 is the same layout localization heuristic of Section 6.6.2. To these, Policies 3 through 5 each add one important metric by which the importance of a braid may be judged: criticality, length, type. Our results show that evaluated individually, braid type causes the largest improvement, while length and criticality have comparable effects.
Finally, Policy 6 is the most successful by combining all of the metrics above. Here, we give higher priority to those braids that are closing rather than opening (so they can release network resources), and those that have higher higher criticality (to remove the bottleneck). If two braids have the same criticality, we decide based on length: shorter braids for the most critical operations, because we want to accomplish as many as possible, and longer braids for lower-criticality operations, because we want to conclude the toughest braids ahead of time.

Figure 6.7 shows our results. The blue bars (associated with the left vertical axis) show how the above prioritization policies improve performance by reducing the gap between the braid schedule lengths and critical path lengths. The last Policy is able to reduce schedule length by up to $\sim 7X$, from $\sim 1200\%$ to within $\sim 70\%$ of the critical path. The red dots (right vertical axis) shows network utilization rate (i.e. percentage of busy links). It demonstrates that better prioritization results in an almost 8-fold increase in network utilization — up to about 22%, which is an acceptable range for a highly scalable approach, comparable to similar circuit-switched networks.

![Figure 6.7: Braid simulation results for the Defect-Based Surface Code. Blue bars show schedule length can be reduced by up to $\sim 7X$, to within $\sim 70\%$ of the critical path for highly-parallel applications (SHA-1 and IM), where risk of contention is high. Serial applications (GSE and SQ) already achieve close-to-critical-path schedules. Red dots show the increase in network utilization when using these policies, up to about 22%. These improvements are achieved through interaction-based qubit layout optimizations and priority-based braid placement optimizations.](image-url)
6.7 Comparing QEC Overheads

Taking into account the combination of applications, QEC families, and technologies, we can now evaluate the design space of a realistic quantum computer. In this section, we first quantify the design regions suitable for use with each type of QEC — regions that were qualitatively discussed in the beginning of this chapter. Next, we discuss how sensitive these regions are to three important attributes: application, technology, and QEC characteristics. This allows us to anticipate future variations in each, as research in quantum algorithms, physical fabrication, and coding theory evolves.

6.7.1 Finding Regions of QEC Favorability

Figure 6.8 shows the main results of our comparative study. The x axis contains a range of physical error rates ($p_P$) for current and future quantum computers. The y axis is the size of computation to be run on those devices, measured by the compiler as the total number of logical operations. The y axis is also indicative of how reliable each logical operation should be ($p_L$): the larger the computation size, the more reliable each individual logical operation must be. Thus, various ($p_P, p_L$) points on this graph indicate various error correction strengths required to fill the gap between the physical and logical domains. We only show results for $p_P$ smaller than the threshold of both codes, where a valid comparison may occur. We also use 2-qubit and measurement gate speeds that are 4X faster than today’s technology, to account for likely future improvements. Section 6.7.3 discusses how current technology would perform.

In each figure, the solid black line delineates favorability regions of concatenated vs. planar surface codes. This boundary is independent of the application and technology, since both codes result in exactly the same teleportation patterns (the only difference is in how logical qubits are encoded). The colored dotted lines, on the other hand, each mark the transition between the favorability of concatenated Steane (below) and double-defect surface (above) codes, for a variety of applications. These lines are application- and technology-dependent, due to the entirely different models of computation (bitwise vs. braids) and communication (teleportation vs. braids) in these codes, which need to be simulated in order to yield accurate contention rates.

Figure 6.8a shows the planar flavor of surface codes is more desirable on trapped ions (i.e. the area above the solid line is larger than all the dotted lines). The cause of this can be traced to the mobility of qubits. Shuttle-based teleportation is a much lower-overhead communication method than braiding, due to the lack of need for many intermediate “channel” qubits, or long waiting periods to stabilize syndrome measurements. However, even though double-defect codes have usually been considered better for superconductors or quantum dots [18, 60, 84] (due to their amenability to nearest-neighbor interactions), Figures 6.8b and 6.8c shows that this is not always the case. The reason for this, and also the large variation among applications, is discussed next.
Figure 6.8: Comparison of the space-time overhead of concatenated vs. (two flavors of) surface codes for different quantum applications, implemented on (a) trapped-ion, (b) superconducting and (c) quantum dot technology. Design points under the curves work better with concatenated codes, while above the curve is better for surface codes. The solid black line delineates favorability of concatenated vs. planar surface codes, and is independent of application or technology since both codes result in similar teleportation patterns. Colored dotted lines are for concatenated vs. double-defect. Planar is generally a better surface code on trapped-ions (black line is lower in (a)), due to the efficiency of shuttling and teleportation compared to braiding. On superconductors and quantum dots, braiding is generally better than teleportation with swaps. The exceptions are highly-parallel applications (SHA-1, IM), because congestion hurts braids more.

6.7.2 Sensitivity to Application Characteristics

Figure 6.8b and 6.8c show that even though double-defect codes are suitable on superconductors and quantum dots for serial applications such as GSE and SQ (as expected, since braids are more efficient than swaps), highly-parallel applications such as SHA-1 and IM should actually use planar surface codes instead. The cause of this can be traced to two sources: (a) from a computation side, the ability to perform SIMD-style operations means...
that parallel instructions may be simultaneously executed inside SIMD regions for planar surface codes, while double-defect qubits cannot be brought in the vicinity of one-another in the same manner. (b) from a communication side, higher parallelism means higher communication contention, but the sources of contention are different. In planar surface codes, ancilla are communicated on the mesh, but since they have very loose data dependency, their distribution may be smoothed to eliminate contention. No such advantage exists in communicating braids, since they are tightly tied to the dependency of data, reducing their chance of avoiding conflict. This is also manifested in Figure 6.8a, where the curve for more parallel applications are higher.

For the Ising Model (IM) application, we have used two variations — with medium and maximal flattening. This is our most parallel application, yet the amount of parallelism discovered depends on the amount of applied flattening at compilation time. More code flattening creates more parallelism, consistent with the upward movement of the boundary.

### 6.7.3 Sensitivity to Technology Characteristics

A second factor that can impact regions of code favorability the physical machine characteristics. Specifically, we study the impact of gate latencies (in all three technologies) and shuttling latency (in trapped ions), since they are highly variable as technologies continue to mature. In general, 2-qubit gates are slower than 1-qubit gates; however, the ratio of latency can evolve.

As Figure 6.9 demonstrates, improving the 2-qubit to 1-qubit relative latency (i.e. CNOTs becoming less slow compared to single qubit gates) benefits concatenated codes on superconducting or quantum dot technology, while benefitting surface codes on trapped-ion technology. The reason is that a major source of delay in superconductors and quantum dots are swaps, which are just a series of CNOTs. Relatively better CNOTs lead to relatively better concatenated codes and planar codes, as compared to double defect. However, in trapped ions, this defining difference does not exist, and better CNOTs only cause improvements in error correction cycle time. Since the surface cycle time is more heavily dictated by CNOTs, this improvement benefits surface codes more. Furthermore, we see in Figure 6.9a that improved shuttling speeds favor concatenated codes, as the boundary moves up.

### 6.7.4 Sensitivity to Allocated Bandwidth

Another important aspect of a QC design is the amount of resources we are willing to spend to build a computer. Specifically, increasing communication bandwidth requires investing more area, either in the form of increased qubits in superconductors and quantum dots, or more traps in trapped ions. Figure 6.10 shows the sensitivity of region boundaries to increased shuttling and swapping bandwidth, with 2X and 4X increases over single-qubit-wide channels. Braiding bandwidth remains unchanged since multiple braids on the same link are disallowed. We see that concatenated code performance improves across the board in trapped-ion computers, owing to more bandwidth allocation for ancilla pre-distribution. The effect is non-linear in superconductors and quantum dots, however, where
Figure 6.9: Sensitivity of the favorability boundaries to gate speed variations, for one serial (SQ) and one parallel (IM) application. Three variations are included: current, improved 2-qubit to 1-qubit gate latency ratio, and improved measure to 1-qubit gate latency ratio. Red and blue lines belong to the SQ and IM applications respectively. CNOTs have the largest effect, especially in superconductors due to heavy influence on swap latencies. Measurements have little effect, since they improve both sides of the comparison almost equally.

the increase in bandwidth comes at the price of many extra qubits (recall that channels in superconducting networks are filled with stationary qubits).

6.7.5 Sensitivity to QEC Code Characteristics

Research on optimizing surface codes have achieved considerable success recently [59, 61, 76, 183]. Our work lays the foundations of doing such integrated comparisons, even when specific implementations continue to evolve. For example, [183] proposes a variation of double-defect tiles that are 2X compressed, causing a direct 2X improvement in total space-time volume. We find that this only causes minor adjustment of the boundaries.
Figure 6.10: Boundary sensitivities to increased channel bandwidth of shuttling and swapping. Red and blue show SQ and IM respectively. Concatenated code performance on trapped ions consistently improves with more bandwidth, whereas conflicting time and space make superconducting/quantum dot changes more contingent on specific design points.

In summary, our results highlight accurate modeling and optimization of communications as a central part of quantum system design. This is demonstrated by the fact that in Figures 6.8 – 6.10, the solid black lines are very robust against various technology or architectural variations, due solely to the fact that the effect of communication in those comparisons are similar and thus normalized. In all other cases, parameters that affect communication also affect the comparison boundaries.
6.8 Future Extensions

6.8.1 Module-Boundary-Aware Qubit Layout Optimizations

The optimization of qubit layouts discussed in Section 6.6.2 are currently based on knowledge from qubit interactions within each module. In that sense, they are intra-module-optimized; however there is still a potentially large cost from shuffling qubits around at each module boundary. It remains as an area of future work to investigate methods for global optimization. Such optimization should take into account two costs: the costs of qubit interactions within the computational steps of every module, and the cost of moving around qubits should two consecutive modules require different initial placements. Optimizing the partitioning of several graphs is an interesting and nontrivial direction to explore.

6.8.2 3-D Packing of Braids

In our treatment of braids as network messages, we have sought to increase scalability. However, this comes at the cost of missed opportunities for optimization. For example, it is impossible to go backwards in time in simulating messages, whereas a real braid could in fact do so (the space-time volume does not care about any of its dimensions; any optimization is allowed in any direction, similar to topological compaction). This could be addressed in 3-dimensional braid packing methods. Doing so in an automated yet scalable fashion remains a future project.

6.9 Related Work

The central aim of our work is to systematically evaluate the tradeoffs of using various forms of quantum error correction. While prior work has often relied on optimizing a particular QEC/technology choice [60, 84, 182], our work provides insights into the conditions where such choices are warranted.

Suchara et. al. [174] have performed, to our knowledge, the only system-level comparison of surface codes versus concatenated codes. Given a (manually-estimated) gate count and parallelism factor for an application, plus parameters for the underlying technology, their model quickly estimates time and space overheads. Our work extends theirs by performing a much more rigorous evaluation of the details of mapping to the architecture, taking into account dependencies of operations, mapping of qubits, and contentions of communication. We do not assume that ancillas are perfectly prepared and available when needed, rather develop methods to ensure their efficient distribution. Furthermore, we also evaluate planar codes as an important hybrid scheme, demonstrating their usefulness under certain conditions that were not known before.

Prior work has primarily focused on hand-optimizing the compression of 3D surface code volumes, using topological transformation [50, 59, 61]. For example, [49] creates a game in which players try to manually compress 3D “pipe” topologies, with direct application to optimizing an equivalent braided circuit. Even though these techniques achieve very good results for small circuits, they are difficult to scale. Some recent works have
begun addressing the issue of automation by performing automated constructions of 3D volumes [138, 141, 142]. However scalability and extending to all gates remains an area of future work in these methods. Our proposed alternative is to map the problem to a networking problem, and use routing techniques which are overall more scalable.

Pedram et. al. [146] propose a qubit layout optimization similar to our graph partitioning approach of Section 6.6.2, combined with mapping to the Minimum Linear Arrangement problem, but for one-dimensional qubit layouts. Generalizing this to the two-dimensional case with similar efficiency remains an open problem.

6.10 Summary

This chapter has laid out a comprehensive study of the leading QEC and technology candidates to implement a future quantum computer, highlighting the tradeoffs of different design choices. We conclude that with device error rates that do not currently meet the Steane error threshold, surface codes are the natural choice due to their high threshold. However with increased reliability, and especially for smaller problems that are likely to be solved in the near future, concatenated codes seem to be the better choice. Further still, looking far into the future and the possibility of solving problems such as the 2048-bit Shor’s factoring algorithm that can break classical cryptography, it is very likely that the asymptotic advantage of surface codes will make them the best choice. We also conclude that the planar variation of surface codes is a better choice for trapped-ion systems, and even superconducting or quantum dot systems when they run highly parallel algorithms such as the Ising Model.
Chapter 7

Conclusions and Future Directions

7.1 Conclusions

This thesis has explored the topic of system design for quantum computation at different levels of compilation, mapping and scheduling. We have leveraged many methods in classical system design such as SIMD operations and circuit-switched networking, while carefully considering the special requirements of quantum computation that necessitate new approaches, such as constraints on qubit communication or the central importance of error correction.

The central aim of this thesis has been to investigate the following questions: how many resources — defined as the number of qubits and gates — are required to execute non-trivial quantum applications on realistic future quantum machines; to what extent can these resources be optimized; and what is the best choice among many possible design parameters for building a quantum computer, notably in choice of quantum error correction, the most expensive aspect of the implementation. To answer these questions, this thesis has introduced an end-to-end toolflow for compilation, mapping and scheduling of large-scale quantum applications on quantum technologies. At each stage, we have discussed the challenges that arise, and offered some solutions for them. We have evaluated our proposals on a broad set of applications and some leading quantum technologies.

High-level quantum algorithms are designed without regard for the underlying implementation. It is therefore critical to use tools that can automatically evaluate a given application prior to its intended run on the technological target. This will not only provide insights about bottlenecks in the application (such as the most resource consuming part of the circuit), but will also help choose the best design among several options (such as choice of error correction or communication method).

7.1.1 Optimized Compilation, Mapping and Scheduling

Since it is likely that quantum algorithms will need to be evaluated by scientists and algorithmicists, we have designed the Scaffold programming language to be intuitive and close to ubiquitous programming paradigms such as C. Scaffold is an extension of C with new quantum constructs — it therefore lends itself well to leveraging existing compiler
infrastructures and optimization routines. While Scaffold is easy to use and compile (many mature compiler optimizations can be applied due to its proximity to C), it does not yet provide strong program verification capabilities. We discuss the difficulty of debugging and verification in quantum programs: simulations are by definition impossible, and checkpoints amount to collapsing the quantum state. We then propose possible future uses of type systems to track entanglement and no-cloning constraints, enabling the programmer to write more accurate code.

A quantum program consists of portions representing the quantum computation (the quantum gates), and portions representing classical control. The control will execute on a classical computer, which will instruct the quantum chip about operations it must execute. However, there is a tension between the large quantum algorithms that we would like to run (quantum supremacy is achieved only in larger programs), and the relative paucity of quantum resources that are available. This tension manifests itself in compilation: we would like to perform aggressive optimizations to reduce resource usage, however that is often achieved when we specialize the code (through procedures such as function cloning and loop unrolling), which will increase its size. Increased code size will slow down the compilation process, and will be hard to load on chips which often have small instruction caches stored in very cold environments (to reduce heat dissipation). In the quantum domain, code specialization is not only possible, but also critical to perform. It is possible because all inputs to a quantum program are known, and thus the entire code may in principle be linearized to a sequence of gates at compile time. It is, however, important to do as well because linearized code is easier to optimize, and also less ambiguity will be left for runtime. For example, if an arbitrary rotation angle is not resolved and converted to a sequence of elementary gates at compile time, this process will have to be done at runtime, potentially introducing delays and ruining the schedule of gates. Therefore, it is essential to find a middle ground between fully linearized, non-scalable code on one hand, and non-optimal code and schedules on the other.

In this thesis, we present the ScaffCC compiler which tackles the problem of compilation scalability at two levels: the choice of compiler output and the process of code generation itself. The compiler output that we target is the quantum assembly language of QASM. We introduce QASM-HL, a variant with hierarchy of modules and loops, that still successfully resolves important parts of the code such as angles of rotation, but remains much more manageable in size. QASM-HL has linearized code within function bodies and within loop bodies, which can be independently optimized and scheduled, and then synthesized together into a wider coarse-grain schedule. Moreover, at the code generation stage, we recognize that even though many mature classical compiler passes can easily be used for static code specialization — such as loop unrolling, procedure cloning and constant propagation — this will cause an explosion in space, which will in turn cause thrashing in memory and extreme slowdown of compilation time. Therefore, we propose an alternative dynamic compilation method, called the instrumentation-driven approach. This approach only runs one compiler pass to instrument the code into a regular executable, to be run on a classical machine. This classical code will generate the expected QASM-HL output upon execution. In effect, the instrumentation-driven approach relegates the task of compilation from static compiler passes to dynamic execution on the classical processor, keeping the code size in check. Again, our choice of a C-like language has helped to easily convert
Scaffold code to one that is executable on a classical machine, without major transforma-
tions.

The above language, compilation and code analysis stages are done at the logical level; i.e. before any error correction is applied. This is an important optimization and analysis stage: error correction is a very expensive requirement of quantum computation, and will end up dominating the computation itself. Even though there are many proposals for optimizing error correction schemes so that they take up fewer resources, an optimized logical-level circuit will yield multiplicative benefits: fewer qubits and gates must be protected against errors, and those that will need to be protected can afford a weaker form of correction since the overall circuit is smaller — computation would need to be sustained for a shorter period.

### 7.1.2 Understanding Quantum Algorithms

The logical-level frontend of our toolflow may be used to compare various algorithm designs as well. As a case study, in this thesis we evaluated several proposals for quantum phase estimation (QPE), a key technique that is used as a subroutine in many quantum algorithms, such as Shor’s algorithm and the Ground State Estimation algorithm. We evaluate tradeoffs between two established methods for QPE: Kitaev Hadamard Tests (KHT) and Approximate Quantum Fourier Transform (AQFT). These methods achieve the same goal through the use of many trials or many rotations, respectively. We further evaluated a more recent method called the Arbitrary Constant Precision Algorithm (ACPA), which is designed as a middle ground between KHT and AQFT in terms of number of trials and rotations. Each of these methods require varying amounts of hardware resources (i.e. logical qubits) and time to execute. Using our ScaffCC compiler, we are able to characterize a large design space for each of these methods, based on what parallelism level the quantum computer supports, how many qubits it contains, and what the target execution time is. We find that KHT is a highly parallel method, but requires many qubits to execute well. In contrast, AQFT is serial but can execute with relatively modest numbers of qubits. In all cases, ACPA is a middle solution. We also evaluate how each of these methods would fare as part of the larger algorithm to be executed: Shor’s or Ground State Estimation. Overall, such benchmark characterizations would allow a quantum computer designer to choose the most suitable QFT method, based on the higher-level application characteristics and the lower-level hardware support (qubits, parallelism, time).

### 7.1.3 Exploring Design Space Tradeoffs

Finally, this dissertation has undertaken a comprehensive study of resource overheads for implementing a cross-product of various applications, QEC codes, and technologies. Since various QC proposals are being investigated simultaneously and there is no consensus at the moment on the best choice of technology or QEC code, such studies are valuable at providing early insights and guiding future research and development. Our study is done through a detailed scheduling of computation and communication cycles, using network and congestion simulation tools developed for this purpose. A major theme has been the
reduction of space-time overheads by focusing on communication optimization. The peculiarities of quantum communication make them different from classical methods — for example, there is no fanout, and qubit data can exist in only one place at a time. We discussed teleportation and braiding as two important methods for communication quantum states, and proposed methods to optimize each of them.

We observe that communication challenges can be addressed at two levels: first through reducing the overall number of communications at the logical level, and second through reducing the overhead of necessary, remaining communications at the physical level.

At the logical level, we optimize qubit mappings to reduce the total number of communications. In particular, the Multi-SIMD architecture that we have proposed can exploit data- and instruction-parallelisms well, but without proper scheduling it can suffer from a large inter-region communication overhead. Therefore, we carefully orchestrate qubit mappings and operation sequences on each SIMD region such that the longest path of computation remains pinned to one region. This reduces inter-region teleportation of qubits. Conversely, in the tiled architecture of surface codes, where each logical qubit occupies one part of the architecture and qubits communicate using braids, we map qubits to each tile such that highly-interacting qubits remain close to each other. This creates shorter braids, mitigating contention rates. The key idea in both of these architectures is that having full knowledge of the program provides us with two useful, fully-determined graphs: the graph of operation dependencies, and the graph of qubit interactions. These can be used to create intelligent maps and schedules that remove unnecessary communications.

Furthermore, we have to ensure that those communications that are necessary do not incur large overheads at the physical level. We achieve this through simulations of communication networks and algorithms that reduce network contentions. In teleportation networks, this can be achieved through smoothing of EPR pairs, since they do not obey dependencies in data and can be pre-distributed at any time. In braiding networks, we propose several policies that prioritize braids and reduce their conflicts. The mapping of braids to networks is itself a contribution of this thesis, since in prior works braids have been optimized as topological 3D volumes, with poor scalability.

7.1.4 Major Observations

A main insight from our comparative study is that teleportation-based codes (examples of which include concatenated codes and planar surface codes) are more suitable when highly-parallel applications are run: the EPR distributions can be pipelined, removing them from the critical path. This result is important, since previously it was believed that braid-based surface codes are ideal for superconductors, due to nearest-neighbor constraints. This work indicates that with enough parallelism, even superconductors must use teleportation.

This thesis has demonstrated the importance of paying attention to the application a quantum computer is designed to implement, a factor largely ignored in prior work. The high gain of running a quantum application is such that each real quantum machine will likely be dedicated to solving a specific task — breaking large RSA encryption keys or finding the ground state of large molecules. We have shown that taking into account the nature of the application may affect the design. We have shown two examples for this: choosing a particular implementation of the quantum phase estimation (QPE) subroutine
based on the application, and choosing the type of error correction and communication based on the application.

7.2 Future Directions

Owing to considerable theoretical successes over the past two decades [8, 60, 67, 97], the challenge of building a quantum computer is now mainly an engineering one. The main goal is to scale up available physical qubits (build bigger computers), while simultaneously reduce the space-time overheads of running a given application (mitigate resource usage). The latter has been the focus of this thesis. Much work remains to be done in all areas of algorithm design, system design and device manufacturing — but increased investment and continued research will eventually pave the path for large-scale, general-purpose quantum machines.

While this thesis targets a universal gate set \((X, Z, CNOT, H, T)\), which in theory can implement any application and can itself be implemented on any technology, the chosen gate set might not be native on all machines. For example, trapped ion-based quantum computers typically implement \(R\) and \(XX\) gates natively [116], and other gates would be implemented based on them. Future work must take this into account as a last mapping phase in the software stack, and optimize the scheduler based on an accurate usage of gates. Other technological peculiarities may also be handled at this stage. For example, two trapped ions can typically interact by turning on two lines that address them — this means even stationary qubits can interact in this way, unless the two qubits are too far away in which case movement becomes necessary. In this thesis we have made the simplifying assumption that any two distant qubits must interact by movement. Consideration of such special cases will yield a more accurate model of particular technologies.

In the mapping and scheduling phase, an important area of future work is to incorporate inter-module knowledge to improve performance. We have mentioned that in order to manage scalability, it becomes necessary to modularize a large program. While as a first pass we have chosen to map and schedule each module separately, this choice may incur extra cost at module boundaries where qubits must be moved around to accommodate the next module’s mapping. It remains an open area of research to optimize mappings in such a way that minimizes this movement, by looking ahead at the larger context of the program.

Lastly, while this thesis provides valuable insights regarding the paths that must be followed in the design of future quantum computers, the most immediate need of quantum computing system design is to fit as much computation power in as few qubits as possible. Therefore, issues of scalability become less important in near-term compilers, while issues such as aggressive qubit reuse gain more importance. Design of compilers and architectures that can allow maximal qubit reuse is a vital research topic for the next few years.
Appendix A

Background on Quantum Computation

This appendix aims to acquaint the reader with some of the key concepts underlying the assumptions, methodology, and results presented in this dissertation. It is intended to review the basic pertinent elements of quantum computing. However, it is by no means a rigorous exploration of the subject. We refer the interested reader to other publications for a more in-depth introduction [4, 89, 132].

A.1 A Quantum Theory of Nature

Quantum computation is the result of using quantum mechanical systems as a means of storing and processing information. These systems, examples of which include subatomic particles or very low-temperature matter, do not follow the rules of Newtonian mechanics. Instead, a new theory known as quantum mechanics was developed to explain their behavior.

Consider the experiment conducted by German physicists Otto Stern and Walther Gehr-lach in 1922 [65], as shown in Figure A.1. They observed that excited atoms do not occupy a continuous spectrum of angular momentum as predicted by classical Newtonian physics, but instead exhibit two distinct quantized states. This is now known as positive and negative spin; we refer to such two-level systems as a quantum bit or qubit. Qubits are the basic carriers of information in quantum computers, and are today manufactured using a multitude of technologies, among them superconductors, ions, and photons.

Further experiments reveal more remarkable properties of qubits. Consider the double-slit experiment first proposed by Thomas Young. If a single electron is shot at a filter containing two slits, it is expected that each electron pass one of the slits at any given time, and leave a mark on the fluorescent receiving screen. However, a surprising interference pattern is instead observed, as would be expected if waves and not particles were passing through. This suggests that electrons possess a wave nature in addition to a previously posited particle nature, and that even single electrons can interfere with themselves (pass through both slits simultaneously). This leads to the definition of superposition, a phenomenon that suggests a qubit can (probabilistically) exist in two superposed states at the same time. The reason we only observe superposition of states in subatomic or very cold settings is that qubits tend to lose their states through interaction with outside systems,

119
Figure A.1: Stern-Gerlach experiment. Silver atoms are excited in a furnace (1), causing beams of single atoms (2) to pass through a magnetic field (3). A classical Newtonian prediction would predict atoms to occupy a continuous spectrum, based on varying deflections based on their charge. However, the actual observation is that only two distinct deflections are made — up or down depending on the atomic spin (5). Spin is an inherent property of the atom and is quantized.

in a process known as *decoherence*. That is why everyday macroscopic systems behave according to the laws of classical physics.

Figure A.2: Double-slit experiment. Individual electrons create interference patterns on the screen, which means they behave like waves (with peaks and troughs). Furthermore, it shows that electrons can pass through both slits simultaneously in order to create an interfering effect. This suggests that electrons exist in a superposition of their possible two-level states.

### A.2 Qubits and Gates: The Quantum Circuit

Any system that can exist in two simultaneous states can act as a qubit. We typically define two orthogonal basis state as $|0\rangle$ and $|1\rangle$ states. A qubit is not confined to a deterministic binary state, unlike classical bits. At any time, the state of a qubit can be a linear combination of the basis states denoted as $|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$, where $\alpha$ and $\beta$ are complex numbers subject to $|\alpha|^2 + |\beta|^2 = 1$. Therefore, a quantum state can be thought of as a vector of norm 1, with an arbitrary phase—a concept which can be extended to $n$ qubits by thinking
of the state of an \( n \)-qubit system as a unit vector in the \( 2^n \)-dimensional space. This can be represented as a point on the surface of a \( 2^n \)-dimensional sphere, commonly referred to as the *Bloch sphere* (Figure A.3).

![Bloch sphere diagram](image)

Figure A.3: The Bloch sphere is a good visualization of the quantum state space, depicting the space of all possible states of a single qubit. The state vector of a single qubit, \(|\psi\rangle = \alpha|0\rangle + \beta|1\rangle\), can lie anywhere on the surface of the sphere. Single-qubit quantum operations (also known as gates) are akin to rotating this vector on the surface of the sphere.

A quantum computer should ultimately provide an answer to a classical query. This is done by mapping the problem to the qubit state space, manipulating those states, and finally measuring them to read the result. In this regard, qubits are very similar to classical probabilistic systems, with the key difference that instead of probabilities they represent amplitudes, which may be negative too. Exploiting this property is the key to devising a successful *quantum algorithm* to solve a particular problem: we interact qubits in such a way that amplitudes contributing to the (desired) right answer magnify each other (constructive interference) and those contributing to all other (undesired) outputs cancel each other (destructive interference).

A quantum algorithm is made up of many individual quantum *operations*. Qubit states are manipulated through these operations. Looking at the Bloch sphere, it is immediately obvious that there are infinitely many possible operations. All quantum operations are merely norm-preserving rotations, which move the state of an \( n \)-qubit system along the surface of the Bloch sphere.

Quantum operations modify superposition states in order to solve computational problems. It is precisely because of this joint manipulation of states that some algorithms can perform faster than their classical counterparts. For example, a quantum mechanical system with 2 qubits can be simultaneously representative of the four states \(|00\rangle, |01\rangle, |10\rangle\) and \(|11\rangle\). Therefore, operating with \( n \) qubits gives the opportunity of manipulating \( 2^n \) vectors in the Hilbert space, allowing multiple paths to be explored simultaneously by the algorithm.

We can inquire about the state of a qubit by observing it, or performing *measurement*. Measurement of a qubit results in a classical output of 0 or 1, with probabilities directly proportional to the amplitudes — the higher the coefficient of a state vector in the superposition state, the higher the chance of collapsing into that state. Measurements usually occur
at the end of an algorithm, since a quantum algorithm must in the end provide a classical
answer to a classical query, only using quantum bits and operations during the computa-
tion. Therefore, quantum algorithms manipulate quantum states of qubits in such a way
that increases the likelihood of measuring the desired answer in the end.

While quantum algorithms can be designed using an infinite repertoire of quantum op-
erations, quantum technologies can physically implement a limited number of operations.
The good news is that arbitrary operations can be approximated by the sequential applica-
tion of operations from a small standard set. This is akin to a classical processor’s instruc-
tion set. Such physical operations are typically called gates. This terminology should not
be confused with the notion of gates in classical hardware design – whereas classical gates
are physically instantiated and static, quantum gates are applied dynamically (for example
using voltages or lasers), and can be altered throughout the course of program execution.

A set of frequently used gates are given special names:

- Bit flip ($X$) gate, phase flip ($Z$) gate or both ($Y$) gate: Reflections about the orthog-
onal axis of $X$, $Z$, $Y$
- Hadamard ($H$) gate: Conversion from the $X$ basis to $Z$ and vice-versa
- $S$ gate: Rotation about the $Z$ axis by $\frac{\pi}{2}$ radians
- $T$ gate: Rotation about the $Z$ axis by $\frac{\pi}{4}$ radians
- Controlled-NOT (CNOT) gate: Bit-flipping a target qubit based on the state of a
control qubit

Following the analogy of a classical logic circuit, a quantum circuit can be used to rep-
resent a collection of qubits and gates, illustrating the timeline of operations performed in a
quantum algorithm step-by-step. In these circuits, quantum single- or multi-bit operations
are shown with boxes labeled with the operation. Measurement is shown with a “meter”
symbol. Single wires denote a qubit, and double wires denote classical bits which are the
outputs of measurement. Finally, “controlled” operations are shown with a wire connect-
ing the controlling qubit to the target operation. The meaning is that the operation will be
performed if and only if the controlling qubit is in the $|1\rangle$ state. The reader can refer to
Figure A.4 for an example of quantum circuit diagram. This is a well known circuit which
performs “quantum phase estimation,” a necessary component of many larger quantum al-
gorithms. Chapter 4 elaborates on it as a case study of many ideas within this thesis. The
use of circuit diagrams to show the computation flow is analogous to illustrating a classical
computation using compiler-level control/dataflow graphs.

Finally, we must consider the model of execution on an actual quantum computer, after
a high-level quantum algorithm is converted into a sequence of gates in the circuit model.

While specific technologies for an implementation vary widely, Figure A.5, known as
the QRAM model, shows a general approach in which quantum computations execute on
a co-processor unit controlled by a classical computer [101]. In QRAM, quantum computa-
tion occurs in the joint presence of a classical and quantum core. The quantum core is
much like a co-processor; it is where the qubits are stored, undergo interactions with each
other, and have their states manipulated by quantum operations. These quantum operations are orchestrated by the classical core. Once a measurement operation is issued, the result is a classical value which is received back at the classical core and processed to find the overall result of computation. Other models such as the Turing machine or the adiabatic model have been shown to be theoretically equivalent [9, 197].

Within the quantum processor are several operating zones or gates, which can each perform distinct operations in parallel on distinct qubits. Each of the operating zones can be controlled to execute any of the gate types this machine has chosen to implement. The gate they implement can be changed from cycle to cycle via control bits sent over from the classical computer. One can think of a quantum computer’s operating zones being analogous to an ALU in a classical computer; with a small number of bits, ALUs can be controlled to perform an addition one cycle, subtraction the next. The runtime of a quantum computer is, to first order, the number of these gate operations (i.e., cycles or steps) that need to occur in sequence in order to complete the computation (several may happen in parallel).

The importance of well-designed classical software for quantum computation emerges from this picture. First, compilation software is needed for lowering abstract quantum algorithms to executable physical gates, a trace of which will be stored on the classical computer. Second, high performance software is needed during the execution phase, to issue instructions, gather measurement results, and detect and correct errors. The latency of this “control” software must be such that it does not introduce stalls in the sequence of quantum operations.

### A.3 Quantum Algorithms and Applications

A quantum algorithm is an algorithm that makes use of access to qubits and operations on them to solve a particular problem, preferably in a manner that is more efficient than any treatment of that problem on a classical computer. The answers obtained from the execution of a quantum algorithm are often correct only probabilistically. In order to be sure of the answer, quantum algorithms are often run several times and a histogram of the
Figure A.5: QRAM model of computation. The control is classically orchestrated by the classical processor, and classical measurement results are received and processed by it. The quantum core is where the quantum operations (gates) take place.

results obtained determines the correct result, with high probability. It is also often the case that checking the answer is efficient, so any obtained result can be verified quickly for correctness (e.g. checking the correctness of factoring amounts to trivial multiplication).

A quantum algorithm can be viewed as a series of state transformations. A valid quantum transformation takes one \( n \)-qubit state to another. That means, a quantum operation is a mathematical transformation of one vector in the \( 2^n \)-dimensional Hilbert space to another — a unitary matrix operator.

While the main motivation behind quantum computing is the supremacy of quantum algorithms, in many instances it is hard to fully determine quantum supremacy for a given problem. One reason is that often it is simply not known whether a best-known, currently-available classical algorithm achieves the lowest complexity bound, or whether better classical algorithms may be devised in the future. Factoring is a good example of this. Moreover, asymptotic scaling must be shown in practice as well as in theory, due to the many factors (computational gates, memory, communication) that could affect a physical realization in both the classical and quantum domains. This is why highly-optimized versions of both classical and quantum algorithms must be compared. For example, a recent study of the quantum annealing algorithm [57] performed on quantum hardware and similarly simulated on classical hardware [120] did not find conclusive evidence of quantum speedup [154].

It is, however, still the case that there is very good evidence for several quantum algorithms surpassing any possible classical solution to the same problem. Specific features of quantum systems are key in granting them fundamental supremacy. Chief among these are quantum interference and entanglement. The following offers two examples of quantum algorithms that exploit these properties:

A.3.1 Role of Quantum Superposition: Random Walk Example

Figure A.6 illustrates two random walks algorithms; on the left hand side, a walker moves left or right, each time with probability \( \frac{1}{2} \). The probability of ending up at each point relative to the origin is shown over time. We see that, as expected, this probability tree is balanced. However, the quantum scenario is markedly different. Movement between states must be specified by a unitary matrix. One norm-preserving matrix to achieve this is the following matrix, whose repeated application yield the right-hand-side probability tree of
Figure A.6. \[
M = \begin{bmatrix}
\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\
\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}
\end{bmatrix}
\]

Figure A.6: Classical (left) vs. quantum (right) random walk. Quantum interference results in a different set of probabilities for arriving at each position. The standard deviation — or the space explored — in the quantum domain is larger, causing faster exploration of space (Figure adapted from [15]).

This simple example illustrates a fundamental difference of quantum and classical algorithms; namely the fact that quantum interference causes a different behavior than classical probabilities predict. An intuitive advantage of quantum random walk over classical can be seen by looking at the expected standard deviation, as an asymptotic function of the number of steps. In the quantum realm, this is quadratically larger, yielding faster space exploration.

### A.3.2 Role of Quantum Entanglement: Shor’s Example

Perhaps the most famous algorithm in quantum computing, and the main attractor of attention to the field is Shor’s factoring algorithm, proposed in 1994 by Peter Shor to factor large numbers in linear time [165].

The intuition behind this algorithm is that finding the constituting prime factors of a large integer can be reduced to finding periods in a closely-related sequence of numbers. And period-finding for a long sequence is more efficient on quantum computers than classical, since a period is a shared property of all the numbers. Hence, a quantum computer, using its ability to operate on multiple numbers simultaneously, can arrive at the period much faster than a classical computer which has no choice but to traverse the sequence one by one.

Consider the problem of factoring the number \( N \). Euler’s theorem [56] suggests that if \( N \) is the product of two primes \( p \) and \( q \), then the modular exponentiation sequence in Equation A.1 is periodic (given \( x \) and \( N \) are co-prime). In particular, the period is a divisor of \( (p - 1)(q - 1) \). With some further classical processing, \( p \) and \( q \) can be recovered.
Figure A.7: Shor’s algorithm. Two main stages are identified: Obtaining an equal superposition of modular exponents as in Equation A.2, and applying an inverse QFT to obtain the period. In addition to superposition, the entanglement of top and bottom qubit registers are instrumental in efficiently determining the period.

\[ x \mod N, x^2 \mod N, x^3 \mod N, x^4 \mod N, \ldots \]  

(A.1)

Exploiting this property, Shor’s algorithm tries to find the period of the sequence in Equation A.1 by first creating a superposition of all numbers \( x^k \mod N \), and then finding their period. Two qubit registers are used: a top one to contain all positive integers \( k \) and a bottom one to contain the corresponding \( x^k \mod N \). Figure A.7 illustrates the schematics of the algorithm. The modular exponentiation part sets up the superposition of Equation A.2, while a Quantum Fourier Transform (QFT) on the first register alone discovers the period of repetition.

\[ |\psi\rangle = |0\rangle |x^0 \mod N\rangle + |1\rangle |x^1 \mod N\rangle + |2\rangle |x^2 \mod N\rangle + \ldots \]  

(A.2)

Looking at the circuit of Figure A.7, a curious property is seen: even though we’re interested in finding the period of numbers \( x^k \mod N \) which are stored in the bottom register, the QFT is applied to the top register. This is due to an important property of multi-qubit systems known as entanglement.

Two (or more) qubits can be placed in an entangled state when they are operated on simultaneously. This causes the qubits to go into a joint state which is not separable into individual component states (pure states). As a result, the qubits are tied together: operating on one affects the other. For this reason, the period of Sequence A.1 is quickly obtainable by interacting all qubits of this circuit and extracting information that is representative of all of them.
In fact, entanglement is a necessary feature of all algorithms that offer an exponential speedup over classical methods, because pure states can be efficiently simulated on classical computers [85, 186].

In brief, superposition and entanglement are the main properties that distinguish computing in the quantum realm from the classical realm, resulting in quantum supremacy. However, quantum supremacy must be proven in practice as well. Quantum algorithms have an idealized view of the hardware, giving rise to a central motivation behind this thesis for developing realistic evaluation infrastructures.

A.4 Quantum Device Technologies

The algorithms discussed in the previous section must be executed on a host machine — a machine capable of providing the physical qubits and operations necessary to carry out the computations of the quantum circuit.

A number of technologies have been proposed for implementing a quantum computer, from solid state circuits to electric charge to particles of light. However, in order for a technology to be deemed viable for general purpose quantum computing, it must satisfy certain criteria. These were first formulated by David DiVincenzo [53].

A.4.1 DiVincenzo Criteria

- **A scalable number of well-characterized physical qubits**: The state of a qubit must always remain within the space allowed by the superposition of its two levels. Furthermore, running quantum algorithms that outperform classical algorithms generally requires large problem inputs. This means that the system must be scalable, especially in terms of operating a large number of qubits.

- **The ability to initialize the state of qubits**: In order to start a quantum computation, we need to know the definitive start state of qubits. This entail initialization: placing them in a known state, typically the \( |0\rangle \) state. The initialization process is often done through qubit “cooling” — the process of letting the qubit anneal to its ground state.

- **Long relative qubit decoherence times**: As mentioned in Section A.1, all matter that exhibits quantum behavior superposition tends to lose it over time. This temporal transition is governed by the Schrödinger wave equations [160], and is the reason behind macroscopic objects behaving classically. To combat this degeneration into the classical domain, we can employ techniques of error correction. However, the decoherence time must be long enough so that such corrective operations can be performed.

- **A universal set of quantum gates**: Quantum states, as seen on the Bloch sphere, are analog. They can occupy a continuous spectrum within a very large Hilbert space, and thus there are infinite valid operations to map them to each other. However, any technology will be able to accommodate a select number of operations. It has been shown, however, that all arbitrary operations may be constructed (to arbitrary
precision) using a chain of operations chosen from a very small set of gates, known as a *universal* set of gates. In order to implement general purpose quantum computing, a quantum technology must admit at least a universal set of gates. This is similar to the concept of a NAND gate in classical logic, which can implement arbitrary classical operations. In quantum computing, a commonly used set of universal gates are the Hadamard, CNOT and T gates, defined previously.

- **A qubit-specific measurement capability:** In the same way that state initialization and the induction of superposition is the entry point from the classical domain to the quantum domain, qubit measurement terminates quantum superposition and brings us back to the classical domain. This is often the last step in a quantum circuit — one which provides the final answer to the problem. It is also often a necessary part of error correction: qubits are continuously measured as a means of obtaining error syndromes, a glimpse into whether errors have occurred and whether they must be corrected.

Surveys of the state of the art in quantum technologies can be found in [107, 185]. In this thesis, we focus on three leading technologies, all of which have made significant progress towards realizing the DiVincenzo criteria — trapped ions, superconductors and quantum dots. Below is a summary of each technology’s characteristics.

### A.4.2 Trapped Ions

A quantum computer based on trapped ions, as proposed by [41], encodes quantum information in the electronic state of charged atomic ions confined within electromagnetic traps. Gate operations consist of applying laser beams or microwave pulses to the ions. The ions are trapped between charged plates (traps), which allow for fine-grain control of the position of the qubits. This technology has been demonstrated to have excellent qubit stability and high-fidelity gates [20, 24, 72]. From a control perspective, it is attractive due to its ability to operate at room temperature, as well as the ease of control on individual ions. These ions can be confined in free space with very high precision, and are almost identical, facilitating multi-qubit addressing [125]. However, the operating frequency of this technology is not high, currently on the order of 10-100 KHz [16]. A major factor distinguishing trapped-ion qubits is their mobility. Qubit transportation can be achieved by changing the operating voltage of traps [93], to move qubits in straight lines or turns [74, 194] with high fidelity and speeds on par with individual gates [29, 188].

Trapped-ion technology is attractive due to its very high control precision and the high stability of atomic ion energy levels. It provides well characterized qubits, long decoherence, and the ability to initialize, measure and perform a universal set of gates with high accuracy [25, 38, 41, 63, 111, 126, 152, 159, 192].

### A.4.3 Superconductors

Superconducting qubits are built using simple oscillator electrical circuits at very low temperatures where they act as superconductors. Qubits are usually manufactured using a superconducting oscillator circuit that admits non-linearity through layers of insulation [34].

128
Superconductivity (frictionless electrical flow) and the introduction of weak links for non-linearity make the circuit behave like an artificial atom, where the energy-level transitions between the ground state and excited state, corresponding to $|0\rangle$ and $|1\rangle$, can be controlled using electronic signals [119, 128, 130]. This technology has a good prospect of scaling up to a large number of qubits [51]. The operating speed of gates is in the high range of quantum technology, currently on the order of 10-100 MHz [140, 187]. On the other hand, this technology is currently less reliable than trapped ions, both in stability of qubits and in gate error rates. The qubits in this technology are stationary, and can only interact with adjacent qubits. A critical advantage of this technology is its low electrical resistance which helps lower the leakage of quantum information, increasing reliability. However, reliability is often hurt by the macroscopic nature of qubits, which contributes to decoherence. Therefore, methods of noise control and isolation from the environment are critical. Microwave pulses inducing Rabbi oscillations are delivered to on-chip qubits to create quantum operations [18, 157].

### A.4.4 Quantum Dots

Another fundamental quantum unit which could serve as a qubit is electric charge. It is possible to create, control, and measure charges, even at the level of single electrons. Quantum dots [117] are tiny electric charges, such as a single electron confined in regions of semiconductor. Such an electron would occupy discrete energy levels, and can be precisely controlled on the surface of the semiconductor wafer using electrostatic pulses. To build a qubit, information is stored in the relative coupling of the electrons, reducing coupling to the environment (for example due to the uniformity of magnetic field effects across electrons). The qubit state is defined in relation to the spin of the electrons: $|0\rangle = \frac{(|\uparrow\downarrow\rangle - |\downarrow\uparrow\rangle)}{\sqrt{2}}$ and $|1\rangle = \frac{(|\uparrow\downarrow\rangle + |\downarrow\uparrow\rangle)}{\sqrt{2}}$. They allow for fast operations [167] and long coherence times approaching milliseconds [26]. However, the fidelity of operations is not as high as the previous two schemes. Quantum dots do not have good long range motion, confining them to nearest-neighbor interactions. This has an adverse effect on scalability, since it may be hard to consolidate control circuitry with the requirements of a dense nearest-neighbor architecture. Operations affecting the states is carried out using microwave magnetic and electric fields. Single qubit operations, as well as preparation and measurement of states are well-established with very fast rates, but scaling to very large numbers of qubits remains a challenge [26, 131, 179].

### A.5 Quantum Error Correction

In Sections A.3 and A.4 we discussed high-level quantum algorithms and low-level physical implementations. This section discusses the intermediary stage necessary to connect these two levels so that interesting quantum algorithms can be run on real machines.

129
A.5.1 Disparities in Error Rate between Algorithmic and Technological Levels

Quantum information is highly fragile, and errors can be induced from various sources outside (e.g. leakage, decoherence) and inside (e.g. faulty gates) the system. Assuming a certain threshold of acceptability for the probability by which results are obtained, different quantum algorithms will impose different minimum acceptable error rates for their constituting operations — that is, the probability that a certain operation will not do as it is supposed to and introduces an error. For example, considering a 50% or higher correctness probability to be acceptable, an algorithm that is made up of a total of $10^9$ operations will require each operation to execute with at most an error rate of $\frac{0.5}{10^9} = 5 \times 10^{-10}$. This is to prevent overall errors from accumulating to more than 50%.

Unfortunately, there is often a very large mismatch between the reliability requirements at the algorithmic level and the reliability guarantees provided at the physical level. For example, it is difficult to imagine a technology whose gates perform better than an error rate of $10^{-3} - 10^{-6}$. That is four to seven orders of magnitude more faulty than the particular algorithmic requirement we mentioned above.

This necessitates an intermediary quantum error correction (QEC) stage to increase the reliability of each individual operation. Anything before the error correction phase is called the physical level and anything after is called the logical level.

A.5.2 General Approach to Error Detection and Correction

Sources of errors are many: An imperfect quantum operation can take the qubit to unexpected states, interaction between data qubits can spread errors from one corrupted qubit to another, and simple passage of time can cause a quantum state to decohere, causing it to become entangled with the state of the environment into a joint, inseparable state.

As with any form of error resilience, redundancy of information must be used to allow detection and correction of errors. We cannot simply copy the state of one qubit into multiple qubits and later take a majority vote, due to the no-cloning theorem which prohibits exact copying of qubit states [193]. However, we can create a large code space where valid code words are separated from each other by a distance, such that any small corruption of data would place the state into an orthogonal error subspace, which can be immediately detected and corrected.

One complicating factor in checking logical qubits for errors is that we cannot directly observe their comprising block of qubits — observing a quantum state collapses the superposition of the state, yielding a simple classical bit. We therefore use ancilla (i.e. helper) qubits for syndrome extraction — learning just enough about the encoded block to understand what kind of error might have occurred on it, without actually learning the full state of the block. The error detection and correction procedure can be repeated in consecutive intervals in the circuit, preventing the spread of errors.

A general error on a qubit can be written as a linear combination of 4 discrete error scenarios: no error, a bit-flip error, a phase-flip error, or both. The key idea behind QEC is to divide the large Hilbert state space into a valid code subspace and an orthogonal error subspace. Syndrome extractions then have the effect of projecting the state onto the basis
vectors of the error subspace. This in effect quantizes errors, such that they can be corrected by applying either no gate, a corrective X gate, a corrective Z gate, or both an X and a Z gate.

A.5.3 Dominance of Error Correction Cost

Due to the large reliability gap between the physical and logical realms, the added redundancy and fault-tolerant overheads to implementing each operation often end up dominating all other resource costs. This means that today, the largest obstacle to creating large-scale quantum computers is the overwhelming effect of errors and the high cost of correcting them. In some estimates, quantum error correction is predicted to utilize up to 90% of qubits in the system [79, 84].

Although error correction has proven to be possible in principle, its actual implementation is very resource consuming. On the theoretical side, a result known as the threshold theorem states that arbitrarily long logical-level computation may be carried out if physical-level error rates are below a certain threshold [8]. This means that error correction circuitry will only impose a polynomial overhead on computation cost.

The constant factors of this polynomial expansion are often very prohibitive, however. The reason is that quantum error correction is very complex. In an effective implementation, data must be encoded at all times, and methods must be devised to compute directly on encoded data. We must be extremely careful about the types of operations that we apply to encoded data, to minimize the risk of error induction beyond repair. In fact, even the process of error recovery itself must isolate errors and prevent contamination within the code block under recovery. Following these guidelines is known as a fault-tolerant protocol. A correct fault-tolerant implementation of a single logical gate might entail thousands to millions of operations [102].

For these reasons, the choice of error correction, computer architecture, and scheduling algorithms can make the difference between feasible and unfeasible computation. Quantifying these tradeoffs is the subject of other chapters in this thesis.

A.6 Concatenated vs. Surface Codes

Section A.5 discussed the importance of quantum error correction and the general ideas behind it. Here, we review two important and established families of quantum error correction — concatenated codes and surface codes. Optimizations and performance comparisons of these two families is a main contribution of this thesis.

A.6.1 Concatenated Codes

Concatenated codes derive their name from the ability to encode one logical qubit in $n$ underlying qubits, then treat each of those $n$ qubits as the input to another stage of encoding, thereby encoding the original logical qubit in $n^2$ physical qubits. Repeating this process yields higher-fidelity qubits, at the cost of extra qubit usage (area) and extra time to perform operations on them (Figure A.8). Common examples of error correcting codes that are
Amenable to this scheme are the 9-qubit Shor [166] and 7-qubit Steane [170] codes. In this thesis we focus on the Steane code as a proof of the viability of our proposals. Lower-overhead concatenated codes do exist [108], and our methods should be extensible to them.

Figure A.8: Concatenated error correcting codes. One logical qubit (gray) is represented by \( n \) physical qubits. To achieve better error suppression, more another level of concatenation can be used for \( n^2 \) qubits. Here, we show the special case of Steane code, where \( n = 7 \).

The Steane code is a generalization of the classical Hamming code, where 7 physical qubits are used to encode 1 logical qubit (hence, the next concatenation level would use 49 qubits). The encoded \( |0\rangle (|1\rangle) \) state is the superposition of all 8 valid Hamming code words with even (odd) parity.

\[
|0\rangle_{\text{code}} = \frac{1}{\sqrt{8}} (|0000000\rangle + |0001111\rangle + |0110011\rangle + |0111100\rangle + |1010101\rangle + |1011010\rangle + |1100110\rangle + |1101001\rangle) \tag{A.3}
\]

Steane codes are weight-3 codes, meaning valid codewords are separated from each other by a distance of 3 bit-flips. Therefore this code can correct a bit-flip, a phase-flip, or due to linearity of quantum mechanics a combination of these, but only on one single qubit per each block.

Steane QEC has the important advantage that most quantum operations on encoded data can be done transversally, that is applying the same operation to the \( i \)th physical qubit of every block has the effect of that operation being applied to the whole logical qubit. Transversal gates are automatically fault-tolerant because they keep the error confined to the \( i \)th qubit of every block. Unfortunately no QEC allows a universal set of all transversal gates [54, 198] (except with high-overhead code-switching techniques [104]), and the employment of a non-transversal gate is unavoidable. The \( T \) gate is our only non-transversal gate.

In order to detect and correct errors, the Steane code must extract bit-flip and phase-flip syndromes into ancillary qubits, without disturbing the quantum state of the data. The information obtained from the two syndromes can then be used to decide whether any correction steps are necessary. Circuits for syndrome extraction are depicted in Figure A.9. The ancilla used to transfer the syndrome information must first be prepared in the \( |0\rangle_{\text{Steane}} \) state. A circuit to do this is shown in Figure A.10.

The state fidelity of these zero ancilla qubits is an essential requirement if we are to do good error correction, and do it sparingly [190]. Since the circuit for an encoded zero state preparation is not transversal, it is not automatically fault-tolerant – a single error during any of the operations of this circuit might cause errors in 2-3 physical qubits in the final
Figure A.9: Syndrome extraction in the Steane code, assuming level-1 concatenation with 7 physical qubits per logical qubit, and ancilla prepared in the $\ket{0}_{\text{Steane}}$ state. A single bit or phase flip in the data block will feed into the relevant ancilla block, corrupting the Hamming parity check after measurement.

Figure A.10: Circuit to prepare an encoded zero state in the Steane code space. The series of operations follow from Equation A.3. In this level-1 encoding, for every logical (encoded) qubit, 7 physical (unencoded) qubits are used.

state. Therefore an extra verification step is needed to make sure the correct encoded state was prepared. A simple verification circuitry is shown in Figure A.11, which prepares 3 encoded zero states and does majority voting to determine if the first one is in the correct state. The procedure is repeated until the encoded zero state is successfully verified.

Putting all of this together, the final circuit for error detection and correction is shown in Figure A.12. This subcircuit would be injected at regular intervals during the quantum program’s execution, in order to keep state disturbances in check, and the overall logical error rate low. Since error recovery itself is error-prone, we repeat the entire recovery circuit twice in order to lower the probability that a faulty syndrome extraction affects our result [148]. Every invocation of this subcircuit requires four fresh, verified qubits in the encoded $\ket{0}_{\text{Steane}}$ state, or twelve non-verified qubits of the same kind. This translates to 84 physical qubits for level-1 and 588 physical qubits for level-2 error correction.
Figure A.11: Verification of the $|0\rangle_{\text{Steane}}$ ancilla state. Since the “Prep $|0\rangle_{\text{Steane}}$” circuitry is non-transversal and hence prone to introducing more errors than the code can tolerate, the prepared state must be verified, against two more states of the same kind. Interaction with and measurement of the two ancillary blocks can verify or reject the original prepared $|0\rangle_{\text{Steane}}$ state. All of this is done continuously in dedicated SIMD regions for an ongoing supply of pure $|0\rangle_{\text{Steane}}$ states.

Figure A.12: Error correction circuit showing the consumption of four verified encoded zero states. These zero states will be used to correct possible bit flips and phase flips on the data qubit. Each syndrome is extracted twice, to catch possible errors in the syndrome extraction itself. All possible errors on the state can be modeled as a combination of bit and phase flips.

### A.6.2 Magic States for T gates

Unfortunately transversal gates in the Steane code are not enough for universal computation. In fact Gottesman shows that a circuit of only transversal operations can in principle be efficiently simulated on a classical computer [66]. Therefore, we need to include one extra gate, which in this study is the $T$ gate. In order to perform the $T$ gate in a fault-tolerant manner, we need to prepare special ancilla states called “magic states.” Figure A.14 shows
how a magic state (also known as a $|A\rangle$ state) is consumed to apply the $T$ gate to the encoded data block. Magic states have two useful properties: They can be prepared and purified using only transversal Clifford gates (Figure A.13), and once they are created, the set of transversal Clifford gates and magic states allow for Universal Quantum Computation. They are called magic because the same state has both properties [32]. The precise state of these magic state ancillas is: $|A\rangle = (1/\sqrt{2})(|0\rangle + e^{i\pi/4}|1\rangle)$.

Figure A.13: Preparing an encoded magic $|A\rangle$ state, using only Clifford gates. All gates are applied transversally to single qubits. Prep $|\text{Cat}\rangle_{\text{Steane}}$ is a simple circuit comprised of one Hadamard and 6 CNOT gates (for level-1), which places all qubits in an equal superposition state.

All in all, in order to apply one fault-tolerant $T$ gate, 14 extra ancillas in level-1 and 98 in level-2 are consumed. Additionally, 7 (or 49) extra teleportation operations are required to bring the prepared magic $|A\rangle$ state near the data block and apply the $T$ gate.

A.6.3 Surface Codes

Surface codes are a family of promising, low-overhead QEC codes [60], where logical qubit information is encoded in the topology of a two-dimensional lattice of physical qubits [151]. A surface code lattice alternates data and ancilla qubits (shown respectively as white and black circles in Figure A.15). Data qubits collectively encode the logical state, while ancillas continuously collect syndrome measurements from their surrounding data. It can be shown that this layout creates two degrees of freedom, so the lattice as a whole acts as an encoded qubit.
Figure A.15: One encoded logical qubit, using the (a) planar and (b) double-defect surface code variations. Ancilla qubits (small black) are used to continuously monitor data qubits (large white), with alternate bit-flip (red) and phase-flip (green) syndromes measured. The planar encoding uses fewer number of physical qubits to encode the same logical code distance (here $d = 4$). The planar qubit is a stand-alone logical qubit with defined borders, while the double-defect is part of a larger lattice, comprising other logical qubits.

In this thesis we investigate planar [33,48] and double-defect [60] encodings as the two main flavors of the surface code.

In the planar encoding (Figure A.15a), a single lattice (or “plane”) is constrained to represent a single logical qubit. The logical qubit can be manipulated by operating on a chain of physical qubits connecting opposite edges of the lattice. Multiple qubits are introduced to the system by building separate lattices, discontinuous from each other, to represent each new logical qubit. In this scenario, the system can be seen as a collection of planar sheets.

The double-defect implementation introduces holes, or “defects” into the lattice, where the measurement operations performed by the measurement qubits are temporarily suspended. By halting the operation of a single measurement qubit, two additional degrees of freedom are introduced that are not tied to the boundaries. This allows the operators to be localized to a string of qubits connecting the defects and surrounding one of them. By defining logical operators in this fashion, a lattice of physical qubits can accommodate more than one logical qubit. Double-defect implementations, consist of one monolithic lattice, where different regions correspond to different logical qubits.

Similar to concatenated codes, the surface codes requires a purified magic state to implement the T gate, which must be prepared and disseminated from a specialized region of the lattice. We call this region a magic state factory.

A full round of syndrome measurement requires each ancilla to interact with its neighboring data to the north, east, south and west, and then be measured. Although these interactions occur serially among themselves, each can be parallelized across all ancillas, such that one surface code cycle is approximately equivalent to the latency of four 2-qubit interactions (plus final measurements). Part of the efficiency of surface codes is due to this
high qubit utilization — almost all qubits are active at every cycle of computation. This allows denser packing of information.

As we mentioned, it suffices to detect bit-flip and phase-flip errors in order to correct arbitrary qubit errors. Therefore, ancillas are alternately designated as those that measure bit-flip syndromes and those that measure phase-flip syndromes (shown respectively as red and green interactions in Figure A.15). By recording a history of syndrome measurements at every surface code cycle, we are able to detect anomalous syndromes. Since there is spatial overlap between syndromes collected from multiple data, a minimum weigh-matching algorithm must be employed to trace the cause of anomaly to one or more data qubits on the lattice. Aggregated syndrome measurements in time can be likened to a 3D space-time volume; the bottom area being the lattice, and the vertical height the progress in time (Figure A.16).

An actual correction need not be applied. Due to the commutative properties of bit-flip and phase-flip operations with other program operations, corrections can be postponed until the end of program. The software only needs to keep track of them. In fact, often they will be cancelled by another correction of the same kind downstream.

Extending the lattice dimensions increases reliability (reduces error rate). The measure of coding strength is called code distance ($d$), corresponding to the dimensions of the lattice. Error recovery will only fail when an undetectable error, mimicking a legitimate logical operation occurs, that is a chain of errors resembling a logical bit-flip or phase-flip operation. In the planar code, this is a chain that connects two opposite ends of the lattice. In the double-defect code, this is a chain that surrounds one of the holes, or connects the two holes. In Figure A.15, $d = 4$. It is clear that the planar encoding is more efficient in terms of qubit usage. The disadvantage, however, is that it does not lend itself well to nearest-neighbor operations, and thus methods such as teleportation or lattice surgery [76] must be employed.

### A.6.4 Handling Data Movement

As we saw, during the execution of a series of quantum operations, many movements are required: Purified $|0\rangle$ states must be moved around to perform Steane error correction; Purified magic ($|A\rangle$) states need to be moved around in order to implement T gates in both concatenated and surface code; and interaction between logical data qubits are often needed to simply perform an algorithm, where the qubits are not adjacent.

In the quantum regime, there are several means of transporting:

**Shuttling**

In some technologies qubits are mobile — they can be physically transported where they are needed. This mode of transportation is called shuttling or ballistic movement. In trapped ions, for example, the voltage of electrodes that create a potential well for trapping ions can be manipulated by precise pulses. This can be done such that the qubit moves between different electrodes, thereby physically moving from one place to another. This mode of transportation is distance-sensitive, and the delay associated with it is directly proportional...
Figure A.16: (a) Error detection in the surface code via syndrome measurements. Bit-flip and phase-flip syndromes (alternate red and green) are measured at each time slice (a surface code cycle), keeping a history in time and space in software. Maximum likelihood calculations over this history can determine sources of measurement anomalies, determining faulty qubits. (b) Defects that are stretched through space create “braids,” which may be used to operate on the encoded logical qubit. Here, an initialization of the qubit to the logical $|0\rangle$ state occurs.

to the traveled distance, and the number of turns on the path. It does not use extra qubits, but channels are needed for qubits to move.

**Swapping**

Contrary to the above setup, in some technologies qubits are not mobile. For example, in trapped ions, qubits are embedded in a chip and cooled to extremely low temperatures, and thus are very difficult to move. Any long-distance communication must be accomplished through a combination of nearest-neighbor operations. Swaps are special quantum operations that consist of a sequence of 3 CNOTs between two adjacent qubits — the effect is that the state of two qubits will be swapped. Several consecutive swaps in chain, therefore, has the effect of achieving a long-distance movement. Clearly this is an expensive operation. Not only is the time to travel distance sensitive, but also the channels that are used for movement also consume extra “dummy” qubits, whose existence is only for the purpose of facilitating swaps.
**Teleportation**

Teleportation is a technique in quantum mechanics that makes almost-instant transmission of exact qubit states possible. It works by entanglement of states, which causes operations on one qubit to affect the state of the other entangled qubit, even if the two are far apart from each other. When the source and destination qubits of the communication are both entangled to an intermediary qubit, they are “linked,” even when existing over long distances. Therefore, the information of one can be transmitted to the other one quickly.

Teleportation is not precisely instant: first, the two source and destination qubits must first be transmitted physically to their long-distance locations. This is achieved either through shuttling or swapping, and is known as *EPR distribution*. Second, the transmission of the actual qubit state is done after the transmission of two classical bits over the long distance, which introduces its own delays. However, the advantage of teleportation is that the hard part of communication (physical shuttling or swapping) is decoupled from the point in the program where the data is needed — this data can be transmitted using merely two classical bits, causing almost no communication congestion. Teleportation is also *distance-insensitive*: the communication delay is just the number of cycles it takes to perform the entangling operations, regardless of the distance between source and destination.

**Braiding**

While the above methods could be performed on both concatenated code blocks and surface codes planes, braiding is a special surface code operation. This operation consists of extending defect through space, to encompass different logical qubits and to connect them. This operation can create the effect of logical CNOT gates between braided qubits. This CNOT can then be used to either interact long-distance qubits, or to interact a qubit with a prepared magic state in order to perform T gates. Figure A.16b shows how these extended braids create “pipes” in the 3D space-time volume. The advantage of braids is that their extension and shrinkage only takes on cycle, regardless of the distance. Their disadvantage, however, is that they cannot cross, and thus the packing of these pipes in the 3D volume must be optimized — a process which is NP-hard [75].
Appendix B

Scaffold Grammar

\[
\langle \text{translation-unit} \rangle ::= \langle \text{external-declaration} \rangle^* \\
\langle \text{external-declaration} \rangle ::= \langle \text{function-definition} \rangle \mid \langle \text{declaration} \rangle \\
\langle \text{function-definition} \rangle ::= \langle \text{declaration-specifier} \rangle^* \\
\langle \text{direct-declarator} \rangle \\
\langle \text{declaration} \rangle^* \\
\langle \text{compound-statement} \rangle \\
\langle \text{declaration-specifier} \rangle ::= \langle \text{storage-class-specifier} \rangle \\
\langle \text{type-specifier} \rangle \\
\langle \text{type-qualifier} \rangle \\
\langle \text{storage-class-specifier} \rangle ::= \text{extern} \mid \text{typedef} \\
\langle \text{type-specifier} \rangle ::= \text{qbit} \mid \text{cbit} \mid \text{char} \mid \text{int} \mid \text{long} \mid \text{float} \mid \text{double} \mid \text{signed} \mid \text{unsigned} \mid \\
\langle \text{struct-or-union-specifier} \rangle \\
\langle \text{enum-specifier} \rangle \\
\langle \text{typedef-name} \rangle \\
\langle \text{gate} \rangle ::= \text{H} \mid \text{X} \mid \text{Y} \mid \text{Z} \mid \text{S} \mid \text{S} _\text{dag} \mid \text{T} \mid \text{T} _\text{dag} \mid \text{PrepZ} \mid \text{MeasZ} \mid \text{CNOT} \mid \text{Toffoli} \mid \text{Rz} \\
\langle \text{struct-or-union-specifier} \rangle ::= \langle \text{struct-or-union} \rangle \langle \text{identifier} \rangle \langle \text{struct-declaration} \rangle^+ \\
\langle \text{struct-or-union} \rangle ::= \langle \text{struct-or-union} \rangle \langle \text{struct-declaration} \rangle^+ \\
\langle \text{struct-or-union} \rangle ::= \langle \text{struct-or-union} \rangle \langle \text{identifier} \rangle \\
\langle \text{struct-or-union} \rangle ::= \text{struct} \\
\langle \text{struct-or-union} \rangle ::= \text{union} \\
\langle \text{struct-declaration} \rangle ::= \langle \text{specifier-qualifier} \rangle^* \langle \text{struct-declarator-list} \rangle \\
\langle \text{specifier-qualifier} \rangle ::= \langle \text{type-specifier} \rangle \mid \langle \text{type-qualifier} \rangle \\
\langle \text{struct-declarator-list} \rangle ::= \langle \text{struct-declarator} \rangle \\
\langle \text{struct-declarator-list} \rangle ::= \langle \text{struct-declarator} \rangle \langle \text{struct-declarator} \rangle^* \\
\langle \text{struct-declarator} \rangle \\
\langle \text{struct-declarator} \rangle ::= \langle \text{structural-declarator} \rangle \\
\langle \text{structural-declarator} \rangle::= \langle \text{specifier-qualifier} \rangle^* \langle \text{struct-declarator-list} \rangle \\
\langle \text{specifier-qualifier} \rangle ::= \langle \text{type-specifier} \rangle \mid \langle \text{type-qualifier} \rangle
\(\langle\text{struct-declarator}\rangle ::= \langle\text{direct-declarator}\rangle \\
\quad | \langle\text{direct-declarator}\rangle : \langle\text{constant-expression}\rangle \\
\quad | : \langle\text{constant-expression}\rangle\)

\(\langle\text{type-qualifier}\rangle ::= \text{const} | \text{volatile}\)

\(\langle\text{direct-declarator}\rangle ::= \langle\text{identifier}\rangle \\
\quad | (\langle\text{direct-declarator}\rangle) \\
\quad | \langle\text{direct-declarator}\rangle [ \langle\text{constant-expression}\rangle? ] \\
\quad | \langle\text{direct-declarator}\rangle (\langle\text{parameter-type-list}\rangle) \\
\quad | \langle\text{direct-declarator}\rangle (\langle\text{identifier}\rangle*)\)

\(\langle\text{constant-expression}\rangle ::= \langle\text{conditional-expression}\rangle\)

\(\langle\text{conditional-expression}\rangle ::= \langle\text{logical-or-expression}\rangle \\
\quad | \langle\text{logical-or-expression}\rangle ? (\langle\text{expression}\rangle) : \langle\text{conditional-expression}\rangle\)

\(\langle\text{logical-or-expression}\rangle ::= \langle\text{logical-and-expression}\rangle \\
\quad | \langle\text{logical-or-expression}\rangle | | \langle\text{logical-and-expression}\rangle\)

\(\langle\text{logical-and-expression}\rangle ::= \langle\text{inclusive-or-expression}\rangle \\
\quad | \langle\text{logical-and-expression}\rangle & \& \langle\text{inclusive-or-expression}\rangle\)

\(\langle\text{inclusive-or-expression}\rangle ::= \langle\text{exclusive-or-expression}\rangle \\
\quad | \langle\text{inclusive-or-expression}\rangle | \langle\text{exclusive-or-expression}\rangle\)

\(\langle\text{exclusive-or-expression}\rangle ::= \langle\text{and-expression}\rangle \\
\quad | \langle\text{exclusive-or-expression}\rangle \hat{\&} \langle\text{and-expression}\rangle\)

\(\langle\text{and-expression}\rangle ::= \langle\text{equality-expression}\rangle \\
\quad | \langle\text{and-expression}\rangle & \langle\text{equality-expression}\rangle\)

\(\langle\text{equality-expression}\rangle ::= \langle\text{relational-expression}\rangle \\
\quad | \langle\text{equality-expression}\rangle == \langle\text{relational-expression}\rangle \\
\quad | \langle\text{equality-expression}\rangle != \langle\text{relational-expression}\rangle\)

\(\langle\text{relational-expression}\rangle ::= \langle\text{shift-expression}\rangle \\
\quad | \langle\text{relational-expression}\rangle < \langle\text{shift-expression}\rangle \\
\quad | \langle\text{relational-expression}\rangle > \langle\text{shift-expression}\rangle \\
\quad | \langle\text{relational-expression}\rangle <= \langle\text{shift-expression}\rangle \\
\quad | \langle\text{relational-expression}\rangle >= \langle\text{shift-expression}\rangle\)

\(\langle\text{shift-expression}\rangle ::= \langle\text{additive-expression}\rangle \\
\quad | \langle\text{shift-expression}\rangle << \langle\text{additive-expression}\rangle \\
\quad | \langle\text{shift-expression}\rangle >> \langle\text{additive-expression}\rangle\)

141
(additive-expression) ::= (multiplicative-expression)
   | (additive-expression) + (multiplicative-expression)
   | (additive-expression) - (multiplicative-expression)

(multiplicative-expression) ::= (cast-expression)
   | (multiplicative-expression) * (cast-expression)
   | (multiplicative-expression) / (cast-expression)
   | (multiplicative-expression) % (cast-expression)

(cast-expression) ::= (unary-expression)
   | (type-name) (cast-expression)

(unary-expression) ::= (postfix-expression)
   | ++ (unary-expression)
   | - (unary-expression)
   | (unary-operator) (cast-expression)
   | sizeof (unary-expression)
   | sizeof (type-name)

(postfix-expression) ::= (primary-expression)
   | (postfix-expression) [ (expression) ]
   | (postfix-expression) ( (assignment-expression)* )
   | (postfix-expression) . (identifier)
   | (postfix-expression) -> (identifier)
   | (postfix-expression) ++
   | (postfix-expression) --

(primary-expression) ::= (identifier)
   | (constant)
   | (gate)
   | (string)
   | (expression)

(constant) ::= (integer-constant)
   | (character-constant)
   | (floating-constant)
   | (enumeration-constant)

(expression) ::= (assignment-expression)
   | (expression) , (assignment-expression)

(assignment-expression) ::= (conditional-expression)
   | (unary-expression) (assignment-operator) (assignment-expression)

(assignment-operator) ::= = | *= | /= | %= | += | -= | &= | ^= | |=
\[\text{unary-operator} ::= \& | \ast | + | - | !\]
\[\text{type-name} ::= \text{specifier-qualifier} +\]
\[\text{parameter-type-list} ::= \text{parameter-list} | \text{parameter-list} , ...\]
\[\text{parameter-list} ::= \text{parameter-declaration} | \text{parameter-list} , \text{parameter-declaration}\]
\[\text{parameter-declaration} ::= \text{declaration-specifier} + \text{direct-declarator} | \text{declaration-specifier} +\]
\[\text{enum-specifier} ::= \text{enum} \text{identifier} \text{enumerator-list} | \text{enum} \text{enumerator-list} | \text{enum} \text{identifier}\]
\[\text{enumerator-list} ::= \text{enumerator} | \text{enumerator-list} , \text{enumerator}\]
\[\text{enumerator} ::= \text{identifier} | \text{identifier} = \text{constant-expression}\]
\[\text{typedef-name} ::= \text{identifier}\]
\[\text{declaration} ::= \text{declaration-specifier} + \text{init-declarator}^*\]
\[\text{init-declarator} ::= \text{direct-declarator} | \text{direct-declarator} = \text{initializer}\]
\[\text{initializer} ::= \text{assignment-expression} | \text{initializer-list} , \text{initializer-list}\]
\[\text{initializer-list} ::= \text{initializer} | \text{initializer-list} , \text{initializer}\]
\[\text{compound-statement} ::= \text{declaration}^* \text{statement}^*\]
\[\text{statement} ::= \text{labeled-statement} | \text{expression-statement} | \text{compound-statement} | \text{selection-statement} | \text{iteration-statement} | \text{jump-statement}\]
\[\text{labeled-statement} ::= \text{case} \text{constant-expression} : \text{statement} | \text{default} : \text{statement}\]
⟨expression-statement⟩ ::= ⟨expression⟩? ;

⟨selection-statement⟩ ::= if ( ⟨expression⟩ ) ⟨statement⟩
| if ( ⟨expression⟩ ) ⟨statement⟩ else ⟨statement⟩
| switch ( ⟨expression⟩ ) ⟨statement⟩

⟨iteration-statement⟩ ::= while ( ⟨expression⟩ ) ⟨statement⟩
| do ⟨statement⟩ while ( ⟨expression⟩ ) ;
| for ( ⟨expression⟩? ; ⟨expression⟩? ; ⟨expression⟩? ) ⟨statement⟩

⟨jump-statement⟩ ::= continue ;
| break ;
| return ⟨expression⟩? ;
Appendix C

QASM Grammars

C.1 QASM-H

\(<\text{Start}\> ::= (\langle\text{Module}\rangle\langle\text{newline}\rangle)^* \langle\text{Main}\rangle\)

\(<\text{Main}\> ::= \text{‘main’} (\langle\text{Variable}\rangle\langle\text{newline}\rangle)^+ \langle\text{ModuleBody}\rangle\)

\(<\text{Module}\> ::= \text{‘module’} \langle\text{name}\rangle \langle\text{ParamList}\rangle \langle\text{ModuleBody}\rangle\)

\(<\text{ParamList}\> ::= \langle\text{Variable}\rangle (’,’ \langle\text{Variable}\rangle)^*\)

\(<\text{ModuleBody}\> ::= (\langle\text{Gate}\rangle\langle\text{newline}\rangle)^+\)

\(<\text{Variable}\> ::= \text{‘qubit’} | \text{‘cbit’} | (\text{‘[‘}\langle\text{name}\rangle\langle\text{‘]’}\text{‘:’}\langle\text{name}\rangle\text{‘]’})\?) \langle\text{name}\rangle\)

\(<\text{Gate}\> ::= (\langle\text{OneQubitGate}\rangle | \langle\text{MultiQubitGate}\rangle | \langle\text{Repeat}\rangle | \langle\text{ModuleCall}\rangle) \langle\text{newline}\rangle\)

\(<\text{OneQubitGate}\> ::= (\text{‘H’} | \text{‘X’} | \text{‘Y’} | \text{‘Z’} | \text{‘S’} | \text{‘S_dag’} | \text{‘T’} | \text{‘T_dag’} | \text{‘Prep0’} | \text{‘Meas0’}) \langle\text{Target}\rangle\)

\(<\text{MultiQubitGate}\> ::= (\text{‘CNOT’} \langle\text{Ctrl}\rangle’,’ \langle\text{Target}\rangle) \]
  | (\text{‘Toffoli’} \langle\text{CtrlArray}\rangle’,’ \langle\text{Target}\rangle)
  | (\text{‘Fredkin’} (\langle\text{CtrlArray}\rangle’,’)’) \langle\text{Target}\rangle’,’ \langle\text{Target}\rangle)

\(<\text{Repeat}\> ::= \text{‘repeat’} \langle\text{num}\rangle \langle\text{Gate}\rangle\)

\(<\text{ModuleCall}\> ::= \langle\text{name}\rangle(\text{CallParamList})\)

\(<\text{CallParamList}\> ::= \langle\text{Param}\rangle (’,’ \langle\text{Param}\rangle)^*\)

\(<\text{Param}\> ::= \langle\text{name}\rangle | \langle\text{name}\rangle \langle\text{‘[‘}\langle\text{name}\rangle\langle\text{‘]’}\text{‘:’}\langle\text{name}\rangle\text{‘]’})\)

\(<\text{Target}\> ::= \langle\text{name}\rangle \langle\text{name}\rangle \langle\text{‘[‘}\langle\text{name}\rangle\langle\text{‘]’}\text{‘:’}\langle\text{name}\rangle\text{‘]’})\)

\(<\text{Ctrl}\> ::= \langle\text{name}\rangle \langle\text{name}\rangle \langle\text{‘[‘}\langle\text{name}\rangle\langle\text{‘]’}\text{‘:’}\langle\text{name}\rangle\text{‘]’})\)
\langle \text{CtrlArray} \rangle ::= \langle \text{Ctrl} \rangle (' , '.' \langle \text{Ctrl} \rangle )* \\
\langle \text{name} \rangle ::= (a-zA-Z)+([a-zA-Z][0-9])*
\langle \text{num} \rangle ::= [1-9]([0-9])*
\langle \text{newline} \rangle ::= \n
\langle \text{whitespace} \rangle ::= \n | \r | \t
\langle \text{comment} \rangle ::= #.*(\n | \EOF)

C.2 QASM-HF

\langle \text{Start} \rangle ::= (\langle \text{Module} \rangle \langle \text{newline} \rangle )* \langle \text{Main} \rangle
\langle \text{Main} \rangle ::= \text{‘main’} (\langle \text{Variable} \rangle \langle \text{newline} \rangle )* \langle \text{ModuleBody} \rangle
\langle \text{Module} \rangle ::= \text{‘module’} \langle \text{name} \rangle \langle \text{ParamList} \rangle \langle \text{ModuleBody} \rangle
\langle \text{ParamList} \rangle ::= \langle \text{Variable} \rangle (' , '.' \langle \text{Variable} \rangle )*\n\langle \text{ModuleBody} \rangle ::= (\langle \text{Gate} \rangle \langle \text{newline} \rangle )*\n\langle \text{Variable} \rangle ::= \text{‘qubit’} | \text{‘cbit’} | \text{‘[’} \langle \text{num} \rangle \text{‘:]’} | \text{‘[’} \langle \text{num} \rangle \text{‘:]’} | \text{‘[’} \langle \text{num} \rangle \text{‘:]’} | \langle \text{name} \rangle
\langle \text{Gate} \rangle ::= (\langle \text{OneQubitFTGate} \rangle | \langle \text{TwoQubitFTGate} \rangle | \langle \text{Repeat} \rangle | \langle \text{ModuleCall} \rangle ) \langle \text{newline} \rangle
\langle \text{OneQubitFTGate} \rangle ::= \text{‘H’} | \text{‘X’} | \text{‘Y’} | \text{‘Z’} | \text{‘S’} | \text{‘S\_dag’} | \text{‘T’} | \text{‘T\_dag’} | \text{‘Prep0’} | \text{‘Meas0’}
\langle \text{Target} \rangle
\langle \text{TwoQubitFTGate} \rangle ::= \text{‘CNOT’} \langle \text{Ctrl} \rangle | \text{‘, ‘} \langle \text{Target} \rangle
\langle \text{Repeat} \rangle ::= \text{‘repeat’} \langle \text{num} \rangle \langle \text{Gate} \rangle
\langle \text{ModuleCall} \rangle ::= \langle \text{name} \rangle(\text{CallParamList})
\langle \text{CallParamList} \rangle ::= \langle \text{Param} \rangle (' , '.' \langle \text{Param} \rangle )*\n\langle \text{Param} \rangle ::= \langle \text{name} \rangle | \langle \text{name} \rangle \text{‘[’} \langle \text{num} \rangle \text{‘:]’} | \langle \text{name} \rangle \text{‘[’} \langle \text{num} \rangle \text{‘:]’} | \langle \text{name} \rangle \text{‘[’} \langle \text{num} \rangle \text{‘:]’} |
\langle \text{Target} \rangle ::= \langle \text{name} \rangle | \langle \text{name} \rangle \text{‘[’} \langle \text{num} \rangle \text{‘:]’} |
\langle \text{Ctrl} \rangle ::= \langle \text{name} \rangle | \langle \text{name} \rangle \text{‘[’} \langle \text{num} \rangle \text{‘:]’} |
\langle \text{name} \rangle ::= (a-zA-Z)+([a-zA-Z][0-9])*
\langle \text{num} \rangle ::= [1-9]([0-9])*

146
\(\langle \text{newline} \rangle ::= \n\)
\(\langle \text{whitespace} \rangle ::= \n | \r | \t\)
\(\langle \text{comment} \rangle ::= \#.*(\n | \text{EOF})\)
Bibliography


158


