Practical Fault Tolerance for Quantum Circuits

Mark Whitney

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-80
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-80.html

May 21, 2009

Practical Fault Tolerance for Quantum Circuits

by

Mark Gregory Whitney

B.S. (University of California, Berkeley) 2000
M.S. (University of California, Berkeley) 2006

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division
of the
University of California, Berkeley

Committee in charge:

Professor John D. Kubiatowicz, Chair
Professor Sanjit A. Seshia
Professor K. Birgitta Whaley

Spring 2009

The dissertation of Mark Gregory Whitney is approved:

Chair Date

Date

Date

University of California, Berkeley

Practical Fault Tolerance for Quantum Circuits

Abstract

Practical Fault Tolerance for Quantum Circuits

by

Mark Gregory Whitney
Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor John D. Kubiatowicz, Chair

Due to very high projected error rates, large scale quantum computers will require substantial fault tolerance just to maintain a minimum level of reliability. We present tools to better analyze the performance of large, fault tolerant quantum computer designs. We find that current uses of quantum error correction are overly conservative in mitigating the impact of gate errors and negligent of other error sources in quantum data communication and memory.

We have developed circuit layout heuristics to generate detailed designs in trapped ion quantum computing technology. From these designs, we can extract much more accurate error models for a given application, including all gate, movement and idle errors on qubits. Using these extracted models, our flexible error simulation environment determines the overall failure probability of the design. Included in this simulation environment is a bit-parallel Monte Carlo technique that is 10 times faster than previous fault propagation simulations. This allows us to evaluate the reliability of designs that are an order of magnitude larger, in the same amount of time.

Using this analysis framework to verify reliability, we have developed a linear programming-based optimization for error correction which decreases overall circuit resources by an order of magnitude. In some cases, our optimization actually improves overall system reliability by removing error correction. We combine this optimization with judicious quantum error correcting code selection to provide efficient designs for large quantum arithmetic kernels used in Shor's factorization algorithm. We show our optimized designs perform 2x to 100x better than previous works in terms of probabilistic area-delay product.

Additionally, the area of our layout of a 1024-bit factoring using Shor's algorithm is $64cm^2$, a substantial improvement compared to the $0.9m^2$ state-of-the-art design from prior work. A design size reduction by this amount will make fabricating such an application feasible much sooner.

Professor John D. Kubiatowicz
Dissertation Committee Chair

Dedicated to my wife and parents for their patience.

Contents

List of Figures

List of Tables

Acknowledgments

I clearly could not have finished this work without the intellectual, emotional and financial support of my advisor, John Kubiatowicz, a.k.a. Kubi. His never-ending source of good ideas and encouragement enabled me to overcome all the inevitable frustrations of scientific research.

Secondly, this work on the Quadence quantum CAD flow was a group effort that included Nemanja Isailovic and Yatish Patel. Our joint work on this project, ranging from brainstorming to implementation to paper-writing, made possible all the results in this thesis. Even more important, our group of 3 (the Kubits) have developed a close friendship that will last long past our time in the graduate program. I look forward to working with these guys on other projects in the future.

My thesis committee gets special thanks for making this final stage in my graduate career more fruitful. Thanks to Professor Sanjit Seshia for his insight into the more rigorous formulations of my work, this thesis will have more impact and clarity. I owe Professor Birgitta Whaley thanks for giving me more opportunities to present my research to a wider audience.

Andrew Cross' ftqctools were instrumental in enabling the whole quantum error correction synthesis process which is the starting point to the whole CAD flow. Without his work, this research would have taken much longer since I would have had to develop my only set of error correcting code libraries from scratch. His advice on error correcting codes was also invaluable.

Lucas Kreger-Stickles was also generous with his help in getting the LQLA benchmark up and running, so we would have a valuable comparison to prior work.

I offer my thanks to Yury Markovsky and Victor Wen for being the "outsiders" who would frequently critique our work to help us target it for wider understanding and applicability. The donated many hours to reading our papers, listening to our talks, and providing valuable feedback.

Finally, I would like to thank my parents and my wife for their combination of endless support and nagging that helped me get to the point where I am today.

Chapter 1

Introduction

Quantum computers will be able to solve a number of important physics and mathematical problems with interesting asymptotic improvements [105, 3, 84, 41]. The most cited potential use for quantum computing is to factor large numbers based on *Shor's algorithm* [84]. In order for quantum computers to solve these difficult and interesting problems, they will need to support a larger number of quantum data bits (qubits) and quantum circuit elements. In order to sufficiently scale a quantum computer, there are a number of challenges to overcome, one of these being the control of quantum decoherence of the data under computation.

Previous estimates of the proportion of quantum computing resources that would need to be dedicated to quantum error control is approximately 95%. This substantial amount of overhead has led to proposed quantum computer designs that measure $0.9m^2$ in area [64]. In most cases it is infeasible to fabricate a design with this area. It is almost certainly the case that a design for an algorithm that uses fewer resources (gates, qubits, instructions, etc.) will be able to be built in the lab sooner as technologies improve.

Past research done on quantum error control has focused on the *concatenation* of error correcting codes, which is essentially recursive application of error correction circuits to the base logical circuit [2]. Much attention has been given to analytical and numerical *thresholds* for elementary gate errors [5, 76]. These thresholds give the maximum gate error rate tolerable by a particular concatenated code, in order to still have a working circuit. These thresholds are based on the *threshold theorem* [2] which states that as long as gate error is below the threshold, the failure probability of the circuit decreases asymptotically to zero as more and more error correction is concatenated on top of it. There are a number

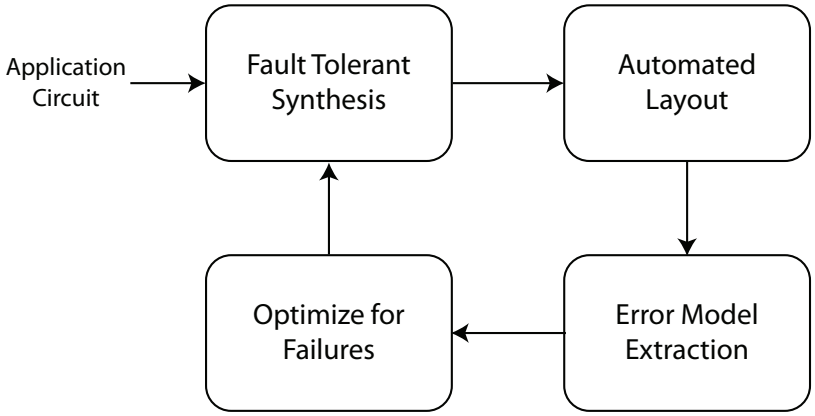


Figure 1.1: Evaluating the affect of communication costs on quantum circuit fault tolerance can be done through an iterative process of fault tolerant synthesis, circuit layout, failure model extraction, and analysis and optimization of the layout and circuit.

of deficiencies with this approach:

- Threshold techniques focus on the reliability of a single logical circuit element. It does not necessarily give a prescription for how to distribute error control procedures throughout a more complex application circuit, like a circuit implementation of Shor's algorithm.
- Most threshold derivations must be pessimistic to account for variation in error correction code properties, or they rely on careful analysis of a single code. Neither of these approaches allow for a fair comparison of the performance of different error correction schemes.
- Communication and memory errors are largely ignored in most threshold evaluations. Work in [37, 97, 96, 56] are the exceptions to this, but these studies based on particular error models for a single choice of error correcting code. Different error models and codes will have a different impact on overall circuit reliability.
- While these thresholding techniques provide a guarantee that a circuit *could* be constructed, they do not provide recipe for how to create a physical representation of the circuit.

While threshold analysis is invaluable for determining overall feasibility of building reliable quantum gates. It will not give much guidance on how to build, optimize, or evaluate

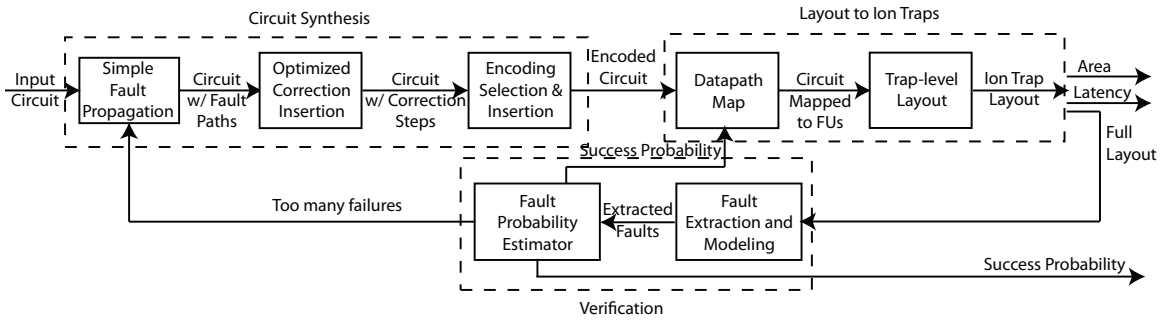


Figure 1.2: Our quantum circuit design flow. It takes as input a quantum circuit and outputs a layout and various metrics on the layout.

a particular quantum circuit. One key result of this work is that *current schemes for fault tolerant quantum error correction waste resources and can make errors more probable in some cases. Through automated fault analysis and circuit optimization we tailor error correction procedures to improve both reliability and resource utilization.*

Even though past work on quantum computing architectures [8, 64, 98] has suggested that the primary activity in any quantum circuit will be error correction. We show in Chapter 7 that this is not always the case, especially when special care is taken to reduce these overheads.

Our recipe for circuit design and optimization presented in this work, shown in Figure 1.2, is as follows:

Optimized Correction We analyze the fault paths through a circuit and determine where error correction sub-circuits are most needed. This technique is in contrast the current common technique of putting error correction "everywhere" in the circuit. This technique is discussed in Chapter 6.

Circuit Error Correction Encoding Starting with a logical application circuit, we automatically insert error correction procedures corresponding to a selection out of our library of QECCs. This is discussed further in Section 2.2.

Datapath Map Several previous works have suggested tiled dataflow architectures for high-level quantum computer design [64, 98]. We introduce methods to map to these tiled datapaths in Section 3.4.1.

Trap-level Layout We use a variety of heuristics to place physical gate and communication elements down on a substrate to give us a physical layout. The techniques we use to do this are given in Sections 3.2.1, 3.2.2, and 3.2.4.

Fault Extraction From a layout, we can identify sources of all types of errors: gate, communication, and memory. We discuss how we model faults in a layout in Sections 4.2 and 4.9.

Fault Probability Estimator We verify that our design meets some prescription for defect tolerance by simulating error propagation through the extracted fault model. These simulation techniques are discussed in Sections 4.1, 4.5, 4.6, and 4.7.

Through this design flow, we improve on the state of the art in quantum fault tolerance in the following ways:

Communication and memory in quantum circuits Due to the high-level view of circuits in existing quantum circuit evaluations, the extent to which qubit communication and memory affects the fault tolerance, latency, and area of a circuit is unknown. We present the first comparative evaluation of quantum error correcting codes that takes into account factors like communication and memory errors.

Constructive physical designs Our tool flow provides the first automated system for creating complete physical designs for given application circuit. This allows us to perform much more detailed estimates of area and resources required to a particular application.

Circuit Reliability & Efficiency Our error correction optimization techniques both reduce the amount of resources needed to implement a given logical circuit as well as reduce the number of fault points in the circuit. This has the effect of reducing both area and failure probability of the resulting design.

For the remainder of this chapter, we will review the basics of fault tolerant quantum computing as well as classical computer aided design flows. Chapter 2 will give an overview of our quantum computer aided design flow. This will give the framework into which all our main contributions fit. Chapter 3 will describe the methods in which we estimate communication in quantum circuits in order to account for non-gate errors. Chapter

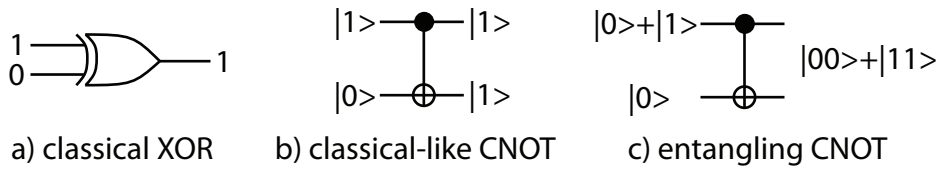


Figure 1.3: A comparison between a classical XOR and its quantum analog: the controlled not or CNOT. The CNOT gate is reversible, thus the additional output. Figure b) outputs the XOR result to the bottom bit. Figure c) shows the same CNOT when the input is a quantum superposition. In this case the output is an *entangled* qubit state, not representable as independent qubit values for the two outputs.

4 describes how we extract fault models from a circuit and how we simulate failure propagation given these models. Chapter 5 ties these two pieces together to analyze the errors in a high-level teleportation network and a low-level QEC code comparison. Chapter 6 talks about our fault tolerance optimization methods. The final chapter, 7 looks at the results of our tool flow on large quantum circuits such as adders and Shor's factorization algorithm.

1.1 Quantum Circuits

We will later talk about how to automatically synthesize, optimize, and layout quantum circuits. First we must understand what a quantum circuit is, and how it differs from the classical circuit model.

Classical computers in CMOS technology have spent a lot of time fighting "quantum effects" as feature sizes shrink into the 10s of nanometers range. A quantum computer, on the other hand, strives to amplify and utilize quantum effects such as entanglement and superposition as much as possible. The basic abstraction of a quantum circuit is a collection of quantum gates connected by wires. This model is similar to a classical circuit specification but there are two main differences:

- Quantum gates are unitary and therefore reversible [9]. This introduces the notion of scratch bits called *ancilla qubits* or simply *ancillae* in order to have the same number of inputs and outputs for each gate when simulating the behavior of non-reversible circuits.

- Due to the no-cloning theorem [104] qubits cannot be duplicated. This prevents any fan-out of wires in a quantum circuit.

Figure 1.3a shows quantum and classical XOR gates, the quantum version is known as a controlled-not (CNOT) gate as shown in Figure 1.3b. If the quantum inputs are not in a superposition state, the output of the gate is the same as the classical version (with the addition of another output for reversibility). If the "control" input to the CNOT gate is allowed to be a superposition, as in Figure 1.3c, things get more interesting. The resulting output state is an *entangled state* and has no classical analog. A confusing way to look at this is that both qubits are in both the 1 and 0 states but always the same.

Quantum circuits operate on these entangled superposition states and this is where the power of quantum algorithms comes from. In the end, the data cannot stay in a superposition though, in order to read out an answer from the quantum computer, the qubits must be measured so that the data can be presented to the classical world. The process of measurement collapses a superposition state into just one definite bit vector. Measurement also helps us understand the output state from Figure 1.3c. The entangled state $|00> + |11>$ means that when we measure, the resulting classical bit vector will be 00 or 11 (with equal probability).

1.1.1 Universal Gates

Due to the more complicated structure of quantum superpositions, there is no single 2-bit universal gates as in the case of the NAND gate in classical logic. Instead, one can use the reversible 3-bit toffoli gate as a universal gate. Since many quantum circuit technologies are practically limited to 1 and 2 bit interactions, we can construct a universal set of 1 and 2 qubit gates as shown in [9]. A standard universal set of 1 and 2 qubit quantum gates comes from [15] and is the CNOT (shown above as a reversible XOR), the Hadamard or H gate (which converts a bit value to a phase value and vice versa), the $\frac{\pi}{4}$ rotation gate, also known as the T gate, and the phase gate. These gates are shown in Figure 1.4 along with some additional gates we will use in the circuits throughout the paper. In reality, different elementary gates are easier or harder depending on which technology one is using. One more "gate" type is needed: measurement. In order to read out data from a quantum computer, it must be measured, measurement is also instrumental in another useful primitive, known as *teleportation*, which is discussed Section 1.2.4. Measurement is

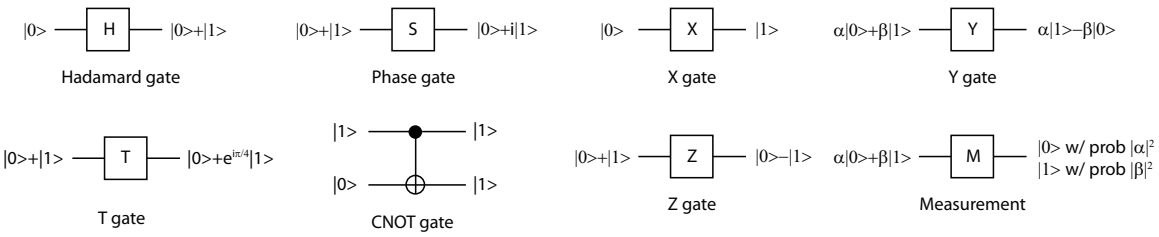


Figure 1.4: Basic gates for quantum circuits: this is a set of gates which supports a universal quantum computing model. The Hadamard gate converts bit values to phase values and vice versa. The phase, T and Z gates rotate the phase of the "1" qubit value by different angles. The CNOT gate is the same as shown in Figure 1.3 and performs the XOR functionality. The measurement "gate" measures a quantum state, returning a 1 or 0 and collapses any superposition to that value as well. The X is a bit flip, Z a phase flip, and Y a combination of both. The X, Y, Z, and phase gates can be generated by the other gates shown here but we include them since they are often included as physical primitives.

not a unitary gate which is why we do not include it in the universal set, but it is still a necessary quantum operation.

1.1.2 Quantum Decoherence

While the power of quantum superposition enables a lot of interesting computing, it comes at a cost. The sensitivity of a quantum superposition state lacks the dynamic feedback that stabilizes bits in digital logic. We have to allow a continuum of possible quantum states per qubit instead of 2. For this reason, the error rates of all operations on quantum data are much higher than operations in classical logic. Errors to quantum states cause what is called quantum state decoherence. Error rates to do anything in any quantum computing technology in the lab right now are in the range of $10^{-2} - 0.1$ errors per operation, this includes having qubits wait around, not doing anything. "Realistic" estimates for error rates in the foreseeable future are said to be around $10^{-5} - 10^{-2}$ errors per operation [91]. Compare this to CMOS transistor error rates which range from 10^{-20} to 10^{-15} errors per gate [83].

The gap is very wide so we would expect to have to pay more attention to errors in quantum circuits than in classical circuits. Indeed, as we mentioned earlier, we have shown that in many cases 95% of quantum circuits will be made up of error correction and

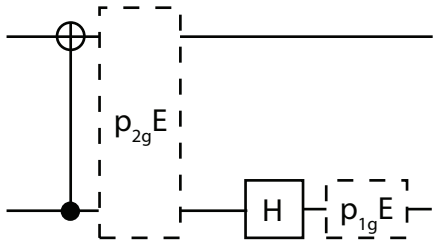


Figure 1.5: Above is a simple independent gate error model. After each gate, qubits involved in the gate acquire an error with probability p_{1g} for 1 qubit gates and p_{2g} for 2 qubit gates.

fault tolerance modules [48]. With error control circuitry imposing such a large overhead in a quantum circuit, it has not even been conclusively proven that the algorithmic gains promised by quantum computations will not be swallowed up by error correction overhead. The methodical synthesis of circuits with fault tolerance will help us better understand trade-offs between circuit performance and fault tolerance. We will show later in Chapter 6, that this overhead can also be reduced substantially, if we are more careful about our fault tolerance architecture.

Some physical sources of quantum operation errors will be covered in Section 1.3.5, but first we look at some of the general, technology-independent models currently used to classify errors and the error control techniques used to combat them.

1.2 Quantum Errors and Error Control

As we mentioned before, quantum error control is a dominant component to all quantum computer designs. We will now give an overview of where these errors come from and how previous works have built up fault tolerant architectures.

A quantum noise process can be viewed as a combination of application of random gates and measurement to qubits [70]. Figure 1.5 shows an example of a simple quantum circuit error model, widely used in many analyses. In this example, errors only happen after gate operations. Error probabilities are parameterized into single qubit and double qubit error probabilities. This simple model has no errors for inter-gate movement, there is also no temporal or spatial correlations introduced between errors in qubits.

In all quantum computing technologies, errors are abstracted into 3 different

sources:

Gate errors Depending on the physical technology, gates could involve complex sequences of applications of electrical and/or magnetic fields, current, and/or EM radiation applied to one or more co-located qubits. These gate processes can introduce errors from apparatus imprecision or tunneling effects between qubits. The abstraction of this error type is that each qubit involved in a gate has some probability of an error being introduced immediately after the gate is finished. Additionally, multi-qubit gates can propagate existing errors from one qubit to another. This point is discussed more in Section 1.2.2.

Movement/communication errors Qubit communication can involve either physical movement of coherent particles or gate-like operations to transfer state across fixed physical resources. In the former case, kinetic motion of particles can introduce motional heating and even particle loss. The abstraction often used is some amount of distance moved introduces a single qubit error with some probability.

Memory/idle errors Even when a qubit is sitting stationary, interaction with the environment, either through coupling with stray EM fields or contact with stray particles, can cause errors. Since there is no physical action the qubit is performing, memory errors are abstracted to a probability of error per unit of time it is stationary.

In studies of the effects of quantum noise, including threshold analyses (discussed in Section 1.2.3), quantum noise is usually categorized under the following characteristics:

(Non-)Local A local gate model accounts for the spatial distance that a qubit has to travel to perform successive gates. This communication overhead can introduce errors from the movement and also increase circuit latency and introduce additional memory errors.

(Non-)Markovian One can assume the environment with which the computational system undesirably interacts with has memory or no memory.

(Non-)Leakage Errors can take the computational system out of the computational basis. This type of error is called a leakage error and is very similar to data erasure errors in classical coding theory.

Figure 1.5 depicts a non-local, Markovian, non-leakage fault model.

A more general formulation for errors introduced into quantum state, known as *decoherence*, comes from [20]. In this formulation, a fundamental split is made between the "computational" space and the "environment". In general, any interaction with then environment leads to errors as it means uncontrolled coupling with stray quantum systems. Therefore, noise is modeled as interaction between the Hilbert space (space of all possible quantum states) of the computation, $\mathcal{H}_C$, and the Hilbert space of the environment, $\mathcal{H}_E$. A "superoperator", or a positive linear map over the computation Hilbert space, gives us a mathematical formulation of decoherence:

$$\mathcal{A}(\rho) \equiv \mathrm{tr}_E(P^E U(\rho^C \otimes \rho^E)U^\dagger) \tag{1.1}$$

$$= \sum_{k,l} \sqrt{\lambda_l}\langle g_k|U|\phi_l\rangle\rho\langle\phi_l|U^\dagger|g_k\rangle\sqrt{\lambda_l} \tag{1.2}$$

where ρ^C represents a probability distribution over possible quantum states in the computational basis, and ρ^E represents a distribution over all the states in the environment (i.e. "everything else"). The idea behind this general equation is that when computing errors, we look at the interactions between the computational states and the environment and then ignore the state of the environment to see how badly the computational states are messed up.

In general, the majority of work on error correcting codes and thresholds analyzes the simplest type of error: independent, non-Markovian, non-leakage errors. They then show how it can be extended in some cases to handle limited cases of some of the other, more difficult, error types. *Our work assumes a simple, Markovian, local fault model.* Chapter 4 discusses more details on the fault model we assume for our studies.

1.2.1 Error Correcting Codes

Qubit state coherence can quickly decline to unacceptable levels resulting in data loss if this data is unprotected. As in classical computing, redundancy may be used to combat high error rates. This can be done by way of Quantum Error Correction Codes (QECC) [70]. A QECC encodes a data or *logical* qubit, into an encoded block of qubits, similar to a classical error correcting code. It was mentioned before that quantum circuits must preserve a continuum of superposition states, so does this mean that QECC must

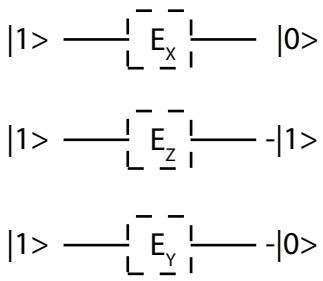


Figure 1.6: Types of errors corrected by quantum error correcting codes: The X error (E_X) flips the bit value, the Z error (E_Z) flips the phase difference between 1 and 0 by π radians, the Y error does both these things, flipping the bit and phase of the qubit.

correct a continuum of errors? Fortunately, the answer is no. These quantum codes are designed such that the error can be measured in the correction process without measuring the data superposition (which would corrupting the data). This means that the continuum of errors is collapsed to a choice of 3 different error types (vs. one error in the classical situation, the bit flip) and the correction process then looks a lot like its classical counterpart. The 3 error types are a bit flip, a phase flip, and both a bit and phase flip as shown in Figure 1.6. The bit flip (E_X) is the same as the classical bit flip error, the phase flip (E_Z)has no classical analog and is similar to flipping the relative phase between two interfering EM waves.

Encoded blocks can be hierarchically composed so we refer to the level of concatenation as the number of times that a code is recursively applied to the data. Figure 1.7 show a simple recursive encoding scheme using a 7 bit CSS code (explained later in this section). We refer to the lowest order bits in this encoding scheme as *physical qubits*, since these are the bits that correspond to physical two-level quantum systems in the circuit. The "Level 1 Encoder" from the top of Figure 1.7 takes a single physical data qubit and encodes it into 7 physical qubits, making a *level 1 logical qubit*. Assuming that each gate in this encoder has an analog for operating on a level 1 logical qubit (7 qubit encoded block), we can recursively encode to get a "Level 2 Encoder", shown on the bottom of Figure 1.7. This generates a level 2 logical qubit. The majority of studies into the error correcting abilities of concatenated codes have focused on the asymptotic properties of concatenation of this very 7 bit CSS code.

So now we have an encoded qubit which offers us some protection as long as the

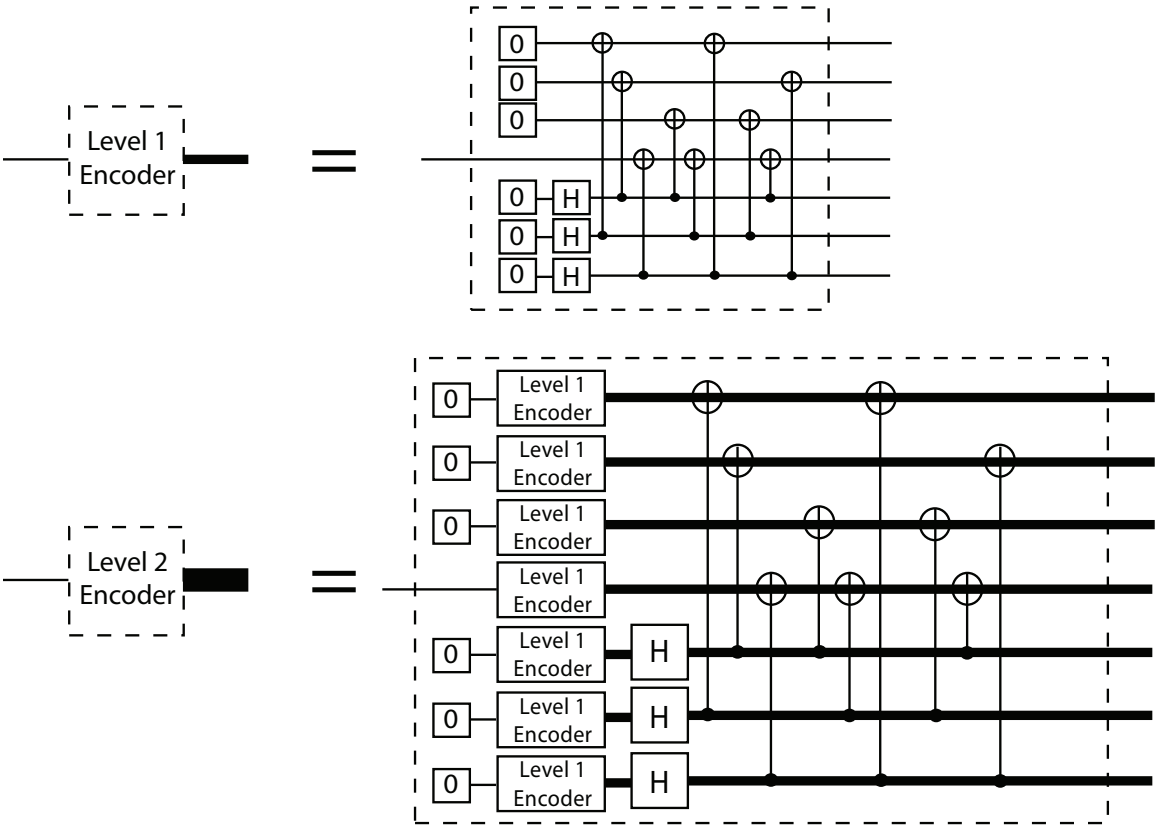


Figure 1.7: On the top, we are encoding a single data qubit into a 7 qubit block code (the [[7,1,3]] CSS code). The boxes with zeros indicate a preparation of a new qubit in the $|0>$ state, if the input qubit is a single physical qubit, this is a level 1 encoder, producing a level 1 logical qubit. The bottom figure is a level 2 encoder, using a level one encoder as a building block to produce a level 1 logical zero valued qubit.

qubit does not need to be acted upon by a gate. Since many consider gate operations to be the most error prone operations in a quantum circuit [103], decoding the qubit to do a gate is not feasible. This means that all gates in our quantum circuit must act on encoded data. We refer to these gates acting on encoded data as *encoded gates*. A single encoded or *logical gate* becomes a set of physical gate operations, dependent on which QECC is used.

Popular QECCs have the property that most encoded gates are implemented as "transversal" gates, as shown in Figure 1.8. Transversal gates are nice because they are simple and have good fault tolerance properties which we will discuss in Section 1.2.2. Unfortunately, it has been proven that additive quantum codes cannot have a fully transversal

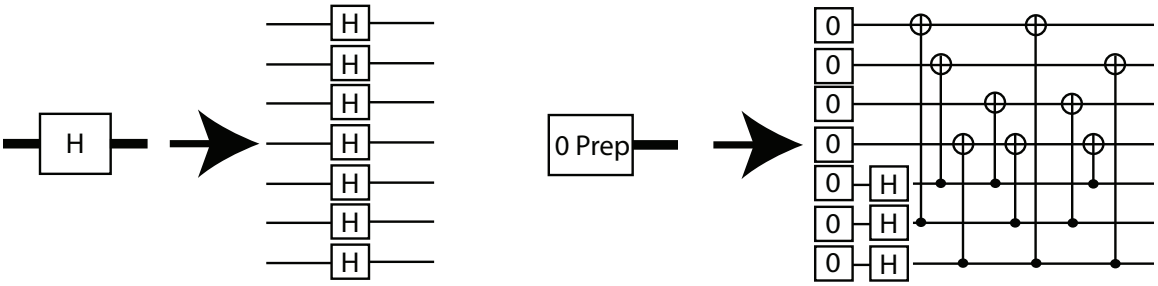


Figure 1.8: On the right is a transversal Hadamard gate over an encoded block of qubits, on the left is a non-transversal encoded zero preparation. A transversal gate does not perform any intra-block qubit interactions.

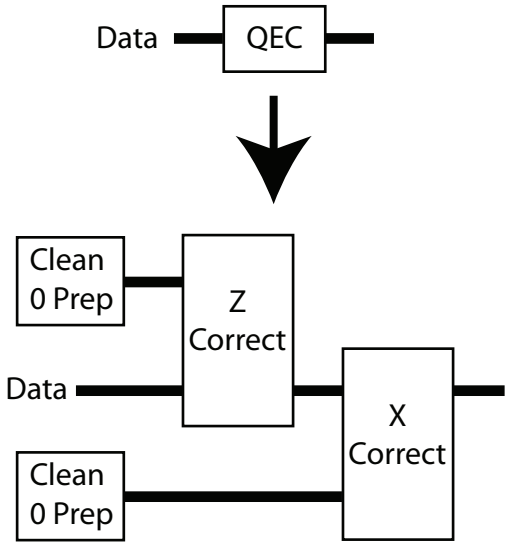


Figure 1.9: Steane-style error correction schemes have the following form: generate two encoded zero states then perform sequential Z and X correction operations.

universal gate set [106]. Thus, more complicated gate procedures must be devised for at least one of the encoded elementary gates we will want to perform, this will be discussed in Section 1.2.2. In practice, for many codes, the non-transversal gate is the $\frac{\pi}{8}$ or T gate. This gate will be used extensively in the adder circuits we will talk about more in Chapter 7.

Using quantum error correcting codes requires that errors are periodically corrected. In classical ECCs correction involves computing an error *syndrome* based on the

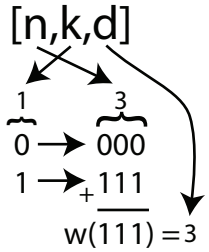


Figure 1.10: Classical error correcting codes use a $[n, k, d]$ notation to describe code parameters. n is the number of bits in the resulting encoded states, k is the number of source bits it encodes, d is the minimum distance between any two different encoded values for the code. For the 3 bit repetition code, the minimum distance is 3 (000 and 111 differ in 3 bit positions).

data values and then flipping the bit(s) which the syndrome identifies as erroneous. In the QECC case, the correction process is complicated by the fact that we cannot directly measure the data qubits to obtain their values or we will collapse the superposition and invalidate the computation. Instead, the correction process uses extra *ancilla* qubits that interact with the data qubits, the error information is distilled in these ancilla without transferring any information about the logical data value. Then, the ancilla is measured to get the error syndrome and bit and phase flip corrections (X and Z gates respectively) are applied to the data. Figure 1.9 shows this process.

The standard notation for quantum error correcting codes is similar to classical ECCs, Figure 1.10 explains the $[n, k, d]$ notation, a quantum code is described the same way but with an extra pair of brackets ($[[n, k, d]]$). The most popular code in quantum computing analyses today is Steane's $[[7, 1, 3]]$ code, which corrects a single error [85]. This code is often concatenated to correct more errors in bigger blocks.

Although, the number of QECCs constructed so far is much less than the number of classical ECCs, there still is already a large selection of codes to choose from when designing an error corrected circuit. Here is a sampling of (non-disjoint) classes of QECCs that people have come up with so far:

Stabilizer codes Originally formulated by Gottesman [35], these codes are specified by giving a generating set of operators which "stabilize" certain quantum states, meaning these operators do not change the states. These states are used to represent logical

bit values. Correctable errors move states out of the stabilized space, which can then be detected. It has been shown that a fault tolerant universal set of gates is possible for any stabilizer code [36], although they might be complicated.

CSS codes From Calderbank, Shor, and Steane [85, 18], this was one of the first code classes discovered. A CSS code is made up of two linear classical codes, on of them being a Hamming code. When using any CSS code, CNOT and H gates are transversal, which makes them attractive for encoding gates. The popular $[[7, 1, 3]]$ is the smallest CSS code. CSS codes are a subset of stabilizer codes.

Error-detecting codes While most focus has been on error correcting codes, error detecting codes play a central role in Knill's postselection fault tolerance technique [55]. The key idea here is that we are trading off overhead of encoding and storing larger qubit blocks for the cost of retrying every time an error is detected in a computation.

Decoherence free subspaces Instead of active correction operations based on measured syndromes, decoherence free subspaces (DFSs) take advantage of specific properties of the predominant noise present in the system and encode the data such that all the data values are in the eigenspace of the noise processes. Thus the data is invariant under some noise [61]. It is unlikely that data can be encoded in a state that is fully immune to noise, therefore proposals have been made to concatenate DFSs with QECCs to protect against more errors [60].

Operator/Subsystem codes Most codes are based on restricting the code space to a subspace of the total computational vector space. Subsystem codes reside in a subsystem of the computational vector space and correspond to a code space with different gauge possibilities [7, 4]. The claim is that correction is simpler than in subspace codes. Subsystem codes such the Bacon-Shor codes [7] are also CSS codes.

Entanglement-Assisted codes Most other types of codes do not make use of any external resources. These codes are used for communication and require the sender and receiver to have shared entanglement for the correction process [17].

In addition to these classes of codes, there are many codes that have been individually mentioned in the literature: quantum Golay codes [90], quantum BCH codes [38], quantum Reed-Solomon codes [39], quantum Reed-Muller codes [88], Bacon-Shor codes [4],

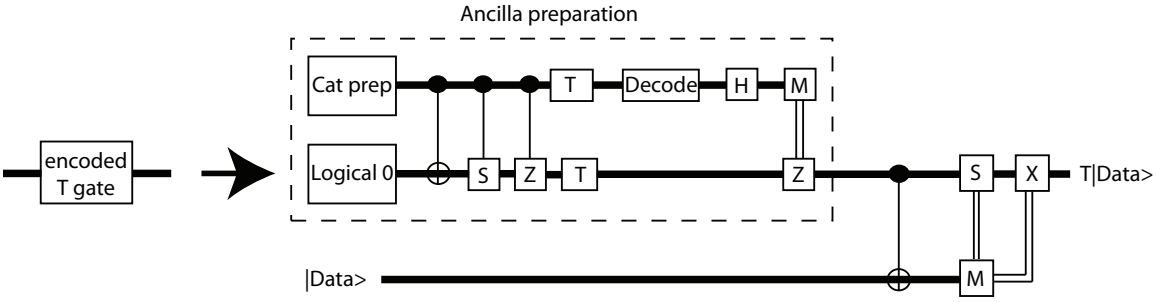


Figure 1.11: Performing a T gate involves preparing an encoded ancilla in the state $|0> + e^{i\pi/4}|1>$ (in the dashed box) and then interacting this ancilla block with the encoded data at the end of the gate operation sequence.

etc. In short, there are many codes to choose from. To date, the two studies to compare more than a couple codes in a common framework were Steane in [90] and Cross et al. in [25]. The focus on CSS codes.

In Chapter 5, we will provide another comprehensive code comparison, including a more complete error model. Like previous studies, we will focus on the CSS codes. Future work would involve a more inclusive comparison.

1.2.2 Fault Tolerance

It turns out that merely encoding data and gates is not enough to sufficiently limit errors in quantum circuits. You must also limit the *propagation* of errors when 2 qubits interact in a gate. This has led to a general theory of *fault tolerant* quantum computation [75]. The basic idea behind this is that a qubit in an encoded block of data can only be allowed to interact with a limited number of other qubits in the same block. The idea is that if a code corrects t bits, there should be clusters of no bigger than t qubits that interact in the block. In the case of the $[[7, 1, 3]]$ code, the code corrects $t = \lfloor \frac{d}{2} \rfloor = 1$ error so there should be no direct intra-block interactions (cluster size 1). Now we see another reason why transversal encoded gates are attractive: they are automatically fault tolerant for any code. In the case of necessary non-transversal gates, intra-block interactions may need to occur, therefore in this case, we use *ancilla* qubits to act as fault tolerant intermediaries in data qubit interactions.

An example of an ancilla-assisted gate is the fault tolerant T gate, shown in Figure

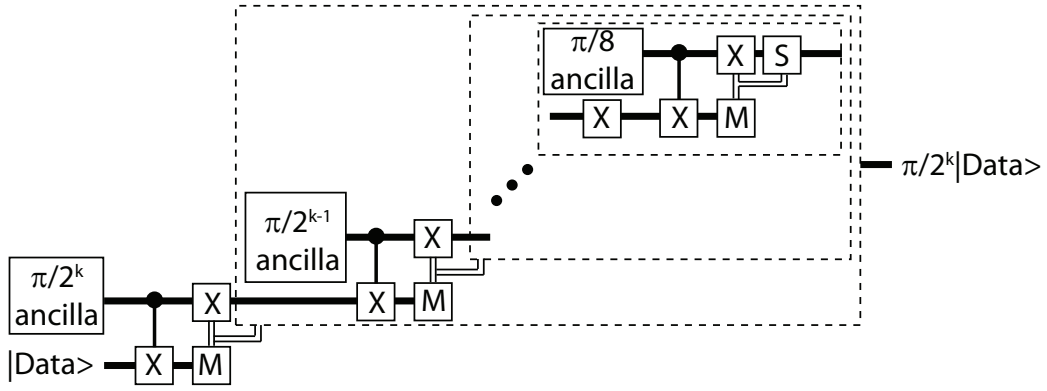


Figure 1.12: Fault tolerant $\pi/2^k$ gates can be performed recursively with a cascade of $\pi/2^i | i = 3...k$ ancilla factories and $k-2$ CX and X gates. Each measure gate output controls both the single qubit X gate and the compound gate involving more ancilla factories. Each measurement has a equal chance of giving the "correct" state, in which the remaining circuit is skipped or a "wrong" state in which a larger rotation has to be done to adjust the state. The actual output data from the circuit connects to the first quantum bit line associated with a correct measurement.

1.11. Notice that a majority of the work is performed in the ancilla preparation stage of the gate. In the case of a general rotation, the ancilla demand is even more extreme, as shown in Figure 1.12. In this case, to rotate by $2\pi/n$, we need on the order of $\log n$ encoded ancilla generated. All this ancilla can be done at any time before the actually gate is scheduled to be performed on the data. We analyzed this optimization in [48].

We mentioned the use of ancilla to perform error correction procedures. This must be done carefully because we cannot allow "dirty" ancilla to interact with data and allow ancilla to data error propagation. Fortunately, producing ancilla with very low error probabilities is much easier than reducing error in data because the ancilla is in a known state. Thus we can perform a process known as *purification* which takes some ancilla bits and produces fewer ancilla bits with less error probability (also referred to as higher *fidelity*). Also, we typically want to avoid interacting multiple data qubits with a single ancilla qubit because error can then propagate through the ancilla.

There are a number of different *ancilla preparation* and *correction* procedures from the literature, and very few comprehensive comparisons have been made between different techniques. Our automated circuit layout and error simulation tool allows a detail

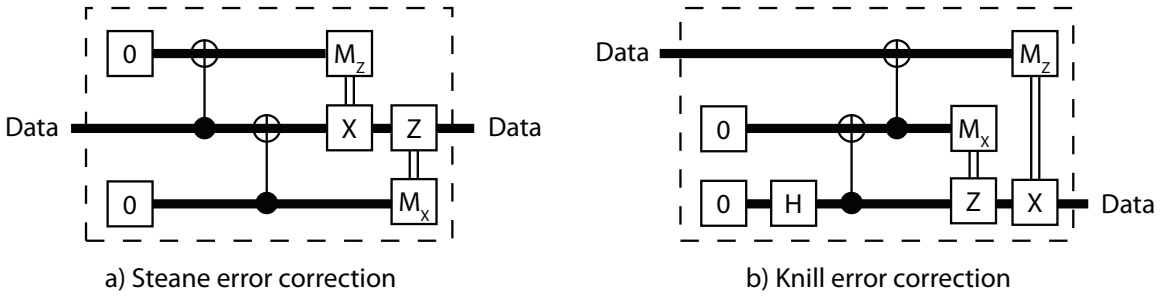


Figure 1.13: A comparison of two popular error correction strategies, the first, proposed by Steane, just CNOTs two encoded zero states with the data, the second, proposed by Knill, actually teleports the data qubit into one half of an EPR pair. Note that the set of operations performed are very similar.

comparison of failure probabilities, area resources for the gate network, and latency/time overhead of the different procedures. Results of this comparison are given in Chapter 5.

Reichardt has a nice qualitative summary of the different fault tolerant correction procedures in [77]. He identifies two of the most successful but different procedures (there are many variations on both):

Steane correction From [90], shown in Figure 1.13a, the procedure uses ancilla bits encoded in the same code as the data. CNOT gates are then applied between the encoded data and encoded ancilla in such a way that errors are copied to the ancilla but the logical data is not. The ancilla is measured and the error information used to apply bit or phase flips to the encoded data.

Knill correction From [55], shown in Figure 1.13b, the procedure effectively teleports data into a cleaner ancilla and the teleportation procedure itself eliminates errors.

In our study in Chapter 5, we focus on Steane correction in order to better compare to previous studies [90, 25]. Since the sequence of operations for both techniques is very similar, we do not consider this to be much of a limitation on our code study. The key difference is that Knill correction could be combined with teleportation-based communication to fold in some of the QEC overhead in some cases.

1.2.3 Noise Threshold Theorems

Much of the study of quantum fault tolerance and quantum error correction is focused on establishing thresholds. Threshold theorems rely on concatenated codes, and try to prove that concatenating various QECCs will lead to an asymptotic elimination of error in a quantum circuit. The original work was done by Aharonov and Ben-Or [2], and Kitaev [54].

In Aharonov and Ben-Or's work, they bound the number of errors possible in an single error correction block or "rectangle" before the rectangle fails, then recursively replace elementary gates in that rectangle with encoded, corrected gates. This recursive application of encoding leads to a exponential decrease in error with each successive recursion. While they set up the analytical framework that would later be followed by other threshold analyses, they do not go into very much detail on how blocks are composed or where the "worst" error paths are, thus they must produce very conservative upper bounds on tolerable gate errors (10^{-6}). Aliferis et al. [5] did a substantially more detailed analysis of the composition of concatenated blocks and critical error paths through these blocks to produce a more aggressive threshold of 2.73×10^{-5}. Besides these works, considerable effort has been put forth to find numerical [55, 95] and analytical [5, 77] thresholds for various codes, under assumptions of various noise models, different locality/communication models, correction techniques, etc.

With the exception of [97, 8, 95, 56], little attention as been placed on gate locality and qubit communication costs. I believe that communication failure rates could have significant influence on a threshold estimates. [95] showed that movement error does not eliminate the possibility of an error threshold for the concatenated $[[7, 1, 3]]$ code. There have been no results of the relative affect of movement errors and latency compared across a variety of code architectures.

In Chapter 5, we will extend the numerical threshold results from Svore et al. [96, 95]. In this work, there is a detailed study of a hand-optimized abstract layout of qubit gates implementing an encoded, concatenated version of each gate type from a universal set, using the $[[7, 1, 3]]$ code. Svore's work focused on the particular code with a particular correction and ancilla preparation strategy. We believe that choices in all these factors can lead to vastly different resource requirements and performances in the circuit, including different threshold estimates. In fact, Svore's final comment in [95] says that exploring the

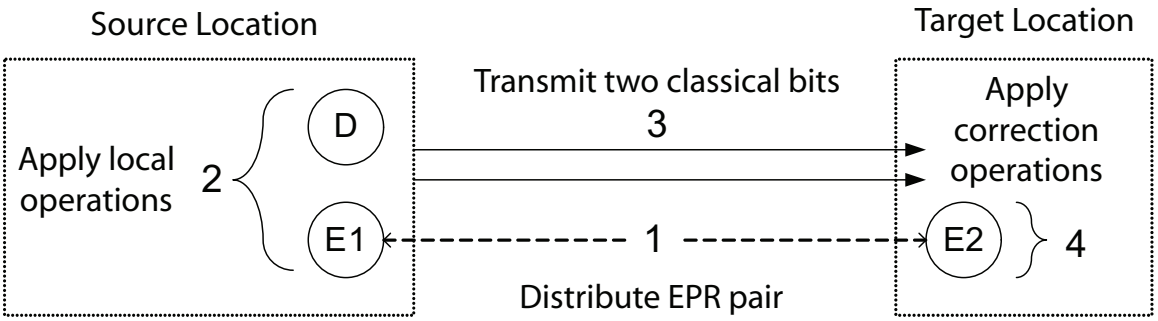


Figure 1.14: Teleporting data qubit D to the target location requires (1) a high-fidelity EPR pair (E1/E2), (2) local operations at the source, (3) transmission of classical bits, and (4) correction operations to recreate D from E2 at the target.

space of different codes is a crucial next step in these numerical threshold studies.

1.2.4 Communication

In order to perform any two-qubit quantum gate, the two qubits must be physically adjacent. Similar to any other quantum operation, qubit movement could be affected by high error rates. Drawing on our work in [49], we make a distinction between short range movement, performed by whatever physical movement primitive is available, and long range movement performed by teleportation.

Teleportation

A useful circuit primitive for communication is a teleportation unit. Figure 1.14 gives an abstract view of teleportation [10]. We wish to transmit the state of physical data qubit D from the source location to some distant target location without physically moving the data qubit (since that would result in too much decoherence). Figure 1.15 shows the circuit representation for this operation (note that E1 and E2 still must be physically separated after the CNOT).

We start by interacting a pair of qubits (E1 and E2) to produce a joint quantum state called an *EPR pair*. Qubits E1 and E2 are generated together and then sent to either endpoint. Next, local operations are performed at the source location, resulting in two classical bits and the destruction of the state of qubits D and E1. Through quantum entanglement, qubit E2 ends up in one of four transformations of qubit D's original state.

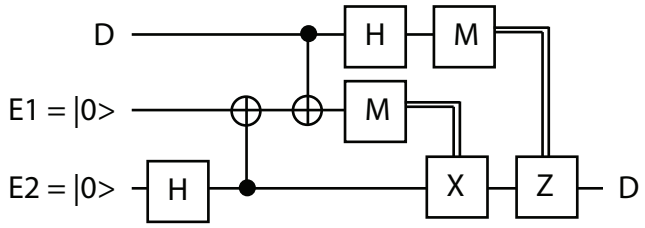


Figure 1.15: Circuit representation for the teleportation operation: The first Hadamard and CNOT gates prepare qubits E1 and E2 in the EPR state. One half of the EPR pair is CNOTed with the data followed by a Hadamard and measurements. The measurement results (classical information represented by double bit lines) are used to apply X and Z gates to adjust the final state.

Once the two classical bits are transmitted to the destination, local correction operations can transform E2 into an exact replica of qubit D's original state [1]. The only non-local operations in teleportation are the transport of an EPR pair to source and destination and the later transmission of classical bits from source *to* destination (which requires a classical communication network).

We can view the delivery of the EPR pair as the process of *constructing a quantum channel* between source and destination. This EPR pair must be of high fidelity to perform reliable communication. As was discussed in Section 1.2.2, purification permits a trade-off between channel setup time and fidelity. Since EPR pair distribution can be performed in advance, qubit communication time can approach the latency of classical communication; of course, channel setup time grows with distance as well as fidelity.

In order to perform teleportation, two adjacent ancilla qubits are interacted to enter an entangled state, making them an EPR pair [70]. Each qubit is shuttled to one endpoint of the teleportation. After some local operations at either endpoint and some classical communication between the two, it is possible to transfer the unknown state of some qubit at the source to the EPR qubit at the destination. The trade-off includes extra work to set up the teleportation, but for the benefit of minimal noise on the data qubit being teleported (since most of the work was done on the EPR qubits).

As discussed in our work in [47], qubits were clustered into regions in which we used physical transportation for data qubits and then inter-cluster data movement was

[1] Notice that the no-cloning theorem is not violated since the state of qubit D is destroyed in the process of creating E2's state.

done via teleportation. Dedicated teleportation units were used for each cluster and EPR distribution channels that provided the needed ancilla were set up between the units. These EPR channels were structured to use only physical movement or a combination of physical movement and chained teleportation across shorter distances. Section 5.4 investigates a number of different resource trade-offs in terms of the amount purification performed on EPR pairs going through these distribution channels.

1.3 Quantum Computing Technologies

Many technologies have been proposed to implement the quantum circuit model we just presented. Far fewer, only a handful, have been experimentally demonstrated to coherently manipulate quantum state. Optical quantum computing systems suffer from decoherence through spontaneous emission of photons, superconducting systems from external field fluctuations and $1/f$ noise from trapped electric charges, solid state techniques from similar $1/f$ noise attributed to stray charges near the qubits. The technology I will be focusing on in this work will be trapped ion quantum computing, the details of the technology and the quantum noise processes involved will be given in Section 1.3.1, but essentially, the main noise processes relate to applying lasers to perform gates, including thermal fluctuations and spontaneous emission and heating during data communication.

The substrate technology we choose for our study is based on *trapped ions* [23, 67]. In this section, we will discuss the basic operation of an ion trap quantum computer and allude to the various issues that arise when trying to control the system. We highlight aspects of the system that will require novel architectural decisions to control.

1.3.1 Ion Traps at a Glance

The target technology for our toolset will be ion traps, which has shown potential for scalability [51]. In this technology, a physical qubit is an ion, and a gate is a location wherein an ion is trapped so it may be operated upon. The ion is both trapped and ballistically moved by applying pulse sequences to discrete electrodes which line the edges of ion traps (see the left of Figure 1.16). The ion moves along in a potential well created by the control electrodes.

Gate operations are performed by precise laser pulses aimed at trapped ions. Measurement of a qubit is performed by exciting the target ion with a different frequency laser

Physical Operation	Error Set 1 [32]	Error Set 2 [91]	Latency in (μs) [73]
One-Qubit Gate	10^{-6}	10^{-4}	10
Two-Qubit Gate	10^{-6}	10^{-4}	100
Measurement	10^{-6}	10^{-4}	500
Zero Prepare	10^{-6}	10^{-4}	510
Straight Move ($\sim$30 μm)	10^{-8}	10^{-6}	10
90 Degree Turn	10^{-8}	10^{-6}	100
Idle (per μs)	10^{-10}	10^{-8}	N/A

Table 1.1: Error probabilities and latency values used by our CAD flow for basic physical operations

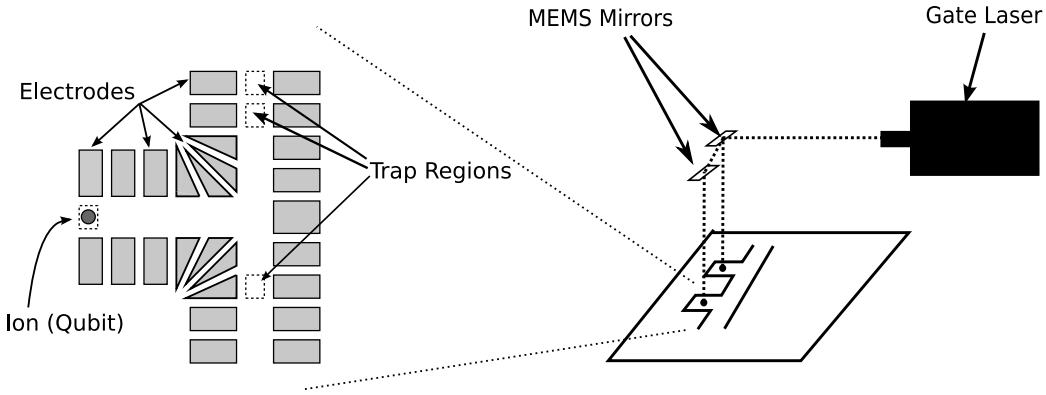


Figure 1.16: Simplified ion trap technology view. Ions (qubits) are trapped between electrodes in the trap regions. Ballistic movement of ions is performed by changing the voltages of the electrodes. A laser is routed to the location of the ions to perform a gate.

pulse and then detecting fluorescence using a CCD. Laser beams can be split and routed by an array of MEMS mirrors to simultaneously fire on multiple gate locations, thus allowing SIMD gate operation [52] (Figure 1.16 from [74]). A working ion trap quantum computer will require coordinated control of trap electrodes, lasers, CCDs, and micro-mirrors to perform the proper concurrent operations to implement quantum gates. Here is a break down of these components.

1.3.2 Trap Electrodes

In order to properly confine a qubit (ion) in a trap, the electrodes around the qubit must be precisely controlled to ensure that the ion does not unexpectedly move into adjacent traps, fly out of the top of the trap channel into the vacuum, or adhere to the electrode surfaces. Ballistic movement of qubit ions between traps is likewise achieved by coordinated application of changing voltages to all the electrodes near the ion being moved to generate the precise electrostatic attractive and repulsive forces necessary.

To get an idea of the type of electrode control necessary to move and confine ions, [45] has the details of an experimental demonstration of a qubit turning around a corner. This procedure can be broken down into sequences of 5 pulses with at least 4 discrete voltage levels on about 15 different electrodes. Coordinating a large number of of physical ion qubits necessary to do a useful computation (at least in the 100,000s) would require concurrent control of a million electrodes. As we just mentioned, control of each electrode will be more complicated than just turning it on and off, as it will have more than two voltage levels. Due to the need for a million trap electrode controllers to drive the appropriate voltages, these structures can benefit greatly from logic reuse.

Fortunately, there is substantial regularity in both the trap layouts we use to design the computer and the ion qubit movement through these regularly laid out blocks. We could imagine grouping adjacent sets of traps into blocks, similar to the layout proposed in [8]. Trap blocks with the same geometry could be controlled by the same replicated logic block. The block controller could then accept directives like "turn qubit around corner" or "hold qubit". Adjacent blocks would then have a simple handshake that would enable local decisions on when one trap block is done confining a qubit ion and when its neighbor takes control. In addition to simplifying global control of electrodes, this could also also eliminate potential skew problems in a monolithic controller, which could result in qubit ion errors or even total loss of a qubit.

Finally, it is worth mentioning that the voltage pulse sequences required to move ions are highly dependent on the exact geometry of the fabricated electrodes [79]. Thus, it may make sense to have a firm abstraction layer between the higher level control architecture and the lowest level trap electrode controller.

1.3.3 Gate Lasers

While trap electrodes are all that are needed to move qubit ions, quantum gates are performed by moving qubit ions to designated traps and then applying laser pulses to them. [80] and [81] have detailed listings of the laser pulses used in an experiment to apply a set of one and two qubit gates. From this listing, we note that each gate could take about 3 or 10 separate consecutive laser pulses, depending on whether it is a single or double qubit gate. Each of these pulses must be applied for a reasonably precise amount of time. [42] and [81] show the qubit ion energy state transition curves under laser application. The important thing to note here is that qubit values are oscillatory in the time evolution under laser application, thus the amount of time the laser pulse is applied is critical in performing the correct gate. The approximate oscillation frequency of the ions used in many of these experiments is around $200\mu s$, thus in order to maintain a gate error of less than 10^{-4} or so, we would need to control laser pulse length to a resolution of roughly $200\mu s \times 10^{-4} = 20ns$.

In addition to precise laser pulse length, substantial optics are necessary to sufficiently focus the laser to a narrow enough beam width in order to address individual ions within the trap. As mentioned in [69], two qubit gates require qubit ions to be adjacent in a single ion trap with a distance between them around 7-20μm, leading to a requirement of a beam width of around $5\mu m$. In this particular experiment, obtaining such a resolution was achieved with a rather large Nikon lens. Also mentioned in [69] is the need for a laser with a very stable frequency, one that is within 1-kHz of the precise transition frequency between qubit ion energy levels. This precludes the use of miniaturized semiconductor laser diodes from any current fabrication technology. [2]

Due to the large size (and probably expense) of the gate lasers and focusing optics, we see a strict resource limitation on the number of laser beams we can produce for our quantum computer. Additionally, double qubit gates require up to 4 different frequencies of laser light, so in order to perform a single gate, we may need 4 large laser apparatuses. The one thing we *can* miniaturize is an optical system to divert and split the already focused and stabilized laser beam to deliver them to the particular trap locations. The technology of electro-mechanical micro mirrors has already been applied on a large scale to

[2]Semiconductor lasers have beams consisting of multiple frequencies due to a large number of available modes just above and below the band gap where the electrons and holes recombine. Additionally, the band structure is highly sensitive to local temperature and current fluctuations, meaning that even if a single mode could be isolated, that particular mode would fluctuate by an unacceptable amount [43].

commercial optical routing technologies [14] and is capable of deflecting beams with over 1000 individually addressed micro mirrors.

For the above reasons, we assume a small number of lasers and a very large and flexible system for routing and aiming the limited number of actual lasers. This naturally lends itself to a SIMD design with individual laser beams being split and routed to many trap locations, allowing a single gate type to be applied at many locations simultaneously using one laser. This imposes a globally synchronous model of operation at the lowest level where large numbers of physical gates require synchronization to be performed by a limited number of lasers.

1.3.4 Measurement

A measurement operation consists of the application of another laser frequency, separate from those needed for gates, and the collection of light fluoresced from the ion with a CCD camera. If an ion is measured in the one state, it fluoresces, if it is in the zero state, it does not. Thus we must observe whether a particular ion trap location is emitting light when the measurement laser is applied.

[69] shows an apparatus set up to perform this collection via CCD. Following the SIMD model for all our operations on qubits due to laser scarcity, we plan to use a single large, high-resolution CCD camera positioned above the entire ion trap computer, with a lens between them to resolve the micrometer scale distances between fluorescing ions. All the measurement lasers are applied synchronously, and once enough time has passed to collect sufficient photons, we read out the image on the CCD and process it to determine which sites were fluorescing.

1.3.5 Noise model

As mentioned in Section 1.2, we categorize error sources in a quantum circuit as being from gates, movement, and memory. Here is a break down of ion trap specific error processes for all three categories:

Ion interaction gates Since gates are performed in trapped ion QC using the interactions between the ions and laser light, there are a number of laser related things that could go wrong. [72] details many of the problems. They can be separated into classical and quantum effects. The classical effects being that the if laser alignment or laser

power is not as one expects, the duration of the gate will be incorrect. Thus, if one is applying a rotation gate with a misaligned or under-powered laser, a qubit will be "under-rotated" if the laser is switched off after the calculated gate time. These problems are not viewed as fundamentally difficult in the case of a single gate, since it is simply a calibration issue, but in the case of systems with many gate locations, this calibration issue could become significant.

The second source of errors comes from quantum fluctuations of the laser field, which causes spontaneous photon emission. A spontaneously emitted photon can couple with the ion-qubit state, which effectively measures the state because the photon then leaves the quantum circuit and it must be counted as measured. The work by Ozeri et al. [72] estimates that all these gate errors can be reduced down to yield a global error probability of 10^{-4}.

Ballistic movement Movement errors come from the ion accumulation of vibrational energy as it is pulled between the trap electrodes in its potential well. Some of this vibrational energy is removed by different cooling techniques, including Doppler laser cooling (using a blue-shifted laser to absorb kinetic energy from ions in motion) and sideband cooling [103] (using a sympathetic ion to dampen the motion of the target ion).

Ion idling Even when an ion is sitting stationary, there are error effects that can corrupt data. Fluctuations in electrical and magnetic fields as well as stray photons from lasers applied to nearby gate locations could all interact uncontrollably with the qubit state.

With this knowledge about ion traps in hand, we can look at how we would build these physical structures into larger quantum circuits.

1.4 Classical Computer Aided Design Flows

The studies built upon in this work rely on techniques from computer aided design (CAD) tools for classical circuits [27, 82]. Before we discuss our quantum design flow, we review some of the parts of a classical CAD flow. A vastly simplified version of a typical classical circuit CAD flow is shown in Figure 1.17. The purpose of the flow is to take in some sort of abstract circuit specification and produce a physical design that can then be

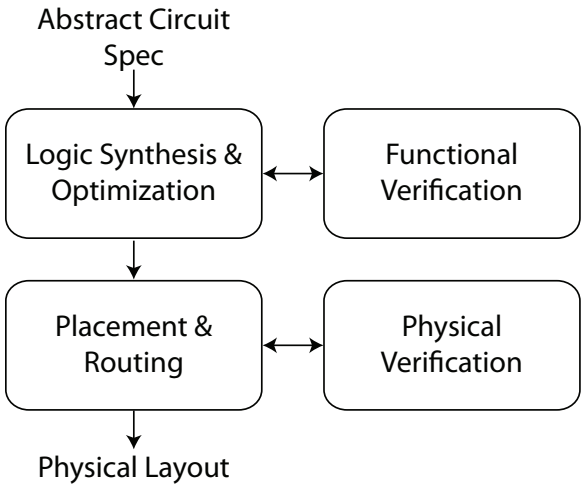


Figure 1.17: Simplified view of a classical computer-aided design flow. A user-specified application circuit specification is first synthesized into some sort of gate network, then physical components are geometrically mapped to a substrate to make physical design. Verification steps ensure equivalence between stages.

fabricated. The abstract spec is usually given in languages like Verilog [99], VHDL [62], or higher level languages like SystemC [40]. The physical design depends on the underlying technology but usually consists of some sort of geometric specification of gate-like and wire-like configurations.

1.4.1 Logic Synthesis and Optimization

The logic synthesis step of a CAD flow is to take a high level specification of the circuit's behavior and create a network of basic logic gates and connections between them that faithfully implements the high level specification. The resulting gate network is typically loosely tied to the basic operations available on the technology we are synthesizing for.

Since the high level constructs could potentially have many functionally equivalent translations to a gate network, the synthesis and optimization stage can iterate through many different networks, trying to minimize circuit latency, gate count, etc.

1.4.2 Functional Verification

Due to the potentially complex interactions from different synthesis and optimization methods in the previous step, it is desirable verify that the synthesized gate network works the same way as the user specification. This step can consist of a variety of methods including *formal verification*, where a mathematical models of the two designs are compared or *logic simulation* where the designs are simulated with identical test inputs to make sure they produce the same output.

1.4.3 Placement and Routing

The individual gates in the synthesized network are mapped into physical elements that are geometrically onto a substrate. The goals of this *placement* stage is to convert each gate element into a physical element in the given technology and to place gates that are directly connected in the network in close proximity to each other. After this, wires are *routed* between the physical gates. Since both wires and gates take up physical space, the placement and routing stages are iterative and converge on a design where everything fits onto the substrate.

1.4.4 Physical Verification

Next, this physical design must be verified that it meets requirements imposed by the fabrication technology. This is typically done through '*design rule checking* against a detailed set of rules of allowable element geometries. It also must match the functionality of the gate network from the previous stage. This is typically done by extracting a more abstract circuit representation from the physical design and comparing it to the gate network.

1.4.5 Metrics

Finally, there needs to be some set of measurements to be used by the classical CAD flow to decide how good a job it is doing. Typically, we are interested in evaluating the quality of circuit optimization and layout.

In terms of circuit optimization, we are interested in the following:

Gate count Reducing the total number of gates can help improve the area of layouts, as well as total resources needed, including total delay and power as well.

Critical path length Reducing the length of the longest chain of dependent gates (the critical path) can help minimize the total circuit delay.

The final layout of an optimized circuit is what determines the cost and performance of a device. The metrics to consider here are:

Area The overall size of a layout directly impacts ease and cost of fabrication. Also, smaller layouts can contribute to lower power usage and less communication delay due to shorter wires.

Delay In many cases, runtime performance is the most important measure of a design. Getting more work done in less time is often the driving factor for new designs and optimizations.

Reliability Device reliability is becoming more of a concern as transistor feature sizes get closer to atomic scales, since quantum effects as well as fabrication equipment precision become more of an issue. There are a broad array of techniques to improve reliability, at all different levels of granularity in the design. In the end, we are interested in the overall probability that a permanent or transient fault will corrupt data such that incorrect output is obtained from the device.

We will see in the next chapter that we consider many of the same metrics when evaluating a quantum circuit. However, we will see reliability take a much larger role in determining the fitness of a design in the quantum realm.

Chapter 2

Overview of Computer Aided Design for Quantum Circuits

We will show how the process of taking a quantum circuit description and creating a physical layout is similar to the classical case. We will start by discussing some of the differences between classical and quantum circuits:

Fault frequency Errors are many orders of magnitude more likely in quantum logic than in classical, this places an additional requirement on a quantum circuit for very strong fault tolerance.

Classical control Implicit in everything mentioned so far is a network for controlling qubit motion and gate operation. We address some of the problems of *classical control synthesis* in this CAD flow. Along with any physical layout of a circuit, we also generate a control schedule to implement the movement and gate operations on the qubits.

Synchronicity Related to the previous item, since a classical control network is available, all quantum gates can be essentially synchronous. This means that we can reuse physical gate locations in a layout to perform more than one gate in the circuit specification. This will reduce movement (reducing movement error) and reduce the area, making fabrication easier.

Reversibility The reversibility constraint on quantum logic requires the use of many more ancillary qubits to be created and tracked throughout the course of the computation.

In addition to the differences in the quantum and classical circuit models, there are differences in the underlying technology used to lay out our circuits. As mentioned in Section 1.3.1, we focus on ion trap quantum computing in this study and so to compare ion traps with classical CMOS:

Bit persistence In ion traps, qubits are physical entities that cannot simply be created or dissipated after the value on them is no longer useful. Dead physical qubits must be disposed of or recycled, and new qubit values must be allocated a new physical ion. The requirement for having many ancilla qubits to implement reversibility has a large impact on this requirement because many qubits are created and destroyed in a quantum circuit.

Planar wiring Qubit ions are suspended in a vacuum above the electrodes and must have space to float along surface channels, therefore it is unlikely there will be more than one layer of ion trap "wiring" on the fabricated chip. Thus, all wiring crossings are actual 4-way intersections where only one direction can be operational at a time. This impacts the area used to place dedicated channels for ion movement, as well as scheduling of ions along potentially shared channels.

Multiplexing resources Since qubits have a physical extent, different qubit values can share a channel/wire in a circuit as long as they are spaced far enough apart to limit unanticipated interactions. Thus, wires can be multiplexed rather easily, which is important since the number of wires is limited to what we can fit in the plane.

Communication cost metrics Strict Manhattan distance is not an accurate measure of wire length because preliminary studies have shown that turning corners and traversing intersections will be more time consuming and acquire more vibrational heating (and errors) than moving straight through a one-way channel [74, 45].

The quantum EDA system we have developed is modeled after a classical EDA tool flow, but accounts for many of these differences between quantum and classical circuits. Figure 2.1 shows the currently available components in our CAD flow. As mentioned earlier, our tools rely on a relatively simple input specification and do not currently do much circuit synthesis from higher level descriptions. Our toolset is "bottom heavy" in that we are interested in getting a detailed physical design that implements simple gate-level circuits.

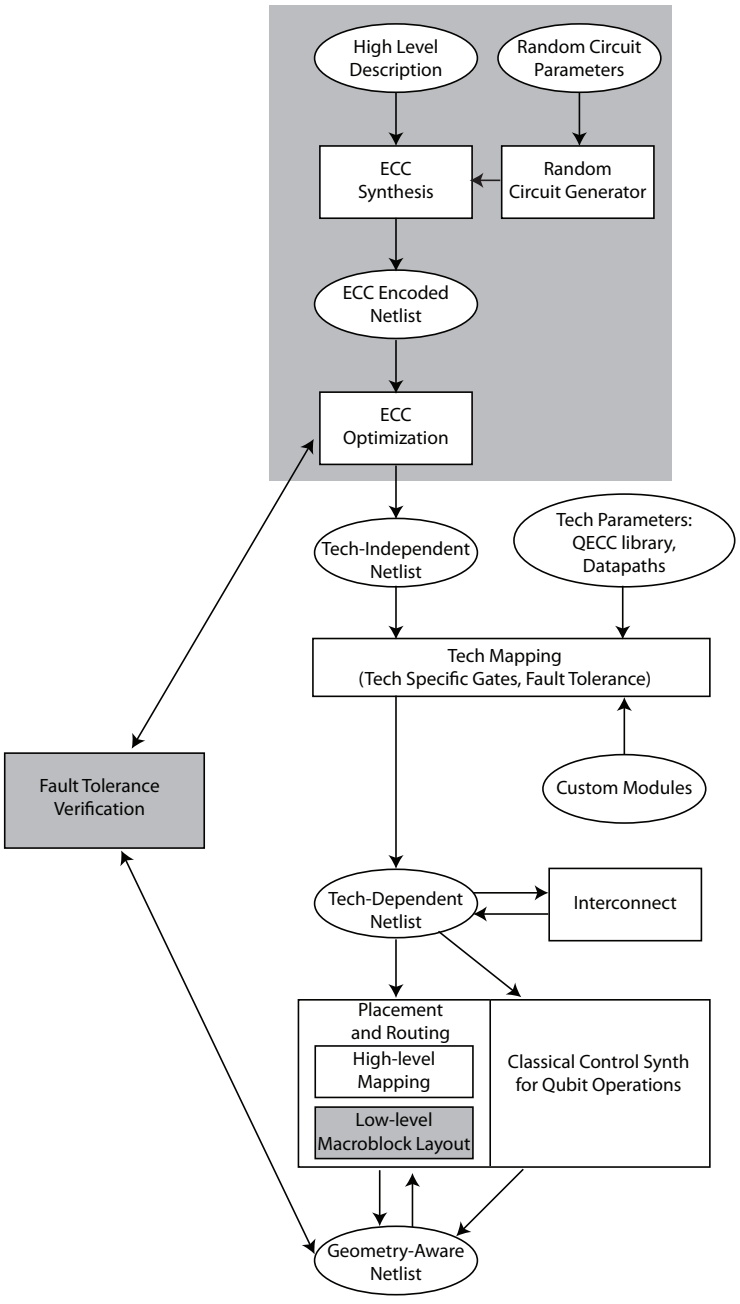


Figure 2.1: A high level view of our computer-aided design flow for quantum circuits. The highlighted blocks denote the contributions focused on in this work.

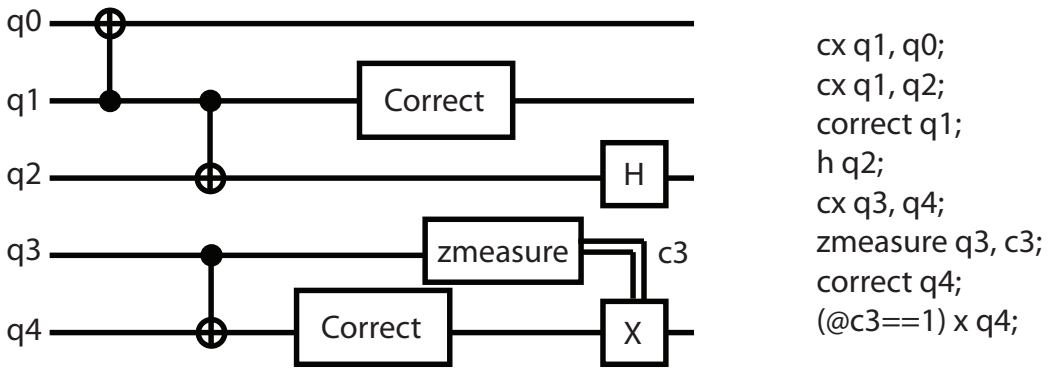


Figure 2.2: A quantum circuit and the equivalent QASM instruction stream representing it.

The components highlighted in grey are the focus of this work and will be covered in great detail, but for now, we will give a brief description of all the components.

2.1 Application Circuit Specification and Representation

The primary method for input of application circuits into the CAD flow is the use of the QASM description language. The original QASM was first introduced by Balensiefer et al. in [8]. QASM is similar to the classical MIPS assembly language [44]. Basic quantum operations and qubit operands, similar to classical registers, are listed in the order in which they are supposed to be executed.

The full QASM instruction set that we use is shown in Table 2.1. The instructions that do not introduce errors are *virtual* instructions used for bookkeeping of qubit states or classical information and do not correspond to actual physical quantum operations. We note that the x/zcorrect operations are not error prone because they are only virtual instructions that do qubit error state updates. They do not correspond to the entire error correction process which contains many error-prone physical gates. We discuss the x/zcorrect and x/zverify instructions in more detail in Section 4.3.

Figure 2.2 shows an example of a quantum circuit and its QASM specification. In this example gates/instructions are read from top to bottom in order, thus gates dependent on the output of earlier gates appear later in the instruction stream. Two qubit gates like the CNOT or "cx" (controlled-X) take the names of 2 qubit registers, the first one

Category	Name	Errors?	# of (cla/qu)bits	Description
Pure Quantum	h	yes	1	Hadamard gate, translates between X and Z basis
	x	yes	1	Bit flip
	y	yes	1	Bit and phase flip
	z	yes	1	Phase flip
	s	yes	1	Phase gate: phase rotation by $\pi/2$
	t	yes	1	T gate: Phase rotation by $\pi/4$
	cx	yes	2	CNOT gate: controlled-X gate, bit flip on target based on control
	cz	yes	2	controlled-Z gate, phase flip on target based on control
	cphase	yes	2	controlled-phase gate, phase rotation by $\pi/2$ based on control
	xprepare	yes	1	prepare input qubit in a particular state in the X basis
	zprepare	yes	1	prepare input qubit in a particular state in the Z basis
	correct	no	1	logical-only operation representing a correction step, encoded gate implementation is code dependent
Pure Classical	or	no	variable	Set output bits based on logical or over all input bits
	xverify	no	variable	Verify that there are no X errors on the classical syndrome bits, sets output bit if there are errors that are not undetectable by the code. Exact syndrome check is code dependent.
	zverify	no	variable	Verify that there are no Z errors on the classical syndrome bits, sets output bit if there are errors that are not undetectable by the code. Exact syndrome check is code dependent.
Quantum-Classical	xmeasure	yes	2	Sets classical bit based on quantum bit value in the X basis
	zmeasure	yes	2	Sets classical bit based on quantum bit value in the Z basis
	xcorrect	no	variable	Corrects bit flip (X) errors on input qubits based on the values of input classical bits
	zcorrect	no	variable	Corrects phase flip (Z) errors on input qubits based on the values of input classical bits
	(predicate)	no	variable	execute given quantum or classical operation if list of predicates are all satisfied

Table 2.1: Summary of all the quantum instructions we use.

being the control and the second, the target. Correction gates operate on a single logical qubit. A measurement gate takes in a qubit and outputs a classical bit; classical bit register names typically starting with a "c". In this example "c3" is a classical bit measurement outcome which then predicates the execution of the last "x" gate. Predicates only compare classical bits to a constant value (no quantum bits) and determine whether the gate they are guarding is executed. So in this example, if the zmeasure outcome sets "c3" to 1, then the "x" gate will be applied to "q4" later.

We have augmented the basic QASM language to handle large scale, modular designs. Modules can be defined hierarchically, composing larger modules from sub-modules. In a QASM definition of a circuit we explicitly declare qubit state and quantum gates. Here is an example of a 1-bit quantum adder in QASM:

```
1  qubit cin;
2  qubit cout;
3  qubit a;
4  qubit b;
5
6  input cin;
7  input a;
8  input b;
9  toffoli cout, a, b;
10 cx b, a;
11 toffoli cout, cin, b;
12 cx b, cin;
13 output a;
14 output b;
15 output cout;
```

The program starts with a declaration of qubit states or "registers" declared with the *qubit* keyword. These states will later be mapped to a physical element that can represent a 2-level quantum system. The qubit declarations are followed by a sequence of gate operations on the qubit states. In this example, we use 3 qubit *toffoli* gates and 2 qubit *cx* (CNOT) gates. This circuit takes a carry-in bit, *cin* and two input bits, *a* and *b*. The sum output is in *b* and the carry-out is in *cout*. We also use the special purpose virtual instructions *input* and *output* to specify which qubits would be set as input/output for the circuit

In addition to simple sequences of gates, we can specify hierarchically structured

programs through use of our added support of *modules*. We can define a sequence of qubits and gates as a module and instantiate it in multiple places throughout the program. For example, here is a circuit for a 4-bit adder made out of 1-bit adders:

```
1  module carry cin, a, b, cout {
2    toffoli cout, a, b;
3    cx b, a;
4    toffoli cout, cin, b;
5  };
6
7  module carry_inv cin, a, b, cout {
8    toffoli cout, cin, b;
9    cx b, a;
10   toffoli cout, a, b;
11 };
12
13 module sum cin, a, b {
14   cx b, a;
15   cx b, cin;
16 };
17
18 qubit cin0, cin1, cin2, cin3, cout;
19 qubit a0, a1, a2, a3;
20 qubit b0, b1, b2, b3, b4;
21
22 carry cin0, a0, b0, cin1;
23 carry cin1, a1, b1, cin2;
24 carry cin2, a2, b2, cin3;
25 carry cin3, a3, b3, b4;
26 cx b3, a3;
27 sum cin3, a3, b3;
28 carry cin2, a2, b2, cin3;
29 sum cin2, a2, b2;
30 carry cin1, a1, b1, cin2;
31 sum cin1, a1, b1;
32 carry cin0, a0, b0, cin1;
33 sum cin0, a0, b0;
```

Note that when calling modules, the register names must effectively be renamed so the physical element with the qubit state of $b3$ must be renamed b in the *sum* module.

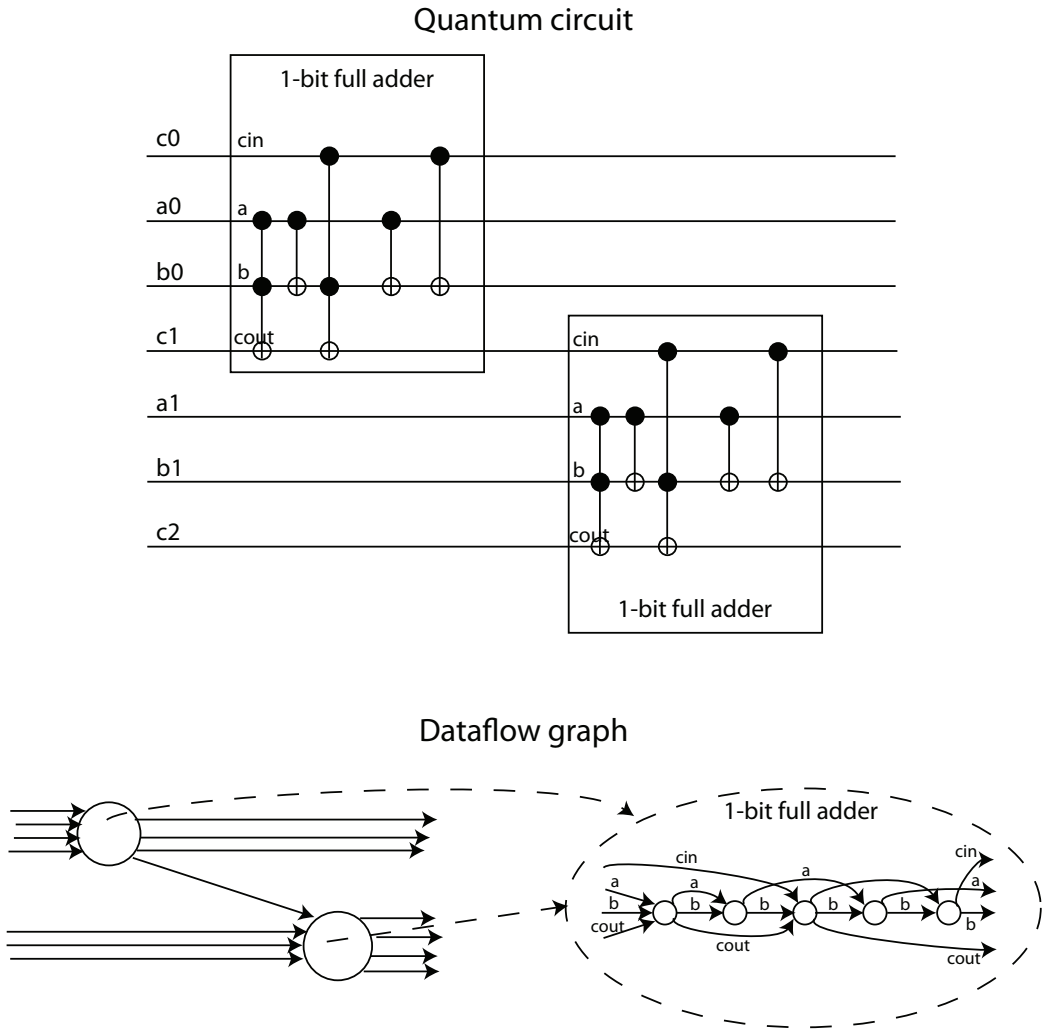


Figure 2.3: Gate networks are represented as linked, modular dataflow graphs. In this example, the top level graph consists of two nodes that each correspond to a 1-bit full adder. They both refer to the 1-bit full adder module dataflow graph.

We will discuss this issue later when we discuss tracking qubit error state.

Application Dataflow Graph

Our core data structure representing the input application logic is a hierarchical, annotated dataflow graph. Figure 2.3 shows an example of such a graph. In this example, the top level graph that consists of a 2-bit ripple carry adder is implemented with 2 nodes that both point to the same full adder graph. We have made improvements on the QASM

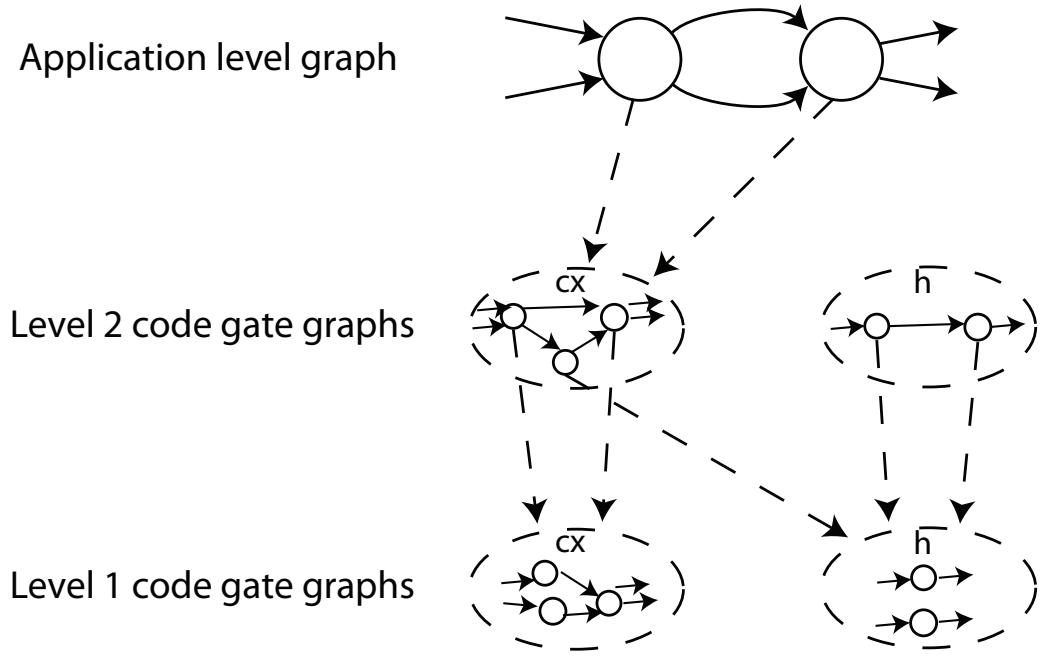


Figure 2.4: Hierarchical dataflow graphs are used to represented different levels of QEC encodings. In this example we have the 2 gate application circuit encoded in 2 levels of codes. Each code has a library of graphs, each graph implementing an encoded version of one gate type.

[8] programming language to support modular design so the user of our tool flow can write modular QASM and a modular dataflow graph will be used to represent it.

We continue to use the hierarchical nature of the dataflow graph all the way down the CAD flow. When we are synthesizing encoding gates for a QECed version of the application, each logical application gate is represented by a module pointing to a graph that represents a specific encoded version of that gate type. If we are concatenating several codes together to yield more reliability, there might be multiple levels in the hierarchy for gate implementations in different codes, as shown in Figure 2.4. The modular representation of a QECed circuit is especially beneficial since fault tolerant subcircuit substitution introduces orders of magnitude more gates (about 500x for a one level [[7,1,3]] code, for example). However, most of the subcircuits are the same so mapping all our logical gates of a particular type to a single graph for the FT version makes our design representation tractable for large circuits.

Method	*Description*
init(dataflowGraph, copy?)	Initialize iterator with dataflow graph and specify whether to keep a running copy of the graph.
boolean isEmpty()	Are there still any unprocessed nodes?
vertex getNext()	Get the next vertex in the hierarchical traversal
iterator getCurrentPosition()	Returns an iterator to the current position. This iterator can be used to add nodes at the current position of the dataflow graph copy that will be returned (must have been initialized with copy? flag set).
replace(vertex)	Replace the current vertex our iterator points to with a new vertex.
graph getGraphCopy()	Return copy of graph iterated through so far.

Table 2.2: Basic DataflowGraphIterator methods. All the basic operations available for graph traversal and modification through the iterator.

Additionally, we may have different elementary gates that can be performed physically depending on the implementing technology. We enable the technology-specific translation by providing technology gate libraries to translate logical level gates into physically implementable gates. Our technology translation currently converts single logical gates to groups of technology-dependent gates so we utilize the hierarchical nature of our dataflow graph again to maintain a modular mapping mechanism.

Note that even though only a single instance of a module is created and stored for a particular graph. When we traverse the graph, we must re-traverse the single module dataflow graph for all the nodes of a particular module type. This add some complexity to the traversal of our modular graphs.

Iterators The majority of all our transformations on the dataflow graph require us to traverse it in dataflow order. Since some of our operations on the dataflow graph are destructive or modify different levels in the hierarchy instead of just adding addition layers, we want our new graph to not modify the original but instead, create a new copy. We have implemented a flexible, iterator that can traverse our graph in dataflow order, hierarchically, optionally producing a copy in which nodes can be added or removed.

DataflowGraphIterator works by performing a combined breadth-first depth-first traversal starting at the top level dataflow graph. Within a particular level, the graph is traversed in dataflow order, i.e. breadth-first, but each node is not considered "finished" until all submodules implementing that node have been traversed (i.e. depth first). Table 2.2 outlines all the basic operations for DataflowGraphIterator. The two methods used to modify the resulting graph copy are getCurrentPosition and replace. Replace is used when we do not want the currently traversed vertex to appear in the graph copy. Replace is useful in creating another level of hierarchy; if a particular node is have a level of hierarchy added beneath it, we actually replace the original node with a new module node that points to the underlying graph. If we only want to augment the graph, getCurrentPosition is used instead. getCurrentPosition is most useful in adding additional gates at the same level of the dataflow graph. Right now we only add correction gates at the same level in the hierarchy.

The breadth first traversal at a given level in the graph is accomplished by keeping a token counter for each node. Whenever a node is completed, the token counters for all nodes that it outputs to are incremented. When a node's counter equals the number of inputs, it is appended to the ready queue for traversal. When a non-primitive node is traversed, the current ready queue and token counter structure is pushed onto a stack of module activation frames and a new frame is initialized for the subgraph. When we exhaust all the nodes in a particular frame's ready queue, that frame is popped off the stack and we mark the vertex corresponding to that subgraph as finished. An example of how the data structure is modified during traversal is shown in Figure 2.5.

We note that when copying the graph, the same structure of singly defined module graphs is maintained. This is essential to maintain a manageable size for our representations of the graph as we add more layers to the hierarchy.

2.2 Quantum Logic Synthesis

As mentioned in Section 1.4, the primary goal of logic synthesis in classical CAD flows is to derive a technology dependent gate network from a high level circuit specification. In addition to this goal, our quantum logic synthesizer also must add additional circuitry to ensure that our circuit is fault tolerant.

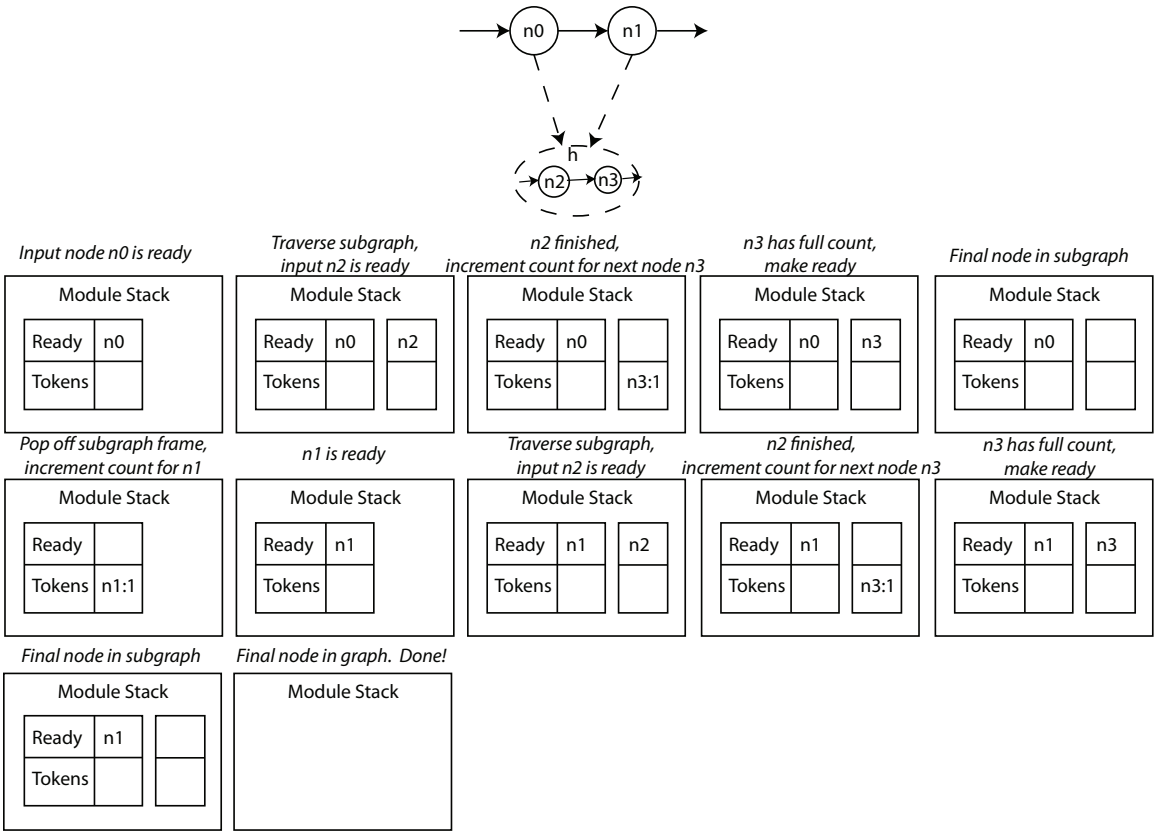


Figure 2.5: An example of a hierarchical traversal through a 2 level dataflow graph. The sequence of module stacks show changes in the DataflowGraphIterator over time as we traverse the hierarchical graph.

2.2.1 Technology Dependent Gates

Since we allow the superset of all interesting quantum gates from quantum computing literature to be used in our QASM definitions, we have a synthesis stage in which we convert QASM gate operations into gate operations that are supported natively by the type of quantum computing technology we are designing for. We specify *technology libraries* to map abstract QASM gates to physically implementable gates for each technology our CAD flow can target. For example, since we limit the number of qubits in an ion trap to 1 or 2 per interaction, we cannot physically implement a toffoli operation, so instead we translate toffolis into a sequence of 1 and 2 qubit gates from the ion trap technology library.

2.2.2 Fault Tolerant Gate Constructions

Once we have a set of physically implementable gates to work with, we must next make them fault tolerant. As shown in Section 1.2.2, we can apply quantum error correcting codes to the problem, transforming each logical gate from the technology-dependent network into an encoded subcircuit implementing the same operation fault tolerantly. For each code our CAD flow supports, we have a library of encoded gates that can be substituted into the circuit. These libraries are generated automatically using Andrew Cross's ftqctools [24].

The selection of QECC to be used in the synthesized circuit is current user-selected.

2.2.3 Random Circuit Generation

We have a number of real application benchmarks with which we will test various components of our tool flow, which are discussed in Sections 7.1.1 and 7.2.1. In addition to these, it is convenient to have benchmark circuits in which we can exert more control over various properties, such as the number of qubits, gates, or overall communication structure. For this reason, we also introduce a method for synthesizing random quantum circuits to test various portions of our tool flow. The generated random circuits have the following parameters:

Gate count Number of total gates that are in this circuit.

Gate type Types of gates included in the random circuit. Typically, we focus on the gates that appear most often in our applications, CNOT, Hadamard, and some non-transversal gate like T are common choices.

Qubit count Number of data qubits that are operated upon in the circuit.

Splitting fraction We will discuss a related concept, known as Rent's exponent in more detail in Section 3.1.1, but the basic idea is that it is an approximate measure of the locality of communication in a circuit. The splitting fraction tells us how to group gates when we are determining what gates should connect to each other when generating the circuit. A fraction of 0.5 will generate a circuit by successively breaking it into 2 equal sized parts and adding connections within each part, then recursively dividing each sub-part in half. A fraction of 0.9 will divide the circuit into one with 10% of the gates and another with 90% of the gates and follow the same recursive procedure.

2.3 Error Correction Circuit Optimization

In the previous section, we discussed the placement of error correction steps in a quantum circuit. The ratio of the number of gates present in an error correction step for a common 7-bit Steane code to the number of gates in an encoded CNOT gate is about 500/7. Thus, the majority of the gates being performed in any given circuit are for error correction instead of performing the actual computation. A few other works have addressed this apparent inefficiency:

Compressed Quantum Logic Array Thaker et. al. [98] proposed converting encoded qubits between different codes depending on the frequency of operations performed on it. This led to a memory-CPU structure, where qubits that were idle in memory were stored in a stronger code and qubits undergoing computation were stored in a mixture of the same strong code and a weak code. Their reasoning was that qubits in memory required fewer corrections since they were not subject to error prone gate operations so it was less expensive to store these qubits in this code in terms of gate count. Some qubits undergoing computation would then be switched to a more lightweight code to facilitate faster computation, since both the encoded gates and the correction steps would be faster. The authors did not investigate the opposite configuration: put the qubits undergoing computation in a stronger code because they are more prone to errors while performing gates and put the qubits in memory in a weaker code because their error rates are lower.

Ancilla Factories Our work in [48] focused on identifying the large, data-independent portion of a quantum error correcting step, the fault tolerant *ancilla generation*, in order to move this circuitry off the critical path. This ancilla generation was then aggregated in *ancilla factories* which then distributed ancilla to multiple error correction steps throughout a circuit. By batch processing error correction ancilla, we found we could drastically reduce the amount of resources necessary for a given computation.

This work will focus on a third type of fault tolerance optimization that could conceivably be used in conjunction with either or both optimization techniques mentioned above.

2.3.1 Retiming Based Optimization

If we follow the error model presented in Section 1.2, we have a well defined set of rules for how errors are generated and propagated through a circuit. We also note that for any non-trivial circuit, some qubits will undergo more gates, movement, of idling than others. Thus, different qubits will have different probabilities of error at different time throughout their life in the circuit. The previous conservative approaches to error correction call for the assumption that each logical qubit be corrected after every logical gate. Thus, it is effectively treating all qubits in the circuit as if they have the same probability of having an error at all times. This is not the case and therefore the treatment is overly conservative.

Our approach effectively analyzes each qubit at each gate and applies error corrections only when necessary. We draw an analog between minimizing latency in synchronous classical circuits and minimizing failures in our quantum circuits. We use the technique of circuit retiming [59] to "recorrect" the given circuit. Based on an approximation of how error propagate in a circuit, we can more effectively distribute error correction steps throughout our circuit.

2.4 Datapath Microarchitectures

The first step in specifying overall spatial placement of computation elements for a quantum circuit is to have a high level organization of the different quantum computer elements. For this, our work builds on several previous works that focused on *tiled-dataflow architectures* for a quantum computer.

Proposed architectures, including ours, have consisted of computation regions connected by an interconnection network using quantum teleportation [64, 47]. High fault rates in quantum computing necessitate the widespread use of quantum error correction (QEC). Further, ancilla state generation is important to aid in the correction process [89] and as an integral part of quantum algorithms, as we showed in [48].

2.4.1 Three Major Organizations

Figure 2.6 shows three major *datapath organizations* that represent the "state of the art" in quantum computing[1]. They are QLA [64], CQLA [98], and our work,

[1]Since circuits are *mapped* to these datapaths, they are not quite "architectures" but rather raw material for constructing architectures.

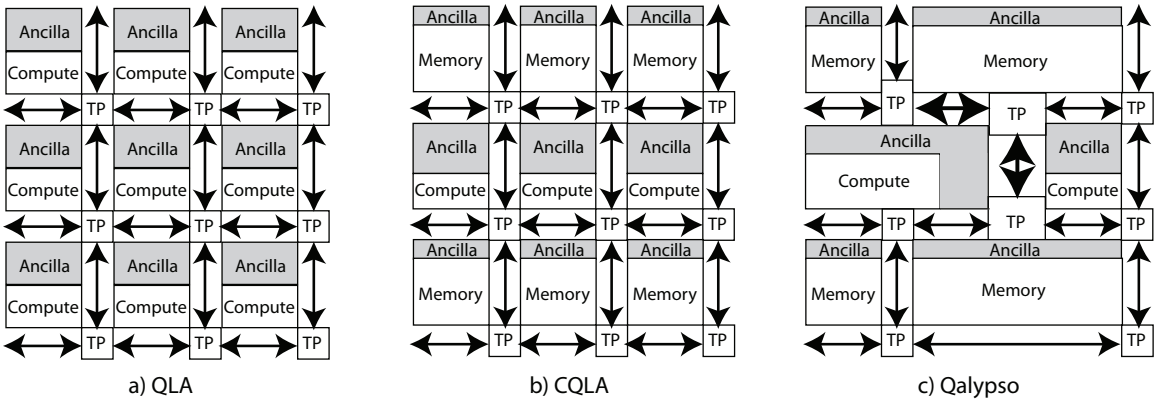


Figure 2.6: Quantum Datapath Organizations: a) Quantum Logic Array (*QLA*): An FPGA-style sea of quantum two-bit gates (compute tiles), where each gate has dedicated ancilla resources. b) Compressed QLA (*CQLA*): QLA compute tiles surrounded by denser memory tiles. c) *Qalypso*: Variable sized compute and memory tiles with shared ancilla resources for each tile; teleportation network can have variable bandwidth links.

Qalypso [48], and can be viewed as a spectrum from inflexible to flexible ancilla distribution. They differ in their configuration of compute regions, ancilla generation areas, memory regions (for idle qubits), and teleportation network resources (for longer-distance communication)[47, 64].

The QLA architecture is most like a classical FPGA, in that all elements are identical: each element contains enough resources to perform a two-bit quantum gate. Each such *compute region* contains dedicated ancilla generation resources, space for two encoded quantum bits, and a dedicated teleportation router for communication.

CQLA improves upon QLA by allowing two different types of data regions: *compute regions* (identical to those in QLA) and *memory regions* (which store eight quantum bits) [98]. To account for different failure modes (idle errors vs interaction errors), data in memory regions are encoded differently from data in compute regions.

Finally, Qalypso improves upon CQLA by further relaxing the strict assignment of ancilla generation resources. It allows optimized, pipelined ancilla generators to feed regions of data bits (compute regions) that can perform more than just two-bit gates. The sizing of ancilla generators and data regions can be customized based on circuit requirements. Qalypso requires analysis (Section 2.4.3) to balance ancilla consumption with ancilla gen-

Datapath	Description
QLA	Original Quantum Logic Array [64], compute regions only, no specialization
LQLA	QLA with an optimized ancilla generator from [56]
CQLA	Compressed QLA [98], compute and memory regions specialization, original ancilla generator
CQLA+	CQLA with a better performing ancilla generator from [90]
Qalypso	Our architecture [48]. Variable sized compute and memory regions, variable resources in ancilla generators and teleport network. "Pipelined" ancilla factory optimized from design in [90].

Table 2.3: Taxonomy of the quantum computer datapath organizations from the literature and our work.

eration. Such analysis can automatically adjust the amount of ancilla bandwidth required in memory regions based on the residency time of qubits.

In all three organizations, each compute or memory region is placed adjacent to a teleport router. Qubits are moved ballistically within regions and teleported between regions.

Proper design of the datapath elements (such as teleportation routers or ancilla generators) is an important factor. Our previous work on ancilla generators from [48] is detailed in 3.2.4. In all these microarchitectures, we also pay careful attention to the teleportation network [47, 64]. We have produced layouts for the routers and EPR generators and utilize these in computing area, latency, and error probability of circuits. Sections 4.9 and 3.2.4 discuss how these numbers are derived and integrated with our evaluation methodology.

2.4.2 An Organizational Zoo

Table 2.3 shows the datapath organizations that we will use in this paper. Three of these, namely QLA, CQLA, and Qalypso come directly from their original papers. LQLA is a variant of QLA utilizing the new ancilla generator and cell layout proposed by Kreger-Stickles. Finally, CQLA+ is a version of CQLA utilizing an improved ancilla generator

from our work in [48].

Evaluating such a disparate set of architectures is always challenging. When possible, we have adapted the exact qubit scheduling provided by authors (such as in LQLA, where the authors provided us with scheduling of qubits for their ancilla generator). To evaluate larger circuits, we have developed a hybrid evaluation methodology (described in Section 4.9) that permits us to stitch together modules.

Out of the prior work architectures, QLA, CQLA, CQLA+, and LQLA, we single out LQLA as the most logical prior work to focus our analyses on. LQLA is our invention, produced by inserting the optimal ancilla factory from [56] into QLA; this was necessary because the authors of [56] did not take a stand on long-distance communication or memory regions. It is specially designed for minimizing sources of faults by careful design of QEC modules. Providing good fault tolerance properties is in line with our goals for all our techniques described here.

2.4.3 Coarse-Grained Mapping and Floorplanning

In light of these tiles datapaths, we must have a method to map groups of circuit elements from our application onto compute regions. The heuristics used for this mapping are not the focus of this work but are briefly explained in Section 3.4.1. Once we have this mapping, we can apply macroblock-level layout heuristics we will introduce in the next section in order to get detailed layouts. As mentioned above, we have a hybrid error simulation model, described in Sections 2.6 and 4.9, for unifying the error model for this piecewise layout.

2.5 Ion Trap Layout

We abstract away much of the ion trap intricacies of laser positioning and electrode shape and spacing by building our ion trap layouts from a set of segments we call *macroblocks*, shown in Figure 2.7. The idea is that while many of the details of a ion trap will change, we will still most likely have designated positions for gates, and channels for moving qubits around with 90 degree turns. Since our heuristics use this abstracted view of communication and gate blocks, we could use these techniques on other technologies as well.

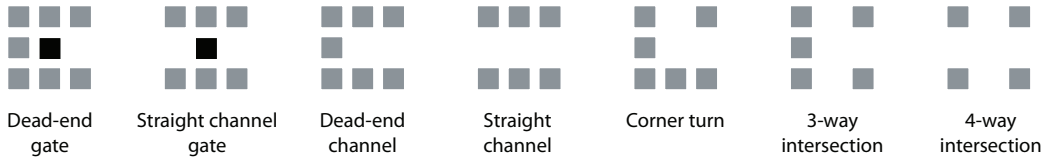


Figure 2.7: The basic building blocks of our ion trap layouts. Each *macroblock* consists of 3x3 electrodes or spaces to provide functionality as a straight channel, a gate, a turn, or an intersection.

The next step is to take our optimized circuit and create a layout of physical components. Not only does this give us a design that can then be fabricated, but it also allows us to create a precise schedule of qubit movement and qubit idling in addition to gates. The schedule of all the qubits' operations can then be extracted to an error model that accounts for all error sources.

Our layout generation flow is broken into two components: a coarse-grained mapping of high-level modules from the original circuit specification to regions on a substrate and then fine-grained layout of actual computation elements within these regions.

2.5.1 Fine-Grained Element Place and Route

Using a library of basic layout building blocks that we call *macroblocks*, we piece together the physical level detailed layout. An example of the macroblocks we use for a layout in ion trap technology is shown in Figure 2.7. The macroblocks abstract away some of the most gory details of the underlying implementation but are easy to translate to the physical technology.

We investigate a number of different heuristics for our fine-grained place and route tool but the most successful combines an adaptation of the classical FPGA tool VPR [13] and our own dataflow graph style layout. We will discuss these heuristics in more detail in Section 3.2.

2.5.2 Layout Graph Representation

Our layouts are represented by a layout graph which contains macroblock nodes (as mentioned in Section 2.5) that are linked together with QNets. The QNets hold information on how connected macroblocks are oriented with respect to one another. Figure 2.8 shows an

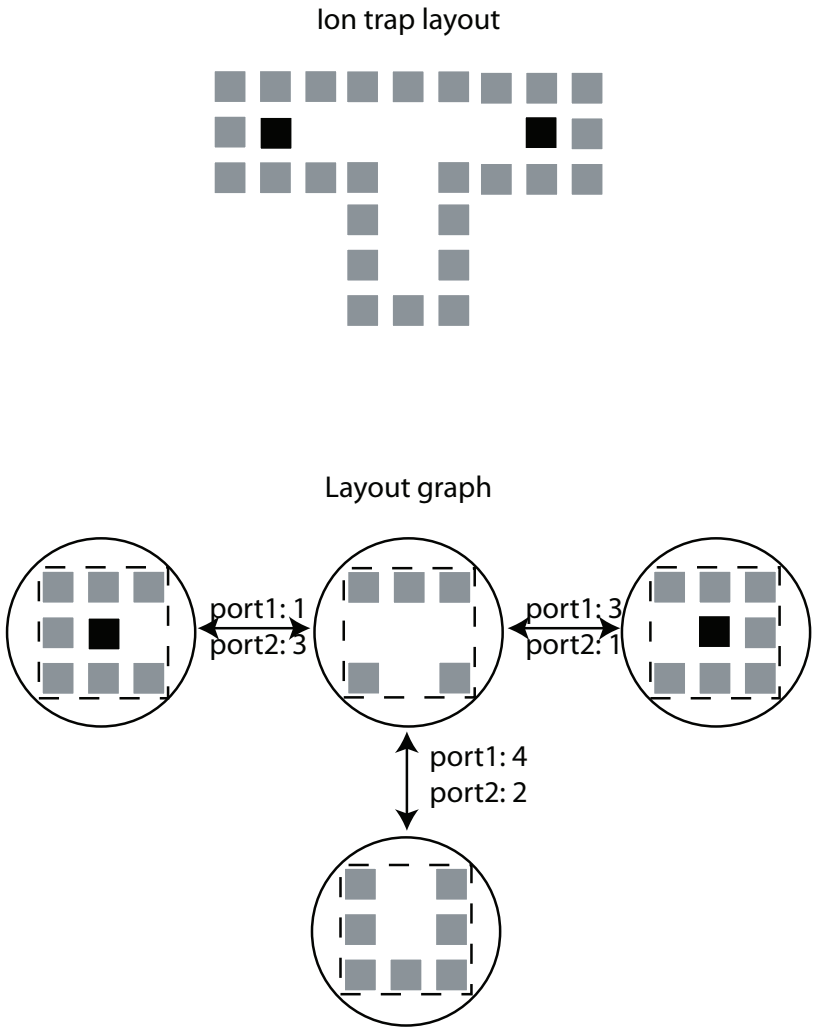


Figure 2.8: A layout and its associated graph. The nodes correspond to macroblocks and the edges correspond to "qnets" which do not have any associated physical entity but determine how macroblocks are oriented with respect to each other.

example of a layout graph structure. Macroblock nodes specify their location and orientation on the substrate. They also contain additional information to be used by the scheduler to track ion movement through the macroblocks.

Layout graphs can have a similar modular structure as our dataflow graphs have. An abstract layout module can refer to a single macroblock or another layout graph. The embedding of a complex layout module is not as simple as in the dataflow case since the sublayout must be spatially fit into the higher level design, but layout modularity again gives us considerable savings in representing the full layout since many structures are often repeated. Some examples of repeated sublayouts are teleportation routers and ancilla factories.

Layout Specification

Our layout specification consists of a sequence of layout module instances, all parameterized by location and a rotation angle, in an XML format. At the lowest level, everything is made up of macroblocks for the underlying technology, like those shown in Figure 2.7. Additionally, we can define higher level modules, made out of macroblocks, which can then have instances placed in the layout. Higher level modules must define ports where they connect up to adjacent modules so that the qubit movement scheduler can track movements across module boundaries. Figure 2.9 show an example of such a modular layout.

2.5.3 Layout Metrics

Similar to the case of classical CMOS metrics for layouts mentioned in Section 1.4.5, we are also interested in area and delay for layouts of quantum circuits as well. Design area is an important consideration especially in light of the fact that previous work has estimated a design for Shor's factorization of a 1024-bit number to be $0.9m^2$ in area. Since the technology under consideration uses a silicon substrate, fabrication considerations dictate more area-efficiency in our designs.

Delay is another important consideration, since quantum computers promise to perform certain tasks asymptotically faster than classical computers. It is important that the constants involved do not swallow up asymptotic gains.

```
<define_module>
  <type>horseshoe</type>
  <module>
    <type>straight_channel</type>
    <location>0,0</location>
    <rotation>0</rotation>
  </module>
  <module>
    <type>turn</type>
    <location>30,0</location>
    <rotation>0</rotation>
  </module>
  <module>
    <type>turn</type>
    <location>30,30</location>
    <rotation>90</rotation>
  </module>
  <module>
    <type>straight_channel</type>
    <location>0,30</location>
    <rotation>0</rotation>
  </module>
</define_module>

<module>
  <type>horseshoe</type>
  <location>0,0</location>
  <rotation>0</rotation>
</module>
<module>
  <type>horseshoe</type>
  <location>60,0</location>
  <rotation>180</rotation>
</module>
```

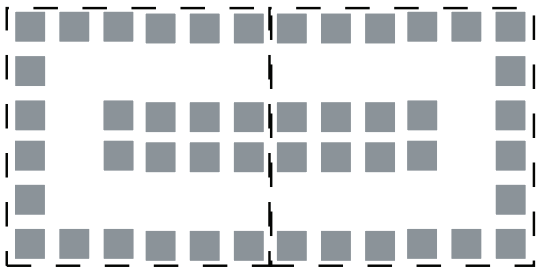


Figure 2.9: Layouts can consist of placements of single macroblocks or definition and then instantiation of larger layout blocks. In this example, we define a larger "horseshoe" block made up of macroblocks and then instantiate two of them in different positions and orientations.

2.6 Fault Tolerance Verification

The goal of the fault tolerance verification tool is to determine the probability of an unrecoverable error on the qubits that would yield an incorrect answer to our computation. Furthermore, we would like to know at which points in the design are data most likely to incur errors.

2.6.1 Determining Failure Probability

We will discuss several methods for computing the probability of failure of a circuit. A circuit is considered to have failed if:

- The circuit is not encoded in an error correction code and one of the output data qubits incurs an error.
- The circuit is encoded and an encoded output qubit incurs more errors than the code can correct.

We can either track errors at the circuit level, accounting only for gate errors or at the layout level, accounting for gate, idle and movement errors.

2.6.2 Hybrid Fault Point Extraction

A key part of our tool is the extraction of a set of possible failure points in a circuit or layout that can then be processed by some estimator of overall failure probability. As mentioned in Section 2.4 and 2.5, the layouts we produce are not completely detailed macroblock-level descriptions of the entire layout. Instead we have a *piecewise-detailed layout* from which we must extract all the gate, movement and idle errors, which we detail in Section 4.9. We can also opt to extract only a subset of the possible error events and simulate those.

2.6.3 Accuracy-Time Tradeoffs

We are forced to trade off accuracy and time in our probability estimations. The full simulation of a quantum mechanical system is exponential in the number of qubits simulated due to the need to track all linear superpositions of bit strings. Therefore, performing a brute force simulation of all possible physical processes, even if they could all be

modeled, is prohibitively expensive. We instead must rely on simpler models of quantum errors and simulate only the error incidence and propagation that impacts the overall failure probability the most. We will now give an overview of the different approximations we will use in our fault tolerance verifier.

Once approach to partitioning the problem is to simulate various levels of detail:

Architecture level Abstracting an error model from a high level quantum computer architecture is one way of getting a rough idea of resource usage and reliability. In Section 4.1, we track the fidelity of a block of qubits between functional units, over a teleportation-based communication network. In this simulation, we are essentially scaling the failure probability of logical qubits in the computer as a function of distance and teleportation operations.

Circuit level We simulate error propagation through gate networks using CHP rules mentioned in Section 1.2. Since gate errors are typically the most probable, this gives us a quick estimate of the reliability of a circuit without needing to run the place and route tool. It also involves the tracking of less errors total since we are not simulating movement and idle errors.

Layout level Movement and qubit idle errors are extracted from the layout according to models we develop in Section 4.2. Movement errors are parameterized to have different failure probabilities and Pauli error types based on different geometrical moves. Idle errors are parameterized based on the time the qubit idles for.

Secondly, we can attack problems at the same level of detail with different methods:

Monte Carlo methods Performing Monte Carlo sampling of errors in quantum circuits gives a simple way to simulate error occurrence. Monte Carlo methods can be costly since we need to perform large numbers of simulation in order to extrapolate accurate error rates. This technique can be used to track errors on any of the three design levels, architectural, circuit, and layout, and is discussed in Section 4.6. However, the simulation time depends linearly on the number of discrete error events, which make the layout level simulation costly for large designs. We present a technique to amortize out some of the overhead from the basic Monte Carlo simulation in Section 4.7.

Direct Probability Calculation Monte Carlo methods seem somewhat wasteful in that they throw away all the error probability information in favor of a single sample. We explore options of directly calculating the error probabilities of all possible combinations of errors on entangled sets of qubits. It turns out that a naive calculation of these probabilities requires exponential resources to track all error vectors but we also investigate approximation methods to determine whether this exponential can be managed. These computation options are discussed in Section 4.5.

2.6.4 Fault Tolerance Metrics

Due to the very high prevalence of errors in quantum circuits, overall probability of success will be one of the most important metrics in the foreseeable future. While reliability is only beginning to become important in classical CMOS circuits, it must be addressed as an integral design parameter from the very beginning for quantum circuits. This is why we focus on evaluation of this metric to drive many of our design decisions.

Since the proposed applications for a quantum computer are all currently in the complexity class NP, we can verify whether the answer produced is correct or not fairly easily with a classical computer. This means that in we have data corrupting errors in a run of our computation, we can just run it over again until we get a correct answer. For this reason, probability of success is closely connected to the layout delay metric. We will revisit this when we talk about our aggregate metric, ADCR in Section 2.7.

2.6.5 Feedback for Further Optimization

Our QEC optimization uses feedback with the failure probability estimator to determine whether it has produced a fault tolerantly functional design. This requires the estimator to provide feedback on how well an optimized design has performed compared to the original, unoptimized one.

In addition to computing the output failure probability, we would like to compute failure probabilities of qubits throughout the circuit to allow targeted optimization of the error correction routines we insert. For example, our initial pass of error correction insertion only looks at the gate-level network. We discuss this limitation in Section 6.3.7 and point out possible future work could be to use feedback on movement and idle errors. We could then place error correction steps after long moves or idles as well as long sequences of gates.

2.7 ADCR: An Aggregate Metric for Probabilistic Computation

As mentioned earlier, delay and success probability are closely connected in the evaluation of any quantum circuit, this is because we are not simply interested in a single run of a circuit on a layout if it does not produce the correct answer. We would instead like to know the expected time to get a correct answer:

$$E(Delay) = Delay_{singlerun} \times E(runs\ for\ correct\ result) \tag{2.1}$$

$$= Delay_{singlerun} \times \sum_{n=1}^{\infty} \frac{n}{P_{success}(1 - P_{success})^{n-1}} \tag{2.2}$$

$$= Delay_{single_run} \times \frac{1}{P_{success}} \tag{2.3}$$

Prior work has focused on maximizing the overall success probability $P_{success}$ at all costs. This might be a suitable sacrifice if we are always on the verge of a catastrophic decline in success probability for a design (probably the case in all current laboratory setups). As the technology matures, it will be more important to evaluate all the design considerations; if we can reduce a layout's delay by 10x with only a 10% reduction in success probability, this is probably a good trade-off. Critical to this evaluation is a comprehensive evaluation of the overall probability of success of a design. If we overestimate this probability, we could end up making trade-offs to get a design that does not work at all. We will investigate this trade-off in detail later in Chapter 6, when we talk about optimization to reduce QEC overhead.

To evaluate the quality of quantum layouts with a single number, we propose a *composite* metric called *Area-Delay-to-Correct-Result (ADCR)*. ADCR is the probabilistic equivalent of the Area-Delay product from classical circuit evaluations:

$$\text{ADCR} = \text{Area} \times E(\text{Latency}_{\text{total}}) \tag{2.4}$$

$$= \text{Area} \times \sum_{n=1}^{\infty} n \cdot \text{Latency}_{\text{single}} \cdot P_{\text{success}}(1 - P_{\text{success}})^{n-1} \tag{2.5}$$

$$= \text{Area} \times \frac{\text{Latency}_{\text{single}}}{P_{\text{success}}} \tag{2.6}$$

For ADCR, *lower is better*. By incorporating potential for circuit failure, ADCR provides a useful metric to evaluate the area efficiency of probabilistic circuits. It highlights, for

instance, layouts that use less area for the same latency and success probability. Or, layouts that use the same area for lower latency or higher success probability.

2.7.1 ADCR-optimal

With the definition of ADCR, we can now talk about designs that are *ADCR-optimal*, or the set of design parameters the yields the lowest ADCR. Since ADCR is meant to be a comprehensive metric for all the stages in our CAD flow, finding the design with the best ADCR takes some amount of iteration and feedback.

In order to get an ADCR-optimal layout for an application circuit we must search over these parameters:

Error correcting code selection Different codes impact area, latency and success probability, due to encoder size, complexity and errors corrected. Thus we must iterate through different encodings in the QEC synthesis phase, optimizing, laying out the encoded circuit, and simulate for failures.

QEC optimization level We will see later in Chapter 6 that we can tune the degree of QEC optimization in order to trade-off success probability for area and/or latency. Thus, we must iterate over different optimization settings, lay out the resulting circuit, and simulate for failures to find a design with a good ADCR value.

Tiled datapath configuration We will talk more about the possible datapath parameter settings in Section 3.4.1, but datapath type, as well as, varying the number and size of compute and memory regions impact area, delay and success probability. Thus we must iterate through different designs in the datapath space, then lay out and simulate for failures.

All these choices gives us a multi-parameter optimization problem. We currently binary search through many of these parameter spaces to find ADCR-optimal designs, but future work includes using better heuristics to narrow the set of candidate parameter settings to converge on a good design faster. We will investigate ADCR sensitivity to these various parameters throughout this work and discuss the multi-parameter optimization search again in Section 7.3.

We will now talk about each tool in more detail, discussing the algorithms and some micro-benchmark results for each. The first topic is placement and routing tools.

Chapter 3

Communication in Quantum Circuits

We are interested in creating a complete design for a quantum circuit. This means we must have a mapping from the input application circuit to a physical substrate. Additionally, in order to perform a complete analysis of all the errors in our physical design, we must account for not only gate errors but movement and idle errors as well. In this Chapter, we will look at different ways to model communication to derive both movement and idle errors. Generating a layout accomplishes both goals, we get a physical design we can then fabricate, and we can extract a complete set of errors, which we will describe later in Section 4.9.

We would also like to use this complete error specification to make some observations on the intrinsic properties of quantum error correction circuits. The drawback to using layouts to do this is that we must choose heuristics for mapping our circuit elements to a substrate. The resulting error estimates from these mappings might not necessarily reflect intrinsic properties of the circuits but instead peculiarities of our layout heuristics. In order to partially mitigate this problem, we develop an abstract communication model based only on circuit structure. We then use this layout-independent method of communication estimation to validate that the layouts are good.

The application of error correction codes come at the cost of increased design size and complexity. In fact, much of the work done in quantum fault tolerance has been either to identify new codes [4, 16, 85], or to prove the tractability of the circuit with the added

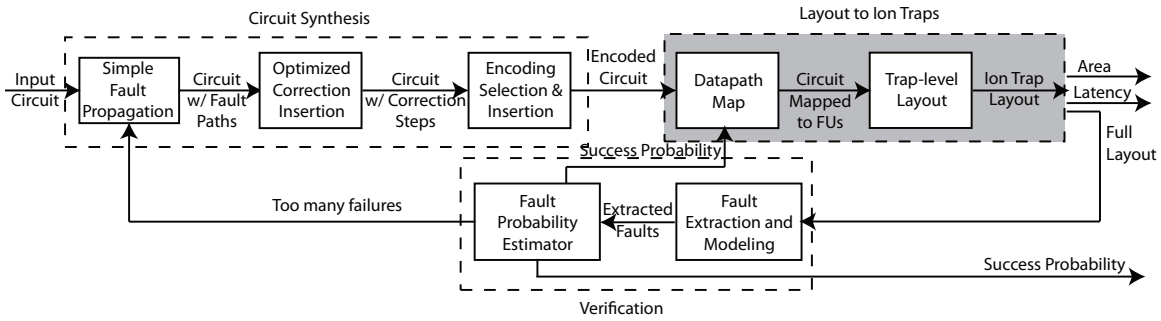


Figure 3.1: Layout and communication estimation portion of CAD flow.

overhead [2]. Little work has been done on fully characterizing the impact of all error sources on a larger fault tolerant design. Past complete error model studies have focused on detailed studies of a single encoded gate with a correction stage appended to it for a single code [37, 97, 95]. Other studies have compared various codes on single encoded gates [25] or larger circuits [90], but have not included all error sources.

Our work is the first to provide general methods for analyzing the impact of communication overhead on error thresholds of QECC-encoded circuits. In order to quantify communication effects, we will introduce various components of our CAD flow (Figure 3.1 that estimate wire length or synthesize full layouts of circuits. Communication overhead in the form of qubit movement error has been demonstrated as a real error source in experimental work [45] and therefore ignoring it gives us only an incomplete picture of the performance of particular QECCs.

As a preliminary example of the impact of movement error on QEC performance, Figure 3.2 shows the overall failure probability of a simple circuit encoded with a 23 bit Golay code [77] and a 25 bit Bacon-Shor code [4]. Previous studies have indicated that the 23 bit Golay code performs "well" in that it is estimated to have a high threshold compared to many other codes [90, 25]. In Figure 3.2, on the far left, we see that the 23 bit code has a better failure probability than the 25 bit code. However, as we introduce stronger qubit movement errors from left to right, the 25 bit code ends up doing considerably better than the 23 bit code. This chapter will detail how we account for this movement error, Chapter 4 explains how we actually derive the final failure probability, and Chapter 5 gives results on a comprehensive code comparison using these tools. The central point is that as we further refine our estimates on QECC performance, we must take into account not only the

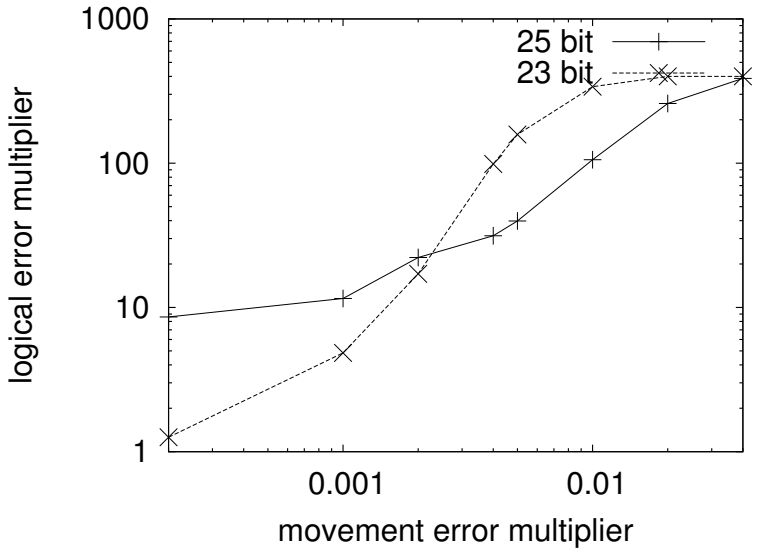


Figure 3.2: Comparing the overall failure probability of an encoded CNOT circuit as a function of communication error for two different QECCs. All numbers are multiples of a base error probability of $\gamma = 0.0025$ (pseudo-threshold of this 23 bit code implementation). Each point on this graph is a failure probability simulation given a particular gate and movement error probability. In this example, the gate error rate is set to γ and the movement error rate is varied over the x axis is in multiples of γ. The "logical error multiplier" is also a multiple of γ and refers to the probability that our data is corrupted (lower is better). Note that a multiplier of 400 is equivalent to complete failure (probability 1). Note that we reach this total failure point before movement error rate becomes equal to gate error rate (multiplier 1 on the x axis).

overhead from gate errors but also from qubit movement.

In our work, we leverage circuit analysis and synthesis techniques from classical computer aided design to make explicit the necessary qubit communication in a given gate network. Using our derived communication estimate or schedule, we can apply a movement error model, combine this with a gate error model, and estimate the overall failure probability for a given quantum circuit. We introduce our adaptation of *Rent's rule* from classical circuit connectivity analysis [58] and the related Donath's technique for wire length estimation [28]. These distances give us an idea of how much communication overhead that qubits are likely to incur. We also build upon our previous results in quantum circuit layout

[101] to produce physical level designs in segmented ion trap technology [51] which allow us to get an exact schedule of qubit movement for a circuit. We compare the average communication distance between our scheduled layout and Donath's estimation and find them in agreement for a large number of circuits. This agreement gives us confidence that these communication patterns are intrinsic to the circuit itself and not artifacts of our layout heuristics.

Both the estimates and exact schedules of communication are then integrated into our Monte Carlo simulation of an entire quantum gate and communication network. We use these Monte Carlo simulations to evaluate the effectiveness of a variety of QECCs in the face of gate and qubit movement error. Our two main techniques for estimating communication in a gate network are complimentary to each other. Donath's wire length estimation is analytical and gives a rapid estimate whereas our layout and scheduling tool flow leads to a constructive estimate from a fully functional design.

3.1 Analytical Estimation of Communication

We are interested in quantifying the communication overhead for a given QECC encoded quantum circuit. We would first like to get a high-level estimate of the intrinsic communication complexity due to the circuit topology, agnostic of the physical technology used in implementation. Classical circuit analysis has had the same interest in interconnection complexity, although for different reasons like communication latency and power. We introduce two related techniques for estimating overall communication complexity and communication distances for a given circuit. We will later use these techniques to validate the quality of the layouts we produce in the second half of this chapter.

3.1.1 Rent's Rule

Work done by E. F. Rent identified a common trend among circuit designs he studied in which the number of input/output connections T for a design could be fit to a power law relation $T = tg^p$, where g is the number of gates in the circuit. t is a constant equal to the average number of inputs and outputs per gate, and p is referred to as "Rent's exponent". It was later shown by Landman and Russo [58] that in general, if one hierarchically partitions a circuit into smaller and smaller blocks of gates, the power-law relationship generally holds for the subsets of gates in the partitions.

Rent's rule has been shown to apply to large numbers of classical circuit designs since it was first formulated, and the Rent's exponent for a circuit has been established as a measure of the complexity of that circuit's interconnect topology [71]. Since the quantum circuit model is modeled after classical circuits and both consist of logic elements connected with some sort of interconnection network, we have developed a tool to compute Rent's exponent for a given quantum circuit.

The algorithm we use to derive a circuit's Rent's exponent is given in Algorithm 1. The algorithm works as follows, we construct a dataflow graph out of a given circuit, with G gates. We go through and repeatedly partition the dataflow graph in successively more and more partitions. For each partition, we note the number of connections crossing the each partition boundary (c) and store this based on the number of gates in that partition (g). C is the array that stores this list of inter-partition connections for each gate count. Once we have all these counts, we compute the average number of boundary crossings within each gate count g list, and store these averages in T (indexed by g). Finally, we fit a power law function to T and g. The parameter r is the Rent's parameter.

Algorithm 1 Rent's exponent calculation

Extract circuit into netlist of gate nodes and connections
$T \Leftarrow 0$
for $n = 2$ to G **do**
  Generate partitioning P of netlist into n partitions
  $g[i] \Leftarrow$ number of gates in partition i
  $c[i] \Leftarrow$ number of netlist connections exiting partition i
  for $i = 1$ to n **do**
    Append $c[i]$ to $C[g[i]]$
  end for
end for
for Each entry S in C **do**
  $T[g[S]] \Leftarrow$ average over all c in S
end for
Fit $T = tg^r$ using nonlinear least-squares (solving for t and r)

Figure 3.3 shows this same process in graphical form.

To produce a "good" partitioning P of the circuit netlist, we use the METIS

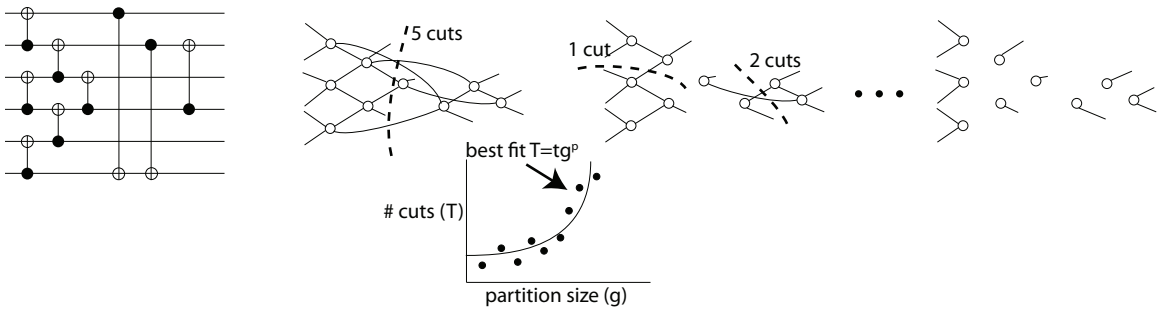


Figure 3.3: Computation of Rent's exponent. First, transform the circuit into a dependency graph, each node representing a gate and each edge representing wires between gates. Next, we recursively partition the graph and count the number of edges crossing the partition boundary each step. Finally, we plot the number of edges cut as a function of partition size and fit a power-law function to it.

graph partitioning library [50] to generate partitions that are roughly the same size and still minimize the number of communication links cut between partitions. While this is an NP complete problem, this library produces reasonably good results in practice. The nonlinear least-squares fit is done using gnuplot [102] which implements the Marquardt-Levenberg algorithm [68].

Rent Exponent Results

Figure 3.4 shows the Rent exponents for the codes listed in Table 3.1. The exact circuit used in this analysis was a single encoded CNOT circuit, with a correction step on each qubit before and after the gate (shown in Figure 3.19). This captures the associated communication complexity of the correction step as well as the encoded gate. It is also the circuit chosen as the test input for a number of other fault tolerant evaluations, such as [95, 25, 5, 97].

These exponents were computed by the method outlined in Section 3.1.1. We see that there is a clear bimodal distribution of Rent exponent values. The bottom codes are all the Bacon-Shor codes and the top codes are the rest that we analyzed. To review, a smaller Rent exponent corresponds to a less connected gate network, giving us an intuitive metric for communication structure simplicity. This result matches previous work showing that Bacon-Shor codes are indeed very simple in structure, compared to many other quantum

$[[n, k, d]]$	Code description	Rent exponent r
$[[7, 1, 3]]$	Steane code	0.45
$[[9, 1, 3]]$	Shor code	0.42
$[[13, 1, 3]]$	Surface code:3	0.58
$[[15, 1, 3]]$	Reed-Muller code	0.59
$[[23, 1, 7]]$	Golay code	0.62
$[[25, 1, 5]]$	Bacon-Shor code:5	0.44
$[[41, 1, 5]]$	Surface code:5	0.63
$[[47, 1, 11]]$	Quadratic-residue code	0.64
$[[49, 1, 7]]$	Bacon-Shor code:7	0.47
$[[49, 1, 9]]$	Concatenated Steane code	0.64
$[[81, 1, 9]]$	Bacon-Shor code:9	0.48
$[[121, 1, 11]]$	Bacon-Shor code:11	0.49

Table 3.1: List of quantum error correcting codes we compare in this study and their Rent exponents computed by Algorithm 1.

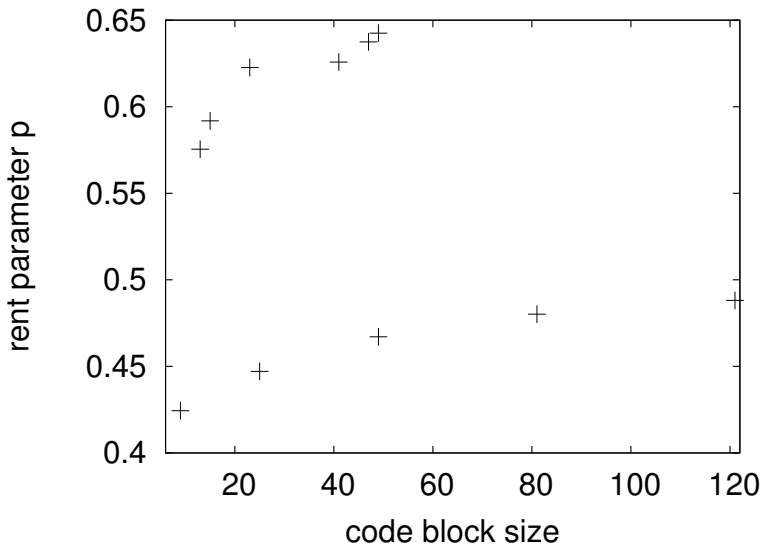


Figure 3.4: Rent exponent for a number of quantum error correcting codes. There was only one family of codes represented by each block size. See Table 3.1 for a list of the codes represented. The line of codes with the lowest Rent exponent are the Bacon-Shor codes.

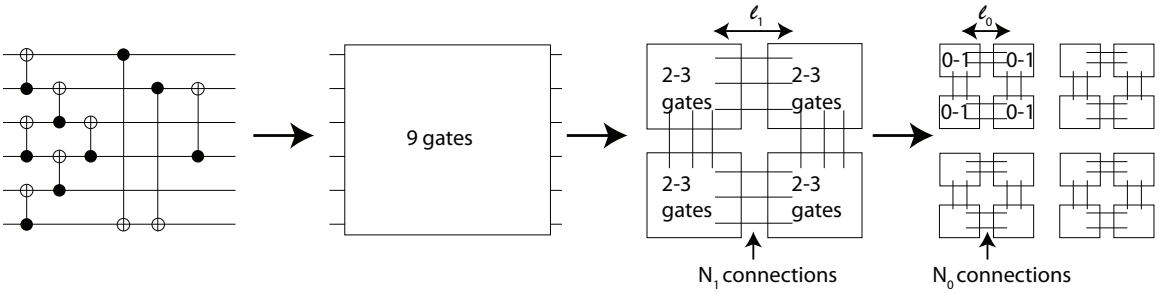


Figure 3.5: Computing the average wire length with Donath's technique. After computing Rent's exponent, we do the following: count the number of gates in the circuit, recursively divide that number into fourths, use Rent's rule to estimate the number of connections crossing fourths at each level and estimate a wire length for those crossings, stop when we have one gate per partition.

codes [4].

We have already mentioned randomly generating circuits with a given clustered structure in Section 2.2.3. The above approach is analytical whereas our random circuit generation is a constructive process, but the Rent exponent and our splitting factor parameter are still positively correlated.

3.1.2 Donath's Wire Length Estimation

Rent's exponent gives us a unitless, relative measure of communication complexity in a circuit. This communication complexity is nice for comparisons between circuits but we could really like a more concrete estimation of the amount of communication taking place in a circuit. Donath's wire length estimation is an established technique from classical CAD work that gives us just such an estimate [28]. We will use this estimate to validate the quality of our own layouts by comparing average "wire length" or communication distance for encoded, laid out circuits in Section 3.3.2.

Donath's estimate for wire lengths on a 2-dimensional substrate uses a recursive quartering of the plane ($K = \log_4 G$ recursions), computes the number of connections crossing quarter boundaries (N_k), and averages the distance between gates in different partitions at each level of recursion ($\bar{\ell}_k$). Higher levels in the recursive partitioning (fewer, larger partitions) correspond to large average distances, $\bar{ell}_k$. Figure 3.5 shows this partitioning and

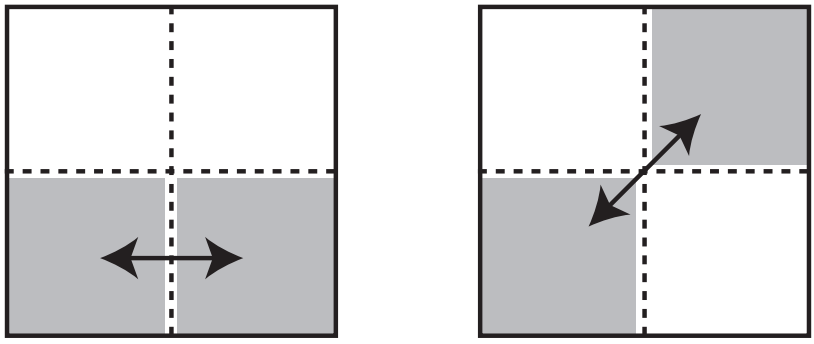


Figure 3.6: The two possible inter-partition communication scenarios for a spatially quartered circuit layout.

counting process.

$$\bar{\ell} = \frac{\sum_{k=0}^{K-1} N_k \bar{\ell}_k}{\sum_{k=0}^{K-1} N_k} \tag{3.1}$$

Equation (3.1) shows the main computation we must do to get the total average wire length, $\bar{\ell}$. Given a recursive partitioning P, k counts through each level. At each level k, we have N_k, which is the number of connections crossing partitions at this level:

$$N_k = 4^k N_k^0 - N_{k-1} \tag{3.2}$$

N_k^0 is the number of connections crossed for one partition at level k and 4^k is the number of partitions at that level. Last, we must subtract out the number of connections from the previous level of partitioning. One way to compute N_k^0 is to use result of the Rent exponent fitting from Algorithm 1, $N_k^0 = tg_k^r$. Thus, we would first compute r and t, then plug them back into this formula to get the number of connections per level. This technique is typically used when the calculation of r is done in isolation of $\bar{\ell}$. If we are doing the partitioning P to compute r just to be used in Donath's estimate, we can instead directly take the averages, $T[g_k]$, for each level for a more accurate estimate of N_k^0, which is our approach.

To compute $\bar{\ell}_k$, we use a stochastic approximation of the distribution of source and destination gates for links crossing the boundaries of the 4 partitions at each of the recursive levels. More details can be found in [92, 93], but we reproduce the result here:

$$\bar{\ell}_k = \frac{4\bar{\ell}_{k,a} + 2\bar{\ell}_{k,d}}{6} \tag{3.3}$$

This equation is the result of looking at the possible inter-partition communications that can take place at a given level. Figure 3.6 shows the possible communications. There are 2 possible D-type communications and 4 possible A-type communications, which explains the coefficients in Equation 3.3. The calculations for $\bar{\ell}_{k,a}$ and $\bar{\ell}_{k,d}$ come from [28] and are as follows:

$$\bar{\ell}_{k,a} = \frac{4\mu}{3} - \frac{1}{3\mu} \tag{3.4}$$

$$\bar{\ell}_{k,d} = 2\mu \tag{3.5}$$

μ is the level-dependent scale factor of 2^k. The derivations of these two values is rather complicated so we refer to [28] for the full treatment. The basic idea is that they are the result of an average over the distances between all possible pairs of points on a grid in the adjacent and diagonal cases.

The description of the above techniques are an introduction to the techniques we will use in Section 3.3.2 to estimate the communication complexity of QECC encoded circuits. These techniques are largely technology-independent, so we will use them to validate our technology-dependent layouts in the next section.

3.2 Macroblock Layout Heuristics and Designs

Now that we have introduced our circuit level communication analyses, we are ready to introduce our constructive layout heuristics. We will then determine the goodness of these layouts in Section 3.3.2 by comparing the average wire length estimates from Donath's estimator with the average wire lengths in some of our heuristically generated layouts. We will first focus on small QECC encoded circuits since we are interested in validating movement profiles of the code structure not application characteristics.

3.2.1 Greedy Place and Route

Our first attempt at placing macroblocks on a substrate is to sequentially go through the gates in the circuit and place them "on demand", then to route wire elements

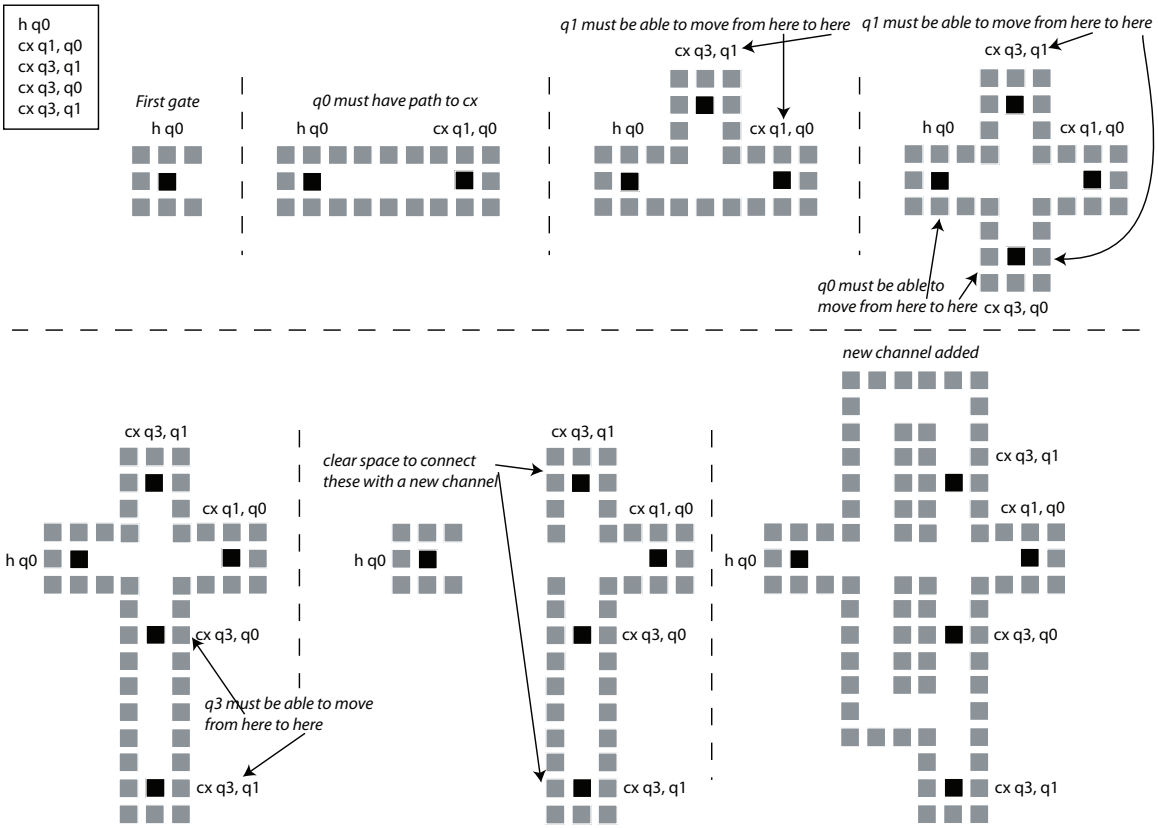


Figure 3.7: Step by step construction of a layout using the *greedy heuristic* to execute the circuit described by the QASM in the upper left box.

between only the gates that need to be connected each time. The intention here is to place gates that directly connect to one another, next to each other and place wires only where necessary.

The heuristic is a simple greedy algorithm that starts with only as many gate locations as qubits (because we assume that qubits only rest in storage/gate locations) and no channels connecting the gates. It iterates with a circuit scheduler, connecting gate locations until the qubits can communicate sufficiently to perform the specified circuit. The scheduler schedules gates prioritized by critical path, paths between connected gates are determined by a shortest path search on the layout graph. The current layout is fed into the circuit scheduler which tries to schedule until it finds qubits in gate locations that cannot communicate to perform a gate. The place and router then connects the problematic gate locations and tries scheduling on the layout again. The iteration finishes once the circuit can

be successfully completed. Our algorithm bears some similarity to the iterative procedure in adaptive cluster growth placement [57] in classical CAD. Gate locations are placed from the center outward as the circuit grows to fit a rectilinear boundary.

The placer can move gate locations that have to be connected if they are not already connected to something else. The router connects gate locations by making a direct path in the x and y directions between them and placing a new channel, shifting existing channels out of the way. Since channels are allowed to overlap, intersections are inserted where the new channels cut across existing ones. An example is shown in Figure 3.7.

This technique has the advantage that, since the circuit scheduler prioritizes gates based on gate delay critical path, potentially critical gates are mapped to gate locations and connected early in the process. Thus critical gates tend to be initially placed close together to shorten the circuit critical path. Additionally, gate locations that need to communicate can be connected directly instead of using a general shared grid channel network, where congestion can occur and cause qubits to be routed along unnecessarily long paths.

A disadvantage of this heuristic is that gate placement is done to optimize critical path, not to minimize channel intersections. This means that the layout could end up having many 4-way channel intersections and turns, both of which have more delay than 2-way straight channels. Additionally, even though critical gates are mapped and placed near each other, the channel routing algorithm tends to spread these gate locations apart as more channels cut through the center of the circuit. We see this spreading effect in Figure 3.7 as gates *h q0* and *cx q1, q0* are separated by more intersections and channels as more gates are connected. We discuss our experimental evaluation of this heuristic in Section 3.3.

3.2.2 Dataflow-Based Layouts

Generally, the greedy layout heuristic performs poorly because every new gate must try to connect itself to other gates as well as it can with little information as to how the qubit traffic its connections generate will affect the rest of the circuit. This optimization is highly localized and does not use enough available data on the global circuit structure.

Assuming we start by again placing a single gate location for each gate in a circuit, what would be the "optimal" placement of gate locations. Our primary goal is to have a circuit induce as little qubit movement, so we want to minimize the total distance all qubits must travel between gate locations. Additionally, since qubits may share communication

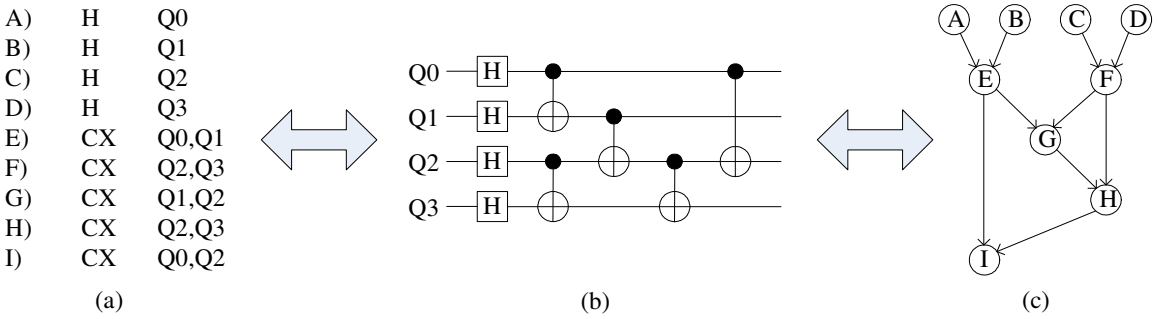


Figure 3.8: a) A QASM instruction sequence. b) A quantum circuit equivalent to the instruction sequence in (a). c) A dataflow graph equivalent to the instruction sequence in (a). Each node represents an instruction, as labeled in (a). Each arc represents a qubit dependency.

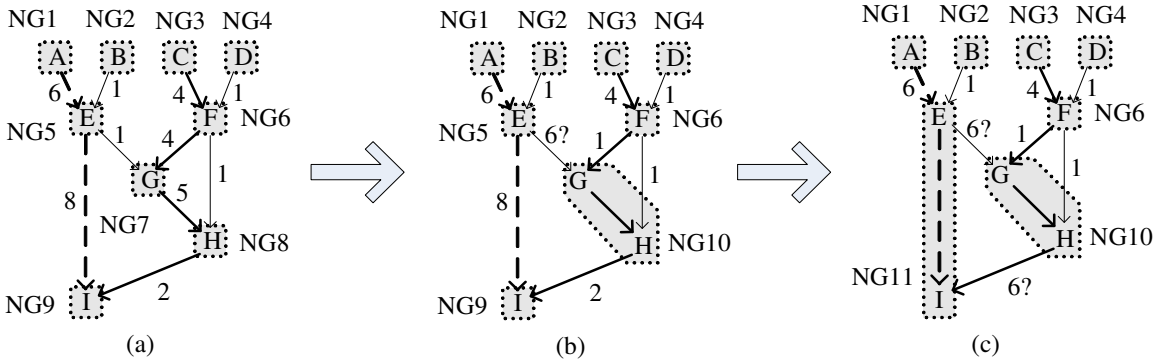


Figure 3.9: a) Each node (instruction) is initialized in its own node group (NG, outlined by the dotted lines), which corresponds to a physical gate location in a layout. Once placed, we extract physical distances between the nodes (the edge labels). b) We find the longest edge weight on the longest critical path (the length 5 edge on the path C-F-G-H-I; solid bold arrows) and merge its two node groups to eliminate that latency.
c) We recompute the critical path (A-E-I; dashed bold arrows) and merge its node groups, and so on.

channels between gates, it would be nice if we could minimize the amount on congestion on the shared channels. This leads us to the overall design decision of minimizing the distance between dependent gate locations in the layout. Short paths between gates minimize qubit movement and reduce the chance that there will be many shared connections over long distances.

To achieve this goal, we would like to find an approximate 2 dimensional embedding for a dataflow graph that minimizes edge length and edge crossings. Of course, for most dataflow graphs, an exact 2-D embedding is impossible, which precludes use of polynomial time algorithms like [66], which performs such an embedding if possible. Additionally, the time to produce this embedding is polynomial but still $O(n^6)$, which is expensive for larger circuits. Additionally, the problem of finding the minimum number of crossings in a graph embedding that does not fit into a plane is NP-Complete [34]. Thus we must take a heuristic approach to solving this problem. We instead create an approximate embedding by breaking the graph up into sets of dataflow nodes based on their "depth" in the dataflow graph. We lay out gate locations for the gates in each set so that they are close to each other and neighboring sets in the graph are also neighboring in the layout.

Dataflow Graph Analysis

Figure 3.8a shows a QASM instruction sequence consisting of Hadamard gates (H) and controlled bit-flips (CX) operating on qubits Q0, Q1, Q2 and Q3, with each instruction labeled by a letter. Figure 3.8b shows the equivalent sequence of operations in standard quantum circuit format. Either of these may be translated into the dataflow graph shown in Figure 3.8c, where each node represents a QASM instruction (as labeled in Figure 3.8a) and each arc represents a qubit dependency. With this dataflow graph, we may perform some analyses to help us place and route a layout for our quantum circuit.

The general idea is that we shall create node groups in the dataflow graph which correspond to distinct gate locations that may then be placed and routed on a layout. All instructions within a single node group are guaranteed to be executed at a single gate location. To begin with, we create a node group for each instruction, giving us a dataflow group graph, as shown in Figure 3.9a. If we lay out this group graph with a distinct designated gate for each instruction, we get a layout in which the starting location of each qubit is specified implicitly by its first gate location, so no additional initial placement

heuristic is needed.

From this layout we can extract movement latency between nodes and label the edges with weights (as in Figure 3.9a). We now find the longest critical path by qubit. The critical path A-E-I of qubit Q0 has length 14 (the dashed bold arrows), while the critical path C-F-G-H-I of qubit Q2 has length 15 (the solid bold arrows). We select the longest edge on the longest critical path, which is the edge G-H with weight 5. We merge these two node groups to eliminate this latency, in effect specifying that these two instructions should occur at the same gate location (Figure 3.9b). We then update the layout and recompute distances. Assuming we merged these two node groups to the location of H (NG8), then the weight of edge F-G changes to 1 (to match the weight of edge F-H) and the weight of edge E-G probably changes to 6 (former E-G plus former G-H), but the exact change really depends on layout decisions. The new critical path is now A-E-I, so if we do this again, we merge node groups NG5 and NG9 to eliminate the edge of weight 8, and we get the group graph in Figure 3.9c.

In merging nodes, there is the possibility that two qubit starting locations get merged, complicating the assignment of initial placement. For this reason, we add a dummy *input* node for each qubit before its first instruction. The merging heuristic doesn't allow more than one input node in any single node group, so we maintain the benefit of having an intelligent initial qubit placement without extra work.

There is an important trade-off to consider when taking this merging approach. A tiled grid layout provides plenty of gate location reuse but is unlikely to provide any pipelinability without great effort. A layout of the group graph in Figure 3.9a (with each instruction assigned to a distinct gate location) provides no gate location reuse at all but high potential pipelinability. This raises the question of whether we wish to minimize area and time (for critical data qubits), maximize throughput of a pipeline (for ancilla generation), or compromise at some middle ground where small sets of nearby nodes are merged in order to exploit locality while still retaining some pipelinability.

Gate Location Placement

Taking the group graph from the dataflow analysis heuristic, the placement algorithm takes advantage of the fanout-limited gate output imposed by the No-Cloning Theorem [104] to lay out the dataflow-ordered gate locations in a roughly rectangular block.

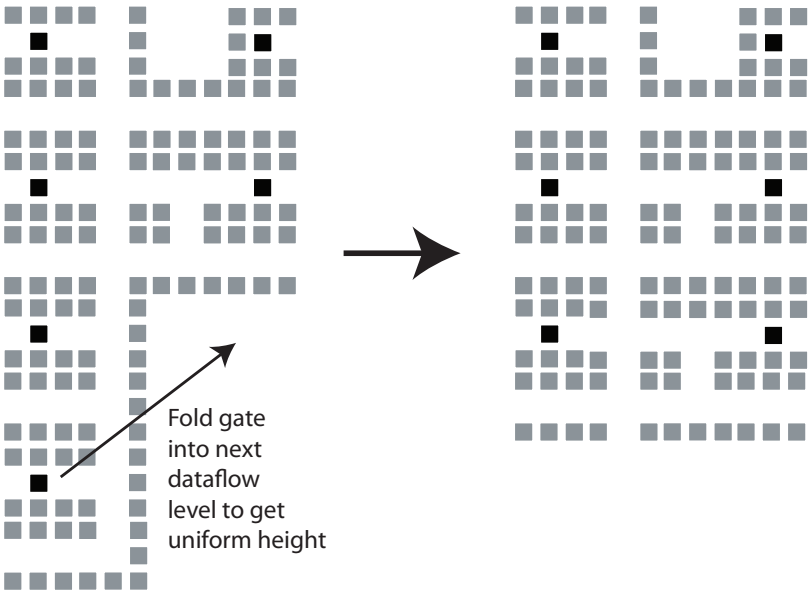


Figure 3.10: In order to avoid dramatic mismatches in dataflow column heights, we can *fold* tall columns into short ones.

We adopt a gate array-style design, where gate locations are laid out in columns according to the graph, with space left between each pair of columns for necessary channels. This can lead to wasted space due to a linear layout of uneven column sizes, so we may also perform a folding operation (shown in Figure 3.10, wherein a short column may be folded in (joined) with the previous column, thus filling out the rectangular bounding box of the layout as much as possible and decreasing area. Note that this technique will inevitably misalign some of the gates that were sorted before since we are potentially moving gates between different columns. In most cases this is a tradeoff between area and performance.

The columns are then sorted to position gate locations that need to be connected roughly horizontal to one another. This further minimizes channel distance between connected gate locations and reduces the number of high-latency turns. The column dataflow heuristic starts by finding the maximum path length in terms of gate count, through the circuit graph to be laid out. We get this by finding the longest path through the acyclic graph. The maximum depth determines the number of gate columns that will be in our layout. The columns are populated with the gates based on their depth in the circuit graph. Once we know which column each gate is in, we have to sort the gates within each column. We want them positioned so that gates that are dependent on one another are roughly in

the same row.

The sorting algorithm is relatively simple, we start with the first, leftmost column (the "input nodes"). We group the gates in this column such that gates that have the same output dependency are together in the column. We fix the leftmost column and group the gates in the next column to the right so that gates with the same input dependencies are together. Finally, we put these groups in roughly the same order as their input dependencies are in. Algorithm 2 gives more details on this algorithm.

Once gate locations are placed, we use a grid-based model in which we first route local wire channels between gate locations that are in adjacent or the same columns. These channels tend to be only a few macroblocks long each. One or more global channels are then inserted between each pair of rows and between each pair of columns of gate locations. These global channels stretch the full length of the layout.

Routing Channels and Route Determination

We call the global macroblock channels that we insert, the "global routing grid". The idea is that for qubits that have to move between non-adjacent columns, this grid connects everything to everything, although there would not be enough bandwidth for all connections at once. We then add additional local channels to connect gates in adjacent columns to allow these shorter movements without using the global grid.

There are no real routing constraints in our simple model since channels are allowed to overlap and turn into 3- or 4-way intersections. We depend on the dataflow column sorting in the placement phase to reduce the number of intersections and shared local channels. While local channels could technically be used for global routing and vice versa, we've found that this division in routing tends to divide the traffic and separate local from long-distance congestion.

With these basic placement and routing schemes, we may now iterate upon the layout, as shown in Figure 3.11. The technology-dependent netlist is translated into a dataflow group graph with a separate gate location for each instruction (Figure 3.9a). This group graph is then placed, routed and scheduled to get latency and identify the runtime critical path (as opposed to the critical path in the group graph, which fails to take congestion into account). The longest latency move on the runtime critical path (between two node groups) is merged into one node group, thus eliminating the move since a node group represents a

Algorithm 2 Column sorting for dataflow layout

```
traverse acyclic dataflow graph in topological order and store each vertex by level in
array Columns[][]
firstColumn = Columns[0]
for vertex in firstColumn do
  nextNeighbors = vertex.outNeighbors
  for neighbor in nextNeighbors do
    firstGroup = neighbor.inNeighbors
    for groupedVertex in firstGroup do
      if groupedVertex is below vertex in firstColumn then
        remove groupedVertex from firstColumn
        insert groupedVertex back into firstColumn next to vertex
      end if
    end for
  end for
end for
for column in Columns do
  for vertex in column do
    nextNeighbors = vertex.outNeighbors
    for neighbor in nextNeighbors do
      neighborColumn = Columns.find(neighbor)
      if neighbor is horizontally lower than vertex then
        remove neighbor from neighborColumn
        insert neighbor back into neighborColumn at same horizontal position as vertex
      end if
    end for
  end for
end for
```

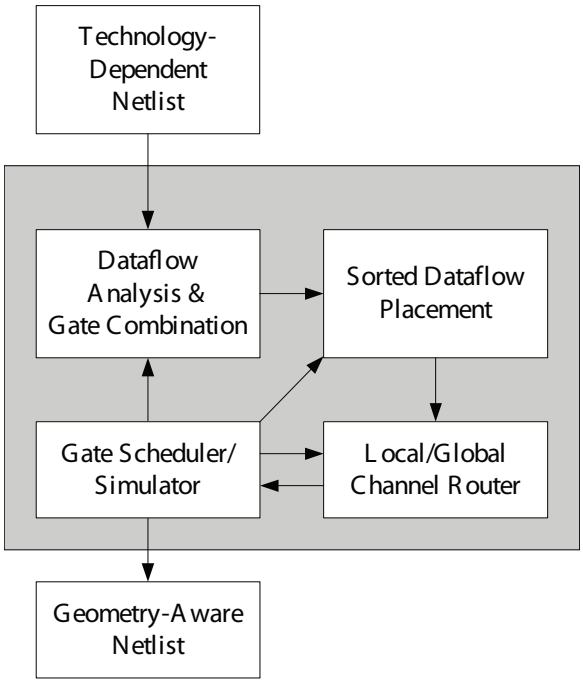


Figure 3.11: The dataflow placement and routing heuristic takes a technology-dependent netlist and translates it into a geometry-aware netlist through an iterative process involving dataflow analysis and placement and routing techniques.

single gate location. This new group graph is then placed, routed and scheduled again to find the next pair of node groups to merge.

Once this process has iterated enough times, we reach a point where congestion at some heavily merged node group is actually hurting the latency with each further merge. We alleviate this congestion by adding storage nodes (essentially gate locations that don't actually perform gates) near the congested node group. This increases the area slightly but maintains the locality exploited by the merging heuristic. If congestion persists, we halt the algorithm, back up a few merging steps and output the geometry-aware netlist.

Drawbacks

The dataflow placement heuristics work reasonably well for small circuits but as the circuit size gets bigger, we run into several problems:

- The difference between the widest and narrowest dataflow level increases, leading to

more wasted space in a column placement layout and more spreading of an original column across multiple folded columns.

- Even though we do a rough sorting to group gates with the same output dependency closer, this sorting cannot place everything together, thus the spread within a column between two gate locations that feed a single location in the next column increases on average.

- With greater circuit depth, the average number of columns between two dependent locations increases. We assume that qubits use the global routing grid for non-adjacent column communication so this increased distance puts more pressure on the global network.

Since this heuristic suffers from numerous issues in scaling to large circuits, we use it only to lay out small, low-level modules. We continue hierarchically, using a different heuristic for the high level module placement problem.

3.2.3 Simulated Annealing Module Placement

It turns out that the dataflow-style techniques work well for small circuits on the order of 100s of gates. For larger circuits, our simple column sorting heuristic is not sufficient to co-locate data-dependent gates. As each independent column gets taller, the clustering of gates within a column becomes more important and our greedy way of handling the problem is insufficient.

Instead, we opt to lay out larger circuits with a hierarchical approach. Our design flow takes advantage of implicit partitioning from our definition of different circuit modules. In practice, our fault tolerant circuits are typically broken down into error correction steps and encoded gates. Our larger circuit are also typically decomposed into submodules. For example, in Section 7.1.1 we discuss decomposing large 1024-bit adders into smaller subadder building blocks. Additionally, we can decompose circuits into submodules using Donath's partitioning method, introduced in Section 3.1.2.

We could then put these building blocks together with the same dataflow layout heuristic. This does not work as well however, since the more complex, higher-level blocks are typically connected to many more other blocks. This makes matching up frequently communicating blocks horizontally more difficult. Again, scalability concerns limit the

VPR option	Description
-place_algorithm=path_timing_driven	With this placement heuristic, VPR tries to minimize both average and critical path wire length by co-locating modules to be connected.
-router_algorithm=timing_driven	Do not try to minimize only the number of routing tracks, also take into account time taken for qubits to reach destinations.

Table 3.2: Parameters we use for our runs of the VPR place and route tool. More details on these options are in [12]

simple dataflow heuristic to small circuit layouts. At higher levels, we turn to a placement technique that is well established in the classical realm, simulated annealing.

Drawing an analogy between quantum and classical circuits, we choose to treat our hierarchical submodules that compose a larger design as programmable blocks in a data routing grid. This view looks a lot like field programmable logic arrays (FPGAs) [19] in classical computation fabrics. Continuing this analogy, we turn to a flexible classical FPGA placement and routing tool to do the initial placement and routing of our higher level blocks. This tool, VPR [13], places functional blocks using simulated annealing [53]. The optimization process tries to minimize the amount space necessary to route all the connections between blocks. Secondary optimization goals include minimizing average wirelength and critical path delay. These goals coincide with our own design goals of minimizing the distance qubits must travel between gates and minimizing area. Our additional goal of minimizing wire turns is not really accounted for in this model but in practice, we observe that simple low-turn-count paths are easier to work with and therefore preferred by VPR in the track based routing system it uses. Table 3.2 shows the various options we pass to VPR and their affects.

Since VPR is designed to place FPGA lookup table blocks and not our modules of quantum gates, we must export our existing circuit topology and dataflow layout to a format that is compatible with VPR. The existing layout information is exported to the *architecture* file and the circuit topology is output to a *netlist* file.

Architecture file This file contains information on the types of modules being placed. Each module has some number of possible input and output ports. For our purposes, we tell VPR that all modules are the same, thus we define a single module type and set the number of inputs and outputs to the maximums over all possible modules laid out with the dataflow heuristic.

Netlist file This instantiates and links together the modules declared in the architecture file, according to the circuit topology. So if we have a sum module feeding a carry module like in a ripple carry adder, this would have a netlist like this:

```
.input cin
.pinlist wirecin1

.input a
.pinlist wirea1

.input b
.pinlist wireb1

.input cout
.pinlist wirecout1

.clb sum
.pinlist wirecin1 wirea1 wireb1 open wirecin2 wirea2 wireb2 open
.subblock: blah0 0 1 2 open 4 open
.subblock: blah1 0 1 2 open 5 open
.subblock: blah2 0 1 2 open 6 open

.clb carry
.pinlist wirecin2 wirea2 wireb2 wirecout1 wirecin3 wirea3 \
       wireb3 wirecout2
.subblock: blah3 0 1 2 3 4 open
.subblock: blah4 0 1 2 3 5 open
.subblock: blah5 0 1 2 3 6 open
.subblock: blah6 0 1 2 3 7 open

.output out0
.pinlist wirecin3

.output out1
.pinlist wirea3
```

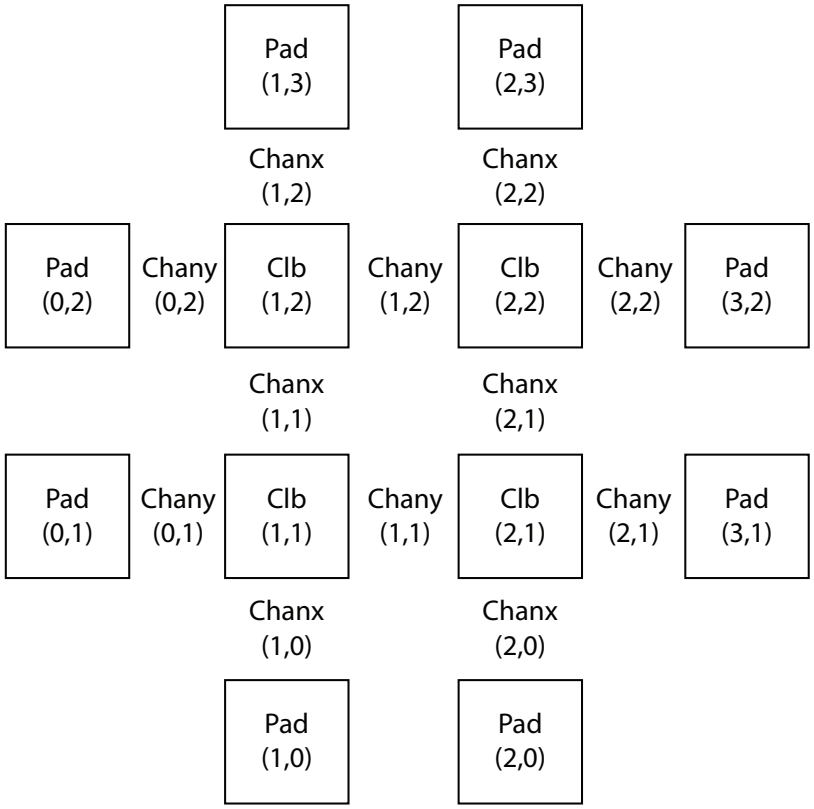


Figure 3.12: Grid of clbs/modules and the routing channels between them. Each clb is indexed by x and y coordinates. Channels are labeled with an x or y indicator and the x,y coordinates of the closes clb in the grid. Reproduced from VPR manual [12].

```
.output out2
.pinlist wireb3

.output out3
.pinlist wirecout2
```

Input and output edges from the module graph are declared as .inputs and .outputs. Each module in the circuit is translated to a .clb element with connections corresponding to the unique edge names for inter-modular communication. VPR assumes each .subblock has only a single output, so for each .clb, we need a .subblock for each output wire.

Once these configuration files are passed into VPR, it performs its optimization and creates a placement file and routing file. Using these, we construct the final layout for

the circuit. First, we must explain VPR's view of the layout. Figure 3.12 shows the gridlike structure that VPR follows to place modules and communication channels.

Placement file This file determines the positioning of each module in the grid. For the example of our carry-sum circuit, the format is as follows:

```
Array size: 2 x 2 logic blocks
#block name   x    y   subblk
cin           0    1   0
a             0    2   0
b             1    0   0
cout          2    0   0
sum           1    1   0
sum           1    1   1
sum           1    1   2
carry         2    1   0
carry         2    1   1
carry         2    1   2
carry         2    1   3
out0          3    1   0
out1          3    2   0
out2          1    3   0
out3          2    3   0
```

Input and output blocks are placed on the edge of the grid (either at x/y = 0 or x/y = 3) and all modules/clbs are placed in internal grid squares. Taking this grid specification, we place the layouts generated by the dataflow heuristic for each module and place them at their respective macro-grid locations. Each macro-grid square must be large enough to accommodate the largest dataflow layout. This could lead to empty space if module layouts have variable sizes. We must also leave room between neighboring modules for a "channel" that consists of multiple "tracks" that are made up of actual channel macroblocks. The number of tracks that go in a channel is determined by the routing file.

Routing file The routing file specifies which channel tracks connect to what. For each channel space between modules, it specifies for each channel, whether there should be a wire or macroblock channel horizontally or vertically. It also specifies which pins on the module inputs and outputs connect to which routing track. Here is an example file:

```
SOURCE (0,1) Class: 0
```

```
OPIN (0,1) Pad:0
CHANY (0,1) Track: 0
IPIN (1,1) Pin:0
SINK (1,1) Class: 0
SOURCE (0,2) Class: 0
OPIN (0,2) Pad: 0
CHANY (0,2) Track: 1
CHANY (0,1) Track: 1
IPIN (1,1) Pin: 1
SINK (1,1) Class: 0
SOURCE (1,0) Class: 0
OPIN (1,0) Pad: 0
CHANX (1,0) Track: 1
CHANY (0,1) Track: 2
IPIN (1,1) Pin: 2
SINK (1,1) Class: 0
```

In the routing file, a particular net/wire/communication link is defined as a sequence of channel and track specifications between a source and sink statement. Nets connect to modules through OPINs and IPINs.

We allocate a single channel per routing track when we lay down macroblocks. We route all encoded qubits down the same inter-block channel since they all have the same source and destination and travel together.

3.2.4 Manual Layouts

In addition to automated macroblock layout heuristics, we can also manually specify macroblock layouts that can then be tied to specific circuits. We will discuss which circuits we choose to lay out manually and why.

Quantum error correction circuits feature complex communication patterns and are particularly sensitive to error. This is because qubits in the encoding and verification circuits are not encoded yet and thus cannot be corrected so easily. Additionally, since previous estimates have estimated 95% of resources will go toward error correction, it makes sense to ensure underlying layouts are area and latency efficient.

There have also been prior efforts to lay out quantum circuits. Svore et al [95] laid out cells of a swap-based quantum computing technology in order to calculate local pseudo-thresholds for the Steane $[[7, 1, 3]]$ code. Kreger-Stickles et al [56] investigated several layouts for the same code in an effort to determine the best one in terms of latency, area, and some

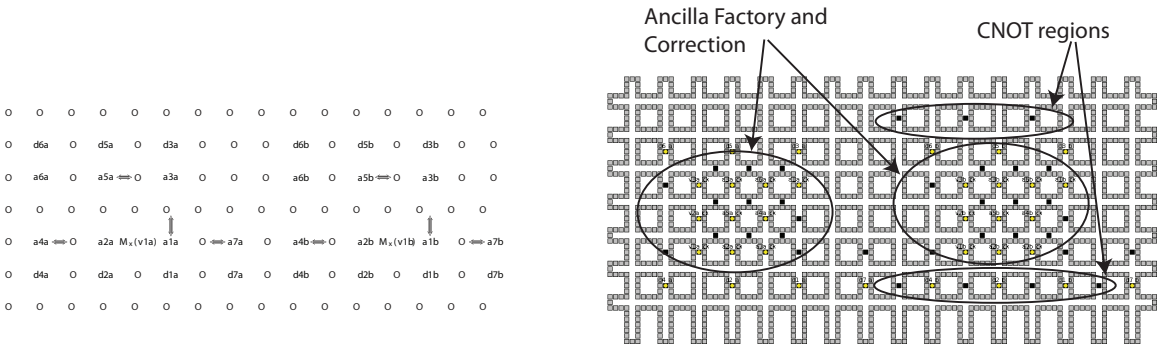


Figure 3.13: On the left is a figure from Svore's work on pseudo-thresholds for the $[[7,1,3]]$ code. On the right is our adapted ancilla factory in ion trap macroblocks, based on her design. This design consists of 2 ancilla factories and correction stages, as well as space for an encoded CNOT gate.

measurement of errors. We have integrated both resulting hand layouts into our own tool flow and have also laid out several other circuits.

Ancilla Factories

Steane originally introduced the notion of an ancilla factory in [89]. As we showed in Figure 1.9, the actual correction of data takes clean encoded zero ancilla. The idea is that specialized modules produce this ancilla. We can optimize these factories for high throughput ancilla production and that is indeed what several works have done.

Our first ancilla factory is an adapted Svore factory for the $[[7,1,3]]$ code from [95]. The design shown in 3.13 actually consists of 2 ancilla factories with correction and an encoded CNOT. This factory was designed to minimize movement and idle errors, so it is not particularly area efficient. We will later compare this ancilla factory's performance with layouts of the same circuit done using our heuristics.

Another ancilla factory from the literature is Kreger-Stickles 4-bit Linear Offset factory for producing $[[7,1,3]]$ ancilla for correction. This design is shown in 3.14 and is the best performing ancilla factory from [56] in terms of fault point count as well as latency. It also has a pretty compact area.

In our work in [48], we developed our own *pipelined* ancilla factory for the $[[7,1,3]]$ that emphasized ancilla throughput per unit area. That design is shown in Figure 3.15, and

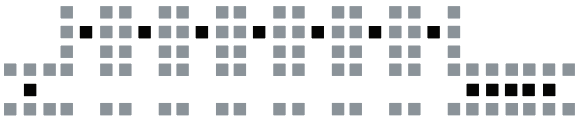


Figure 3.14: "Linear Offset" [[7, 1, 3]] ancilla factory from Kreger-Stickles work on area/error optimal ancilla factories for the Steane code. The 7 bits on the top are the ancilla to be encoded and verified, the rest of the bit are for the verification process.

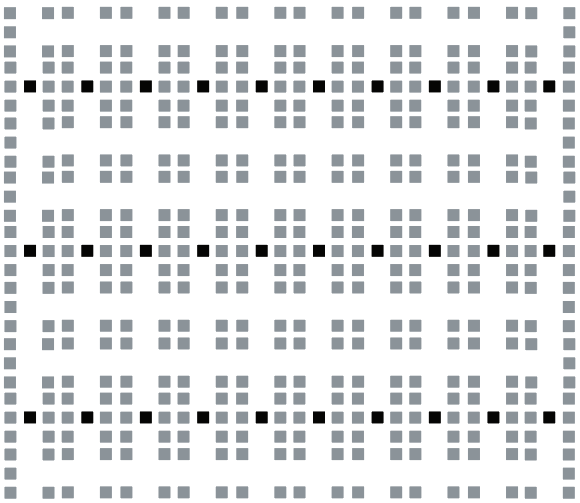


Figure 3.15: Our own pipelined [[7, 1, 3]] ancilla factory. Ancilla move through the gate locations from top to bottom.

is the only design that can prepare multiple ancilla states in parallel out of the factories mentioned here. This pipelined ancilla approach is particularly fitting since the Qalypso microarchitecture, to be discussed in Section 3.4.2 allows for flexible resource allocation to achieve high utilization of its ancilla production resources.

Teleportation Units

As we mentioned in Section 2.4, teleportation is a key component to the large scale tiled dataflow architectures. In order to fully analyze all the errors in the system, we have laid out a teleportation router in macroblocks, in order to accurately extract all the errors involved in our communication network and to get an accurate area estimate.

A high-level floorplan of a network router is shown in Figure 3.16, from our work

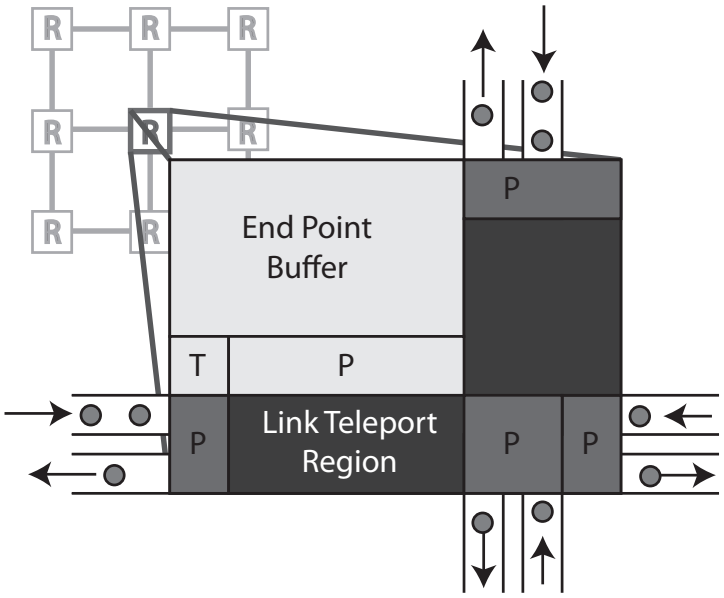


Figure 3.16: Network Router. Dark gray areas support single hop links between neighboring routers. Light gray regions handle connections that terminate locally. The size of the End Point Buffer is dictated by the size of the logical qubit being teleported.

in [63]. The area of the router consists of purifiers (P) for EPR purification, teleporters (T) for connections that span multiple routers, and buffers to store qubits while waiting for the connection establishment. The area dedicated to each of these components is dependent on the maximum load the router sees, as determined by the mapping phase. Our tools cannot yet construct a detailed layout of arbitrarily sized routers. Instead, in an effort to obtain realistic network area estimates, we utilize a detailed layout of a specific sized router to extrapolate the sizes of larger routers.

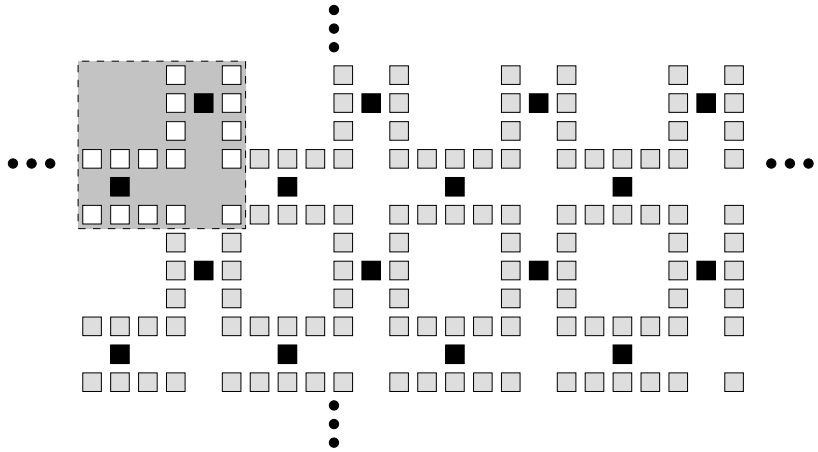


Figure 3.17: QPOS grid structure constructed by tiling the highlighted 2×2 macroblock cell.

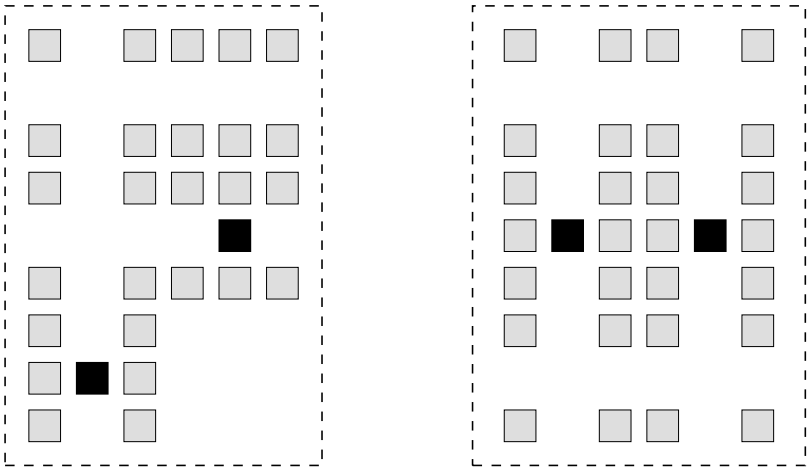


Figure 3.18: Comparison of the best 3×2 cell for two different circuits. (a) The best cell for the $[[23, 1, 7]]$ Golay encode circuit. (b) The best cell for the $[[7, 1, 3]]$ L1 correct circuit.

3.2.5 Grid-based Layouts

In order to additionally validate our layout heuristics we consider grid-based layouts that are from previous work or exhaustively searched. In all grid-layout works, a layout is constructed by first designing a primitive cell and then tiling this cell into a larger physical layout. For example, the authors of [64, 65] manually design a single cell, and for any given

quantum circuit, they use that cell to construct an appropriately sized layout. In [94], the authors automate the generation of an H-Tree based layout constructed from a single cell pattern. Similarly, [8] uses a cell such as in [94] but also provides some tools to evaluate the performance of a circuit when the number of communication channels and gate locations within the cell is varied. We use a combination of these methods to implement a tool that automatically creates a grid-based physical layout for a given quantum circuit.

The grid-based physical layouts generated by our tools are constructed by first creating a primitive cell out of the macroblocks and then tiling the cell to fill up the desired area. For example, Figure 3.17 shows how a 2×2 sized cell can be tiled to create the layout used in [65] (referred to henceforth as the QPOS grid). These types of simple structures are easy to automatically generate given only the number of qubits and gate operations in the quantum circuit. Furthermore, grid-based structures are very appealing to consider because, apart from selecting the number of cells in the layout and the initial qubit placement, no other customization is required in order to map a quantum circuit onto the layout. The regular pattern also makes it easy to determine how qubits move through the system, as simple schemes such as dimension-ordered routing can be used.

The approach we use to generate the grid-based layout for a given quantum circuit is as follows:

1. Given the cell size, create a valid cell structure out of macroblocks.
2. Create a layout by tiling the cell to fill up the desired area.
3. Assign initial qubit locations.
4. Simulate the quantum circuit on the layout to determine the execution time.

The first step finds a valid cell structure. A cell is valid if all the macroblocks that open to the perimeter of the cell have an open macroblock to connect to when the cell is tiled. Also, a cell cannot have an isolated macroblock within it that is unreachable. Once we tile this valid cell to create a larger layout, we must decide on how to assign initial qubit locations. The two methods we utilize are: a systematic left to right, one qubit per cell approach, and a randomized placement. The systematic placement allows us to fairly compare different layouts. However, since the initial placement of the qubits can affect the performance of the circuit, the tool also tries a number of random placements in an effort to determine if the systematic placement unfairly handicapped the circuit.

This layout generation and evaluation procedure is iterated upon until all valid cell configurations of the given size are searched. We then repeat this process for different cell sizes. The cell structure that results in the minimum simulated time for the circuit is used to create the final layout.

Figure 3.18 shows the best cell structure found by conducting a search of all 2×2, 2×3, and 3×2 sized cells for two different circuits. The main result of this search is that the best cell structure used to create the grid-based layout is dependent on what circuit will be run upon it. By varying the location of gates and communication channels, we tailor the structure of the layout to match the circuit requirements.

While this type of exhaustive search of physical layouts is capable of finding an optimal layout for a quantum circuit, it suffers from a number of drawbacks. Namely, as the size of the cell increases, the number of possible cell configurations grows exponentially. Searching for a good layout for anything but the smallest cell sizes is not a realistic option. Furthermore, while small circuits may be able to take advantage of primitive cell based grids, larger circuits will require a less homogeneous layout. One approach to doing this is to construct a large layout out of smaller grid-based pieces, all with different cell configurations. While this approach is interesting, we feel a more promising approach is one that resembles a classical CAD flow, where information extracted from the circuit is used to construct the layout.

3.3 Layout Performance

Given all these choices of layouts and layout heuristics, we are ready to make some comparisons to see under what conditions which ones perform well. One goal of this section is to determine whether the layouts generated by our heuristics are "good" in the sense that they do not detrimentally impact latency or area, or introduce an unnecessary amount of errors. To ensure this goodness, we will compare layouts produced by our heuristics with hand-tuned layouts from the literature, layouts from exhaustive searches of the design space, and estimations from Donath's wire length estimation from Section 3.1.

3.3.1 Latency and Area Comparison

We will start by evaluating the area and latency of all the heuristics and layout searches on some benchmark QECC modules. We will also compare these to the results

Circuit name	Qubit count	Gate count
$[[7,1,3]]$ L1 encode [85]	7	21
$[[23,1,7]]$ L1 encode [90]	23	116
$[[7,1,3]]$ L1 correction [5]	21	136
$[[7,1,3]]$ L2 encode [85]	49	245

Table 3.3: List of our QECC benchmarks, with quantum gate count and number of qubits processed in the circuit.

from some other layouts from prior work.

Benchmarks

Relatively high error rates of operations in a quantum computer necessitate heavy encodings of qubits. As such, we focus on encoding circuits (useful for both data and ancillae) and error correction circuits to experiment with circuit layout techniques. We lay out a number of error correction and encoding circuits to evaluate the effectiveness of the heuristics used in our CAD flow in terms of circuit area and latency, as determined by our scheduler. Our circuit benchmarks are shown in Table 3.3. We use two level 1 (L1) encoding circuits, a level 2 (L2) recursive encoding circuit and a fault-tolerant level 1 correction circuit.

For these encoding and correcting circuits, throughput is a more important measure than latency, implying that they would benefit greatly from pipelining. Nonetheless, a high latency circuit could introduce non-trivial error due to increased qubit idle time. On the other hand, correction circuits are much more latency dependent, since they are on the critical path for the processing of data qubit blocks.

Evaluation

We have evaluated a variety of layout design heuristics on the four benchmarks shown in Table 3.3. The results are in Table 3.5. "QPOS Grid" refers to the best scheduled macroblock-level layout heuristic from the literature [65]. "Optimal Grid" refers to the best grid with an area matching the QPOS Grid used that was found by the exhaustive

Category	Layout Heuristic or Design	Description
Exhaustive	Optimal Grid	From Section 3.2.5. Our work on exhaustive search over different grid structures from [101]. Searches all possible grid sizes and grid "building blocks" up to a certain size. Designed to minimize latency.
Manual	Svore	Mentioned in Section 3.2.4, from previous work in [95]. Hand laid out and scheduled grid structure for the $[[7,1,3]]$ code, which was used to prove a threshold exists for this code even with movement errors accounted for. Designed to minimize movement and idle error, so has good area/latency tradeoff.
	Kreger-Stickles	Mentioned in Section 3.2.4, from previous work in [56]. From this work, we use the "linear offset" layout, which one of the best performing layouts in their study. They investigated a number of area-minimal designs to find the ones that have the fewest points where errors may occur, including movement, idle and gate errors. Designed to have minimal qubit gate, movement and idle events.
Heuristic	QPOS Grid	From Section 3.2.5. Quantum Physical Operations Scheduler [65]: Arranged as a grid of gate locations. Gate sequencing is done based on priorities assigned based on critical paths through the program dataflow graph. Designed to minimize latency.
	Greedy channel and gate location placement	Heuristic from Section 3.2.1, also in our work in [101]. Layout is constructed by going through dataflow graph, adding gate locations for gates and channels only when a scheduler cannot make further progress. Designed to be minimal area.
	Non-folded DF, 2 global channels, critical combining	Dataflow heuristic from Section 3.2.2, also in our work in [101]. Columns of gate locations are laid out in dataflow order with routing channels added between the columns. For this version of the heuristic, columns are *not* folded over to normalized column height. Two vertical channels are allocated for "global" routing (between non-adjacent columns). Critical combining means we are using our dataflow group graph analysis.
	Folded DF, 1 global, critical comb.	Similar to the non-folded DF case above but long columns are folded into short columns to get a more compact layout, where every column is nearly the same height. Also, only a single vertical global routing channel is inserted between columns.
	Folded DF, 2 global, critical comb.	Similar to above but with 2 vertical global routing channels.

Table 3.4: Description of the different layout types compared in Table 3.5

search over a uniform grid layout. This is our best performing heuristic but is prohibitive in runtime. "Greedy" refers to the heuristic described in Section 3.2.1. "DF" refers to the dataflow-based approach from Section 3.2.2. "Non-folded" means the dataflow graph is laid out with varying column widths; "folded" means the layout has been made more rectangular by stacking columns. The number of global channels is between each pair of rows and columns of gate locations. "Critical combining" refers to our dataflow group graph merging heuristic from Section 3.2.2.

To start by comparing the top performing layouts in terms of latency, we notice that the Svore hand layout of the $[[7,1,3]]$ encoder has the best latency out of all layouts. We expect this layout to do well since it is optimized to minimize movement and idle errors,

Circuit	Layout Heuristic or Design	Latency (μs)	Area
$[[7,1,3]]$ L1 encode	QPOS Grid	548.0	49
	Optimal Grid	509.0	49
	Svore hand layout	310	56
	Kreger-Stickles hand layout	520	14
	Greedy channel and gate location placement	648.0	36
	Non-folded DF, 2 global channels, critical combining	768.2	231
	Folded DF, 1 global channel, critical combining	795.4	126
	Folded DF, 2 global channels, critical combining	712.4	182
$[[23,1,7]]$ Golay L1 encode	QPOS Grid	2268.0	575
	Optimal Grid	1801.0	575
	Greedy channel and gate location placement	2457.0	168
	Non-folded DF, 2 global channels, critical combining	2169.2	3880
	Folded DF, 1 global channel, critical combining	2264.0	713
	Folded DF, 2 global channels, critical combining	2248.2	1394
$[[7,1,3]]$ L1 correction	QPOS Grid	1300.0	1271
	Optimal Grid	771.0	1271
	Svore hand layout	990	156
	Kreger-Stickles hand layout	1430	30
	Greedy channel and gate location placement	1932.0	756
	Non-folded DF, 2 global channels, critical combining	999.8	2378
	Folded DF, 1 global channel, critical combining	1501.2	690
	Folded DF, 2 global channels, critical combining	1121.2	1496
$[[7,1,3]]$ L2 encode	QPOS Grid	2411.0	1365
	Optimal Grid	1367.0	1365
	Greedy channel and gate location placement	4791.0	936
	Non-folded DF, 2 global channels, critical combining	1582.4	4087
	Folded DF, 1 global channel, critical combining	1828.6	1617
	Folded DF, 2 global channels, critical combining	1944.8	3381

Table 3.5: Latency results for a variety of ECC circuits with different placement and routing heuristics.

which should yield low overall latency. Even more interesting is if we compare the latency of the Svore layout with our optimal grid search. In this case, the optimal grid search actually does better. We consider this indicative of the fact that as circuits get more complex, a designer becomes less effective at intuitively laying out a good circuit.

Considering the best area layouts, the Kreger-Stickles design wins. This is not surprising, since it is also optimized to minimize some combination of possible fault points as well as area. Both Kreger-Stickles layouts have higher latency than the optimal grid and Svore layouts.

The exhaustive search over grids yields the best latency for all benchmarks, which is not surprising. This kind of search becomes intractable quickly as circuit size grows, and additionally, it is based on the unproven assumption that a regular layout pattern is the best approach. We include this data point as something to keep in mind as a target latency.

Among the non-exhaustive search layouts, we first note that no single heuristic

or hand layout is optimal for all four benchmarks and that, in fact, no single heuristic optimizes both latency and area for any single circuit. Dataflow-based place and route techniques in general produce the lowest latency circuits. We find that the optimal global channel count per column (1 or 2) depends on the circuit being laid out. This is an artifact of the lack of maturity in our routing methodology. We intend to explore more adaptive routing optimization in our ongoing work.

The dataflow approach and the QPOS Grid tend to trade off between latency and area. However, we expect that the dataflow approach will show greater potential for pipelining, thus allowing us to target circuits such as an encoded ancilla generation factory, for which throughput is of greater importance than latency. We also observe that non-folded dataflow layouts are likely to have even greater pipelinability than folded ones, but at the likely cost of greater area. Although, we should note that the area estimates for the non-folded DF-based layouts are in fact overestimates due to our use of a liberal bounding box for these calculations.

We find that the greedy heuristic tends to find the best design area-wise for small circuits, but the latency penalty increases with circuit complexity. This is expected, as greedy is unable to handle congestion problems, so it works best for small circuits where congestion is not an issue. It is for the opposite reason that the DF heuristics fail on the $[[7, 1, 3]]$ L1 encode. They insert too much complexity into an otherwise simple problem.

Prior work has tended to assume a specific regular grid structure and to schedule operations within this structure. Our investigation into a variety of grid structures and showed a performance variance of a factor of four as we varied grid structure and initial qubit placement. For the small encoder circuits ($[[7, 1, 3]]$ and $[[23, 1, 7]]$ L1 encoders), both the greedy and dataflow heuristics are within 50% of the exhaustive grid search and prior work in QPOS in terms of latency. The hand layouts show good area and latency performance in general, being comparable or better than the optimal grid layout.

As we scale up in size, we see the latency of the greedy layouts increase markedly. The single channel folded dataflow layouts give a good area-latency performance tradeoff for larger circuits, yielding a better area than the optimal grid in one case and within 20% in another. The latency of the dataflow layouts is also good for larger circuits, coming within 30% of the optimal grid layouts in this case too. For this reason, we opt to go forward in our study using the folded dataflow layout heuristic as our chosen macroblock-level layout technique.

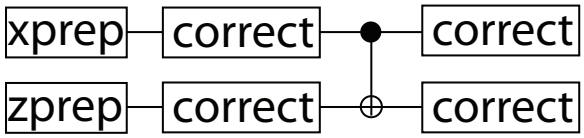


Figure 3.19: In this circuit, each qubit line represents a logical encoded qubit and the gates are logical, encoded gates. The correct blocks correspond to a Steane-type error correction step [87].

3.3.2 Validating Layouts with Donath's Estimate

Now that we have looked at the performance of our heuristics as compared to one another and also to hand laid out circuits, we will compare them to the results of our analytical Donath technique from Section 3.1. We have already seen the dataflow heuristic beat some of the hand optimized layouts in terms of latency for particular circuits, we will now do a more systematic comparison of the folded dataflow heuristic to Donath's wire length estimation for all the quantum error correcting codes we will later compare.

Since the error correction encoding and verification operation is particularly sensitive to errors, we want to ensure that the number of additional, non-gate errors introduced by the layouts of these circuits is kept at a minimum. In order to further validate the quality of our layouts, we want to ensure that qubits are not traveling excessive distances to get from one gate location to the next.

We start our validation by looking at communication patterns in QECC encoders. The purpose of this analysis is to provide validation that the dataflow layout heuristic we use for laying out the code blocks in this chapter do not introduce unreasonable amounts of communication for any of the codes. We use Donath's wire length estimation technique described in Section 3.1.2 on the encoded cnot circuit shown in Figure 3.19, which is the same as the circuit used for Figure 3.4. Using Donath's estimates as reference values, we compare the *folded dataflow layout* from Section 3.2.2 for the same circuits.

More specifically, for every code, we compute an average wire length for using Donath's estimation. To compare with this, we lay out the circuit with the dataflow heuristic. Given the layout, we go through each route for a gate-to-gate communication specified in the circuit, and compute the length in macroblocks. We take the average over these route lengths and compare the actual average with the estimated average.

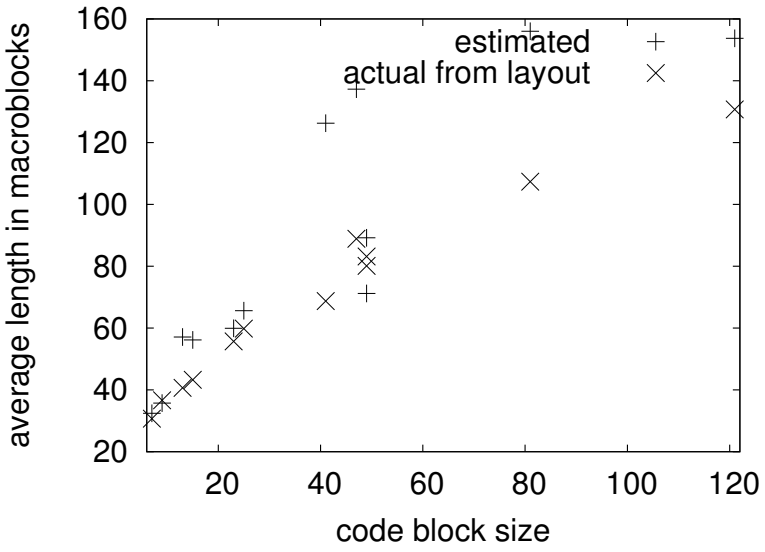


Figure 3.20: Comparing average communication distance estimates from Donath's estimation to real distances in an ion trap layout with the folded dataflow heuristic.

Figure 3.20 shows average wire lengths for the codes listed in 3.1. In this graph, points should be compared vertically to determine the difference in the two estimates. The "actual from layout" numbers are macroblock-accurate averages from the dataflow heuristic. We see that in most cases, the dataflow heuristic yields average distances that are substantially better than those estimated by Donath's rule. One explanation for this, besides the fact that our heuristic is good, is that Donath's estimation assumes a uniform distribution of starting and ending points, for cross partition connections. Our dataflow heuristic tries to align communicating gates in the same horizontal line, which would decrease the average distance from what the uniform distribution would yield, at every level of the partitioning.

We do not expect these two techniques to match exactly, only that our layout heuristics do not introduce extra movement that is way off the expected value or that the layouts unfairly penalize particular codes with extra movement. We see that in fact, the only codes in which our heuristic yields worst average lengths than the validating estimator are the $[[9, 1, 3]]$ code ($< 5\%$ difference) and the $[[49, 1, 7]]$ code (12% difference).

Datapath	Data Regions			Memory Regions			Ancilla Generator	Non-Trans Gates	Interconnect
	Total	Qubits	Gens	Total	Qubits	Gens			
LQLA	D	2	2	none			[56]	anywhere	fixed-size routers, one per data/memory region
Qalypso	D	D_q	D_{ag}	M	M_q	M_{ag}	[48]	placed with custom ancilla	variable sized routers adapted to design

Table 3.6: Details of various datapath organizations. A datapath consists of a number of Data Regions and in some cases Memory Regions. Each Data/Memory region is sized to hold a specific number of Qubits and Ancilla Generators (Gens) and regions are connected via an Interconnection Network. The D and M variables in the table signify values that are only determined after a quantum circuit is mapped onto the datapath.

3.4 Coarse Grained Mapping and Routing

Our macroblock-level heuristics are not well suited for gate counts in the millions, even with the VPR-based simulated annealing approach, so must yet again try to organize things hierarchically in order to handle the scales of circuits we are interested in.

3.4.1 Tiled Microarchitectures

As we mentioned in Section 2.4, we build upon work on tiled microarchitectures for quantum computers in [64, 98, 56] for our high level organization.

The parallelism of these tiled datapaths are in part convenient because apparently ubiquitous quantum error correction *must* be done in parallel to keep up with potential idle error sources. We have already discussed the structure of the various microarchitectures so we will now only discuss the two datapaths we will be using going forward for our large scale circuit mappings. If we refer back to Figure 2.6, we see that each tile can contain area dedicated to compute, memory, and ancilla resources. Table 3.6 shows a comparison of various microarchitectural parameters that determine relative sizings of these regions.

3.4.2 Qalypso and LQLA

The "LQLA" datapath consists of the non-specialized tiled datapath of QLA [64], except that the ancilla generation unit for error correction in each tile is replaced with

the "Linear Offset" ancilla factory from [56], also discussed in Section 3.2.4. In this work they showed that this particular ancilla generator layout has good error, latency and area properties. Since these goals match with ours as well, we opt to use this ancilla cell in QLA as our baseline datapath.

Qalypso refers to our own datapath, introduced in [48]. Qalypso provides complete flexibility in the number of Qubits (D_q) and Ancilla Generators (D_{ag}) per data region as well as number of Qubits (M_q) and Ancilla Generators (M_{ag}) per memory region. In order to provide this flexibility, we must have a good method by which to map computation to these elements.

3.4.3 Partitioning the Circuit

During the mapping process, the mapper must determine the total number of data regions (D) and memory regions (M) required. For LQLA, $M = 0$ which makes it easy to determine D as it is simply sized to accommodate the number of qubits used in the quantum circuit. The addition of memory regions introduces trade-offs in area and exploitable parallelism (latency). A datapath with a single compute region and a sea of memory can only perform one operation at a time — resulting in longer latency with a minimal area. A datapath such as QLA with all compute regions and no memory can exploit all possible parallelism in the circuit but with extremely high cost in area.

The mapper determines where each data qubit will reside during the course of the execution as well as when and where each quantum gate will execute. It starts with a coarse-grained partitioning of modules to compute-regions that minimizes communication. Next, the mapper attempts to schedule each gate operation so that it occurs as late as possible, while prioritizing operations on the critical path. The mapper relocates qubits into memory regions (if available) to free up compute regions for subsequent operations. As the mapper progresses, it tracks the location and times of all gate operations, error corrections, and network connections needed to perform the quantum circuit. The mapper discourages imbalanced mappings, such as those that over utilize network links or ancilla generation resources.

If the target datapath has fixed ancilla generation resources, the mapper attempts to map operations to regions with unused ancilla bandwidth. In datapaths with flexible ancilla generation, like Qalypso, the mapper assumes that operations will never wait for

ancilla, while still attempting to balance ancilla usage. A later phase (described below) matches ancilla generation resources to demand.

In Chapter 7, show results comparing our own Qalypso microarchitecture with the previous state-of-the-art QLA-based approach from the literature, LQLA. We will see that Qalypso offers substantial benefits in terms of reduced area and latency due to its more flexible and efficient ancilla generation infrastructure. For further evaluation of different tiled dataflow microarchitectures, we refer the reader to our work in [63].

In this Chapter, we have discussed the spatial layout of gates on a substrate both to get a fabricatible design and also to determine the impact of communication. We combine macroblock-level layouts of individual compute regions in a tiled dataflow microarchitecture to get a full computer design. We also make use of technology agnostic wire length estimation techniques from classical CAD in order to evaluate the fitness of our macroblock layout heuristics.

Going forward, we will now look at how errors are extracted from these layouts and microarchitectures and how they can be simulated to yield overall system reliability measures.

Chapter 4

Fault Tolerance Verification

To perform a detailed analysis of the error probability of circuits, we need a way to model the incidence of errors and their propagation through the circuit. There have been a number of works that have analyzed the fault tolerance of circuits with varying degrees of generality and accuracy. The initial ground-breaking work done by Aharonov et al[2] established the existence of a *threshold* for quantum error correcting codes. The threshold being the maximum gate error rate that can be corrected with unlimited error correction. This work analyzed CSS codes of distance greater than or equal to 5 and provided a very conservative estimate the maximum correctable physical gate error rate. These rigorous bounds on the maximum gate error rate were subsequently improved with different codes and more sophisticated analyses [5, 78]. All of these works focused only on establishing a threshold for circuits with only gate errors. Additional work has been done to establish the continued existence of a threshold in the face of qubit idle and movement errors [37, 97, 96]. Through these works, we know it is possible, at least in principle, to build a fault tolerant

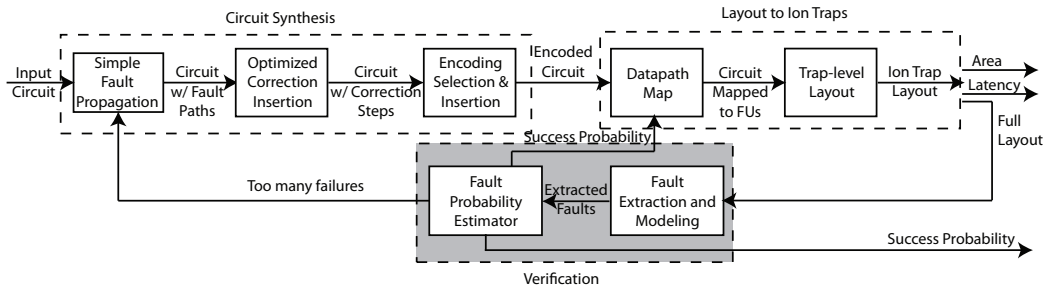


Figure 4.1: Fault estimation/verification portion of CAD flow.

quantum computer using error correcting codes.

The next question we must ask is: what is the best way to proceed? The rigorous threshold bounds establish the existence of properly fault tolerant quantum computers but do not analyze tradeoffs in code architecture to find a design that has low latency or small area. There are other, more heuristic analyses that compare different fault tolerant architectures in an effort to find the "best". Code comparisons in [90] and [25] have suggested that the Steane $[[7, 1, 3]]$ code and the Golay $[[23, 1, 7]]$ code have the highest thresholds, although neither of these took movement or idle error sources into account. Higher-level studies have addressed how different memory/compute architectures impact computer performance, including fault tolerance [56, 98]. None of these works provide a systematic study of all the resources involved in choosing a particular fault tolerance architecture.

We have already introduced tools to automatically synthesize and lay out QECC encoded circuits such that area, latency, and qubit movement are minimized. This enables us to do a detailed study of the area and latency costs of different encodings. Our next step is to present our fault tolerance verification tools, so that we can evaluate whether these circuits and layouts can be used to build larger systems with a low probability of data corruption.

We have developed a number of fault tolerance verification methods with differing accuracy and runtime tradeoffs. We will talk about how to compose these methods in order to build a flexible toolkit for analyzing the errors in circuits and layouts from the small to the large scale. The intention is to combine these tools to accomplish two complimentary tasks:

- Provide a *comprehensive comparison of error correcting codes* with all error overhead to help determine a resource efficient fault tolerant architecture.
- Enable the fault tolerance verification of large scale application circuits, incorporating as many error sources as possible while still maintaining a reasonable runtime.

In this chapter we will first present a high level model of error propagation in quantum circuits and layouts, then talk about some ways to use this model to simulate errors through an entire design. Next we will talk about some ion-trap specific refinements on the earlier error model and then discuss a strategy for simulating errors on very large designs.

4.1 Fidelity-Based Error Estimates

To begin our exploration of fault simulations, we introduce an important measure called *fidelity* for quantifying the difference between two quantum states. This is the simplest measure we use for evaluating error propagation in our quantum computer. We will then give an overview of how our fidelity calculation was used to estimate a communication network failure probability in [47]. We choose to start with a communication network, since the error generation and propagation is relatively simple compared to arbitrary circuits that use error correction. While this method of error probability calculation is limited to a specific set of cases that have minimal qubit entanglement, it is useful for quickly calculating error probability for communication. This method will be used in conjunction with other, more versatile methods explained later in this chapter.

4.1.1 Overview

Fidelity is a measure of the difference between two quantum bit vectors. Because of quantum entanglement, each of the 2^n combinations of bits in a vector of size n are physically separate states. For a given problem, one particular vector is considered a reference state that other vectors are compared against. For example, if we start with a superposition of bit vector $[0000 + 1111]$, and we send the bits through a noisy channel in which bit 3 is flipped with probability p, we will end up with a probabilistic vector of $((1-p)[0000 + 1111] + p[0010 + 1101])$. The fidelity of the final state in relation to the starting ("error-free") state is just $1 - p$. So, in the case of an operational state vs. a reference "correct" state, the fidelity describes the amount of error introduced by the system on the operational state [70]. A fidelity of 1 indicates that the system is definitely in the reference state. A fidelity of 0 indicates that the system has no overlap with the reference state.

We characterize errors by calculating the fidelity of qubits traversing the various quantum channels and gates necessary to move qubits around a communication network and apply gate operations. We will combine models of the individual communication and gate components so that we get an overall *complete communication fidelity* as a function of distance and architecture.

In general, we estimate the fidelity of a quantum state as it evolves through a fault

filled operation sequence by this basic equation:

$$F_{final} = \prod_{e}(1 - p_e)F_0 \tag{4.1}$$

where p_e is the probability of an error for each possible faulty event in the circuit .

4.1.2 Communication Fidelity Model

In ballistic movement, the fidelity of a bit after going through the ballistic channel over D segments is:

$$F_{new} = F_{old}\prod_{i=1}^{D}(1 - p_i \tag{4.2}$$

p_i is the probability of movement error along segment i. We will discuss D and p_i later in Section 4.8.

The fidelity of a qubit teleportation is more complicated because it involves a combination of single (p_{1q}) and double qubit gates (p_{2q}) and qubit measurement (p_{ms}) [31]:

$$F_{new} = \frac{1}{4}\left(1 + 3(1 - p_{1q})(1 - p_{2q})\frac{(4(1 - p_{ms})^2 - 1)}{3} \times \frac{(4F_{old} - 1)(4F_{EPR} - 1)}{9}\right) \tag{4.3}$$

The fidelity after a teleportation involves the fidelity of the data before teleportation (F_{old}) and the fidelity of the EPR pair used to perform the teleportation (F_{EPR}).

Although ballistic movement error does not appear in this formula, it should be mentioned that the fidelity of the EPR pair will be degraded while being distributed to the endpoints of the teleportation channel. Thus, even though the qubit undergoing teleportation incurs no error from direct ballistic movement, there is still fidelity degradation due to EPR pair distribution.

We produce EPR pairs from two qubits initialized to the zero state using a few single and double qubit gates. The fidelity of an EPR pair immediately after generation is:

$$F_{gen} \propto (1 - p_{1q})(1 - p_{2q})F_{zero} \tag{4.4}$$

F_{zero} is the fidelity of the starting zeroed qubits. Generation time involves one single and one double qubit gate.

Since we must also account for errors in idle qubits, if we assume that EPR pairs are already located at the endpoints of our channel, the teleportation time has the form:

$$t_{teleport} = 2t_{1q} + t_{2q} + t_{ms} + t_{classical\ bit\ mv} \times D \tag{4.5}$$

4.2 General Pauli Error Model

As mentioned in Section 1.2, we conceptually divide quantum circuit and layout errors into three types:

Gate errors occur on qubits undergoing a gate operation. Gate errors can be based on quantum operation type or physical location. Additionally, gates are the only places where we model qubit interaction, where an error on one qubit could affect another qubit.

Movement errors occur on qubits that are moved ballistically between gate locations.

Idle errors occur on qubits that are not doing anything due to decoherence through relaxation.

We choose not to model errors from inter-qubit coupling except in the case of multi-qubit gates, thus movement and idle errors only occur on single qubits, and gate errors only occur on either one or two qubits. For all the simulations and failure probability estimates in the rest of this work, we assume all errors are uniformly depolarizing. This is consistent with other studies and is also conservative compared to biased error models [6]. A depolarizing error introduces X, Y, and Z errors with equal probability, so if p_{error} is the probability that *some* error occurs on a qubit ρ_q:

$$\mathcal{E}_{1qubit}(\rho_q) = (1 - p_{error})\rho_q + \frac{p_{error}}{3}(X\rho_q X + Y\rho_q Y + Z\rho_q Z) \tag{4.6}$$

Where $\mathcal{E}$ is an error operator on the qubit state ρ_q and each error (X, Y, Z) is equally likely with probability $p_{error}/3$. In the coupled 2 qubit error case for gate operations like CNOTs, the operator is as follows:

$$\begin{aligned}\mathcal{E}_{2qubit}(\rho_{q12}) = (1 - p_{error})\rho_{q12} + \\ \frac{p_{error}}{15}(XI\rho_{q12}XI + YI\rho_{q12}YI + ZI\rho_{q12}ZI + \\ IX\rho_{q12}IX + XX\rho_{q12}XX + YX\rho_{q12}YX + ZX\rho_{q12}ZX + \\ IY\rho_{q12}IY + XY\rho_{q12}XY + YY\rho_{q12}YY + ZY\rho_{q12}ZY + \\ IZ\rho_{q12}IZ + XZ\rho_{q12}XZ + YZ\rho_{q12}YZ + ZZ\rho_{q12}ZZ)\end{aligned} \tag{4.7}$$

Gate name	Fault propagation rule
h	Transforms an X error to Z and vice versa
phase and T	Transform an X error to a Y error, no effect on Z errors
cx (CNOT)	Z Pauli errors propagate from target to source, X Pauli errors propagate from source to target.
cz	X errors on either qubit propagate to Z errors on the other qubit, Z errors do not propagate.

Table 4.1: Effect of gates on qubit errors

ρ_{q12} represents the joint error state of both qubits subjected to the error. p_{error} does not need to be constant across all error types; in most cases throughout this work, the basic movement, idle and gate error rates will all be different. We denote these as p_{gate}, p_{move} and p_{idle} and will discuss in Section 4.8 more details on actual values of these probabilities. In some cases we allow qubit gate errors to be parameterized with 3 different probabilities: $p_{measure}$ (for measurement gates), p_{1gate} (for non-measurement single qubit gates), and p_{2gate} (for 2 qubit gates). Previous work has pointed out that measurement gates may have different error rates due to different physical processes [91].

4.2.1 Gate Error Propagation Model

In addition to introducing new errors on qubits, we must also propagate existing errors between qubits in a 2 qubit gate interaction. This is important since quantum error correcting codes are designed to handle independent errors on the physical qubits making up an encoding. Any qubit that interacts with a qubit that experienced a fault is now dependent on that erroneous state and therefore must also be counted as erroneous. Additionally, certain gates actually transform Pauli errors of one type to another type. For example, the Hadamard (H) gate transforms phase (Z) information to bit value (X) information and vice versa.

From [1], we know how different types of errors flow from one qubit to another depending on the gate type. Table 4.2.1 summarizes these rules. For a review of the QASM instruction set in general, Section 2.1 has a detailed listing of all instructions. Any gates that are not listed in Table 4.2.1 are assumed to directly pass all error types from input to

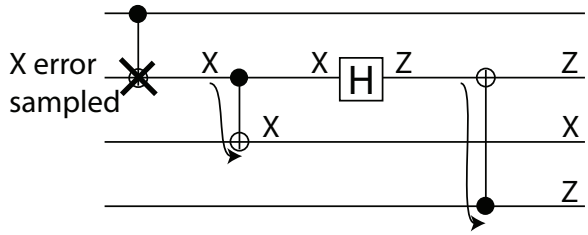


Figure 4.2: This circuit shows a single X error on the target qubit operated on by the first CNOT gate. That X error propagates from the control to the target qubit in the second CNOT gate. Next, the X error is transformed into a Z gate by the Hadamard gate. Finally, the Z error propagates from the target to the source in the last CNOT gate.

output unmodified.

In some cases only certain errors are propagated in certain directions. Figure 4.2 shows an example propagation through a circuit. We will obey these fault propagation rules in the simulation techniques we will present later in this chapter.

4.3 Errors and Classical Information

The incidence and propagation of faults in the typical set of quantum gates was handled in Section 4.2, but there are more operations we must handle. There are a number of gate-like operations done in a fault tolerant quantum computer that do not fit under the category of purely-quantum gates. These involve classical information in one manner or another. We specifically note 3 types of classical-bit-involving operations that we must handle specially in our models: qubit measurement, predicated quantum operations, and qubit corrections. We now present our own modeling techniques for tracking correlations between classical bits and the error states on qubits.

To present the mechanics of these specially handled instructions, we will be using the QASM notation presented earlier in Section 2.1.

4.3.1 Qubit Measurement

Measurement is essentially a transfer of quantum to classical information. In order to properly model a faulty classical result, we should have the probability distribution over the qubit values as well as the associated error probability distribution. Then, we can

simulate the collapse of the qubit state superposition and transfer the "measured" value to the classical bit. Since we are tracking only the error distribution and not the full quantum state space, we cannot do this and instead approximate.

If we look closer at where classical data is used in all the circuits we are interested in, we note that they are only used as error syndromes in fault tolerant verification and correction procedures. This means that the error information is more important than quantum state for transferral to the classical bits. This is ideal because the error state is the only thing we are tracking. Here is an example of a QASM measurement instruction:

```
xmeasure c0, q0;
zmeasure c1, q1;
```

So in our model, a Z error on qubit `q0` would set `c0` to 1 and a X error on qubit `q1` would set `c1` to 1.

Additionally, we can model errors in the measurement operation by correctly transferring error from the qubit to the classical bit with some probability and flipping the classical bit with an error probability. So in the above example, if `q0` has a Z error, the error information transfers to `c0` with a probability of $(1 - p_{gate})$ and sets the opposite value with a probability p_{gate}. This action corresponds to the measurement process producing a mistaken reading of the qubit state, thus giving the opposite classical bit value.

4.3.2 Qubit Corrections

Figure 4.3 shows the error correction procedure we use for all of our fault tolerant circuits in our simulations. There are two separate phases, Z error correction and X error correction.

In the first CNOT gate, Z errors flow from the target (data) to the control (az). The `xmeasure` operation measures the Z error syndrome and then the classical syndrome outcome is sent to the zcorrect block. This block is made up of two parts, a physical model of the actual phase flipping based on the classical syndrome and a virtual `zcorrect` instruction that tells the error simulator to update the internal error state of the `data` qubit:

```
# virtual instruction
zcorrect cz_1, ..., cz_n, data_1, ..., data_n : t;

# physical gates
(@cz_1 == 1) z data_1;
```

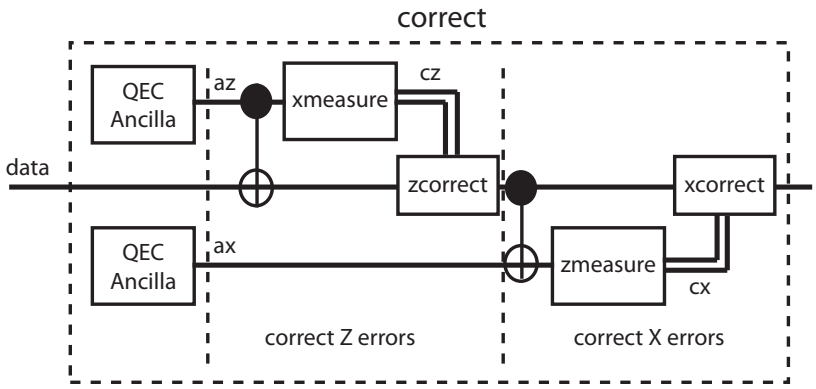


Figure 4.3: Schematic of the error correction procedure operation. Each logical "correct" operation is replaced with the above circuit. Correction of X and Z errors are done in two separate phases.

```
(@cz_2 == 1) z data_2;
...
(@cz_n == 1) z data_n;
```

Where ax, az, cx, cz, and data are all encoded blocks of physical qubits or classical bits, each addressed by the above notation "encodedname_bitindex".

The X error correction case is very similar. In the second CNOT, X errors flow from control (data) to target (ax) and the `zmeasure` gate then measures the X errors and produces the classical syndrome. This is passed to the xcorrect block that looks like:

```
# virtual instruction
xcorrect cx_1, ..., cx_n, data_1, ..., data_n : t;

# physical gates
(@cx_1 == 1) x data_1;
(@cx_2 == 1) x data_2;
...
(@cx_n == 1) x data_n;
```

In order to correct a logical block of qubits, we effectively reset the errors on them as long as there are not more errors than the code we are using can handle. Our correction operations take equal numbers of qubits and classical bits as input. Each classical bit corresponds to the error on the qubit in order, so in this example `cx_1` indicates a X error on qubit `data_1` and `cz_1` indicates a Z error on `data_1`. The `t` at the end of the instructions indicate that each instruction can correct up to 1 error of each type in this code.

Correction type	Qubit errors	Clabit value	Count error?
xcorrect	X: no Z: don't care	0	no
	X: yes Z: don't care	0	yes
	X: no Z: don't care	1	yes
	X: yes Z: don't care	1	yes
zcorrect	X: don't care Z: no	0	no
	X: don't care Z: yes	0	yes
	X: don't care Z: no	1	yes
	X: don't care Z: yes	1	yes

Table 4.2: All the possible error conditions for X and Z correction virtual instructions. Each qubit maintains whether it is in an X and/or Z error state. In some of the above cases, the error state of one of these types does not matter since the operation only accounts for the other type.

The `x/zcorrect` operations effectively count the number of "errors" of that type that are present in the given qubits. If that number exceeds the number of correctable errors, then no action is taken (no correction). Table 4.2 lists all possible combinations of a qubit error state and its associated classical syndrome bit, and what is counted as an error.

The actual predicated physical bit or phase flips are separate from the operations performed by the virtual instructions described in Table 4.2 and will be explained further in the next section. Rules we have defined in this table are conservative because we consider either a qubit having an error or an error in syndrome measurement as a total failure on that qubit. Second, we assume `t` is the maximum number of errors correctable regardless of the pattern of errors on the qubits, in some cases, some codes may be able to correct certain error patterns of more than `t` errors.

4.3.3 Predicated Quantum Operations

As noted above, we only used classical predication for error correction and verification operations. Therefore, it is sufficient that the classical bit only signals the existence of the measured type of error on the qubit. To revisit the example of the physical xcorrect operation above, the physical correction operations would look like this:

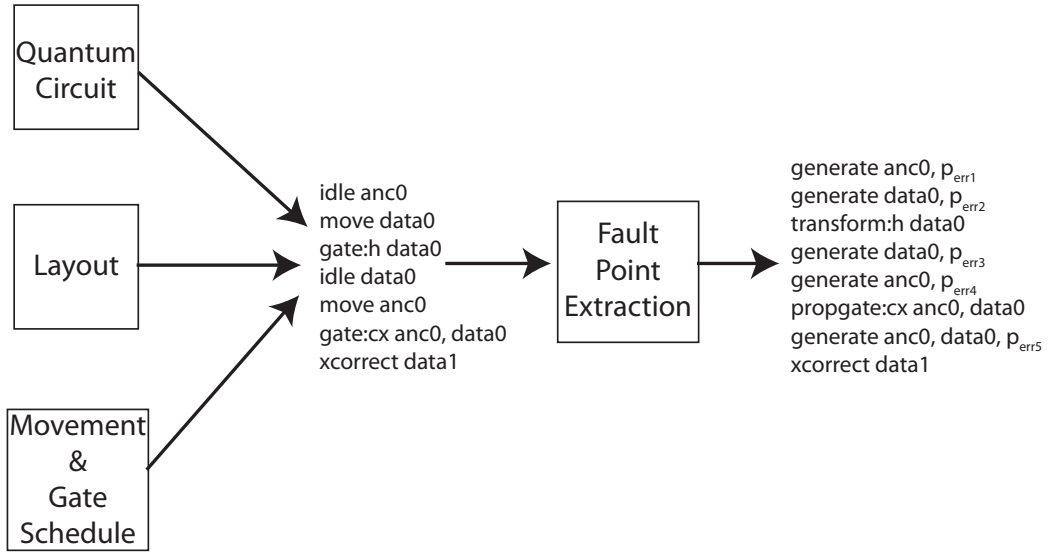


Figure 4.4: Producing a fault point stream. The schedule, layout and circuit produce a series of movement, idle and gate events. The fault point extractor produces a stream of error probabilities on specific qubits.

```
(@cx_1==1) x data_1;
(@cx_2==1) x data_2;
(@cx_3==1) x data_3;
```

As mentioned in Section 2.1, the `(@bit1,0)=` notation is a predicate, so the instruction is only executed when the classical bit `bit` has the designated value. In this case, the `x` and `z` instructions are conditionally fixing the error on that qubit in the encoding. This models the physical quantum operation that would actually correct the error. Even though we do all the work in updating the error state on the qubit in our virtual `x/zcorrect` instructions, these bit/phase flip instructions are necessary to model errors which could occur in the correction procedure itself.

4.4 Fault Point Streams

So far we have discussed types of errors and introduced how errors propagate through a given circuit or layout. Before we discuss simulation of these errors we need a more concrete description of what exactly our simulations are simulating. The basic process for producing a *fault point stream* is shown in Figure 4.4.

Event type	Description
generate: $q1 \ldots qn\ p$	Error is generated on qubit(s) $q1$ through qn with probability p
transform:*gate* q	Transform errors on q according to rules for gate *gate*
propagate:*gate* $q1\ q2$	Propagate and transform errors between qubits $q1$ and $q2$ based on the error propagation rules for gate *gate*
transfer:*gate* $q\ c$	Transfer errors on q to the classical value on c according to rules for gate *gate*
xcorrect: $q1 \ldots qn, c1 \ldots cn, t$	Correct X errors on encoded qubit block $q1 - qn$ based on syndrome bits $c1 - cn$ and number of correctable errors t
zcorrect: $q1 \ldots qn, c1 \ldots cn, t$	Correct Z errors on encoded qubit block $q1 - qn$ based on syndrome bits $c1 - cn$ and number of correctable errors t

Table 4.3: Error events produced by fault point extractor.

Using a combination of layout, scheduling, and circuit specifications, we get a list of events in the run of the circuit on the layout where errors could occur. The fault point extraction step takes these events and produces a simple stream of error-centric events upon which errors are produced and propagated. Table 4.3 summarizes these events.

The actual error extraction procedure is discussed in more detail later in Sections 4.8 and 4.9, but for now we will assume this fault point stream is available. The remaining error simulation methods in this chapter will all use this fault point stream in order to track and generate error throughout the run of the circuit.

With our assumed underlying fault model explained, we now move on to ways of efficiently simulating fault events and propagation through a circuit.

Operation	Description
$applyErrorMap(QubitSet\ qs, ErrorMap\ e)$	Transform error states in qs using error map e. The error map specifies new error events that occur on a subset of qubits in qs and the probabilities of each new event.
$applyCorrectMap(QubitSet\ qs, CorrectionMap\ e)$	Transform error states in qs to a corrected state as dictated by the correction map.
$merge(QubitSet\ qs1, QubitSet\ qs2)$	Take two QubitSets and create a merged QubitSet consisting of a Cartesian product of the two probability spaces.
$split(QubitSet\ qs, Qubit\ q)$	Split qubit q off of qs, computing the marginal probabilities of qubit states in qs without q.

Table 4.4: List of operations in Chi's QubitSet [21] failure probability model.

4.5 Joint and Marginal Probabilities of Failure

Using the error event stream defined in Section 4.4, we will directly compute the joint probability distribution of all the qubit errors in the system. We will review the work already done on this by Chi et al [21] and discuss the time and space limitations to the original proposal. To address these limitations, suggest performance improvements and why we eventually abandoned this technique as a whole due to a poor accuracy-performance tradeoff.

4.5.1 Previous Work: QubitSets

As a starting point for a deterministic probability model, we look at the notion of QubitSets by Chi et al [21]. The idea behind this data structure is to track the joint probabilities over the possible error combinations in a group of qubits. This is potentially much more accurate than tracking only individual qubit fidelities, as done in Section 4.1, since it captures important qubit correlations. In fact, proper fault tolerant error correction procedures cannot be simulated at all with the simple fidelity model, since the procedures rely on the correlated error transfer from the data to be corrected onto ancilla bits. The fidelity-only model does not account for any correlations. This is why we only investigated the fidelity model for simple error processes in communication and EPR creation and purification.

We now summarize the rest of Chi's QubitSet model, before presenting it on a more rigorous footing and discussing extensions. A QubitSet represents all the error probabilities on a set of qubits. Table 4.4 summarizes the basic operations used to track errors through a circuit with a QubitSet. For each new operation in a circuit, the QubitSet model first

merges together the disjoint sets containing all the qubits involved in the operation (if they are in separate sets), to create a new joint set. It then applies either $applyErrorMap$ if the operation is a normal error-inducing one, or $applyCorrectMap$ if the operation is a correction. They additionally propose a few optimizations to reduce the state space tracked.

- If the operation is a measurement, *split* is applied to remove the measured qubits, since we will no longer use their state.
- When qubits are merged, if a merged error state has a probability below a constant threshold, it is excluded from the merged set.

The *split* operation offers some modest savings in resources by effectively providing garbage collection to the QubitSet scheme, but the physical qubits that make up logical data qubits have long running state that is never measured until the end of the circuit. The resources used by these states will still dominate with an exponential space and time requirement. The second optimization, which prunes out low probability events works pretty well in Chi's empirical results, but their test circuits had relatively shallow depth. With a longer running circuit, their method of discarding low probability events will, in many cases, lead to a QubitSet with no probability. Additionally, Chi's "preservation" method could underestimate the failure probability by an unbounded amount, a problem previously mentioned in [77].

4.5.2 Joint Probability Evolution

Chi et al gave a general algorithmic description of his QubitSet technique but did not provide a description in a mathematical context. We have done so here to aid in better specifying our own implementation of this technique.

To define the joint probability distribution of errors over all qubits we start with $E_i \in \{X, Y, Z, I\}$, which we define to be the random variable representing the error on qubit qi. The set of all errors sampled at a particular time step t would be $\{E_1, E_2, ..., E_n\}_t$ where n is the total number of qubits in the system at that time. Thus, the set of errors due to gate, movement and idle qubit operations that occur throughout the circuit is the time series:

$$\mathcal{E} = \{\{E_1, E_2, ..., E_n\}_t, t \in T\} \tag{4.8}$$

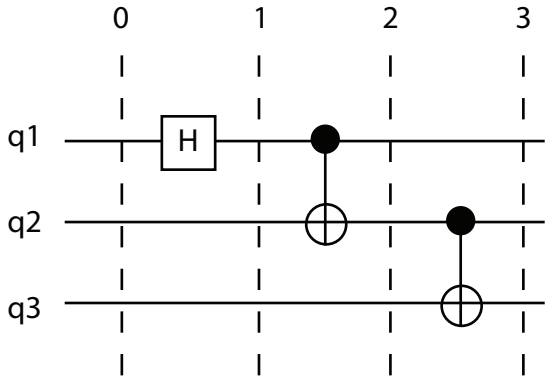


Figure 4.5: Computing the probability over a 3 qubit cat state ($p(e = X) = 0.1, p(e = I) = 0.9$, only accounting for gate errors): At time step $t = 0$, there are no errors on any qubits so $p(f_i = I) = 1.0$. At $t = 1$: $p(f_1 = I) = 0.9, p(f_1 = X) = 0.1, p(f_2 = I) = 1.0, p(f_3 = I) = 1.0$. The first CNOT propagates the error from qubit 1 to qubit 2 so they are not independent anymore. At $t = 2$: $p(f_1 = I, f_2 = I) = 0.81, p(IX) = 0.030, p(XI) = 0.030, p(XX) = 0.13, p(f_3 = I) = 1.0$. Finally, at $t = 3$, the error probability must be expressed as the joint over all qubits: $p(III) = 0.73, ..., p(XXX) = 0.16$. Note that even though $q1$ and $q3$ did not directly interact, they are both conditional on $q2$.

Where the probability $p(E_i = e_i)$ at time t is defined by the operation error probabilities from Section 4.2. We can also track the current set of faults on each qubit at time t as $\mathcal{F} = \{\{F_1, F_2, ..., F_n\}_t, t \in T\}$. While $\mathcal{E}$ is a stationary process, $\mathcal{F}$ is not. As mentioned in Section 4.2, inter-operation correlations that are not mediated by the qubit state itself and therefore model it as a Markov process with probability distribution:

$$P(\mathcal{F}(t)) = p(f_1, f_2, ..., f_n, t) \tag{4.9}$$

i.e. the probability at time t that qubit $q1$ has the error f_1, $q2$ has f_2, etc. This is a Markov process since $P(\mathcal{F}(t))$ is only dependent on $P(\mathcal{F}(t-1))$ and $P(\mathcal{E}(t))$.

The distribution $P(\mathcal{F})$ becomes difficult to compute as the number of qubits and depth of the circuit increases. The reason is that while the probability that each qubit has an error is initially independent of all other qubits, they become increasing conditional on one another.

At the beginning of a given circuit, the number of probabilities we must keep track of is $O(4n)$ for n qubits. If we assume that every qubit is indirectly affected by every other

qubit in a given circuit, the number of probabilities we need to track balloons to $O(4^n)$ for the joint probability distribution over all qubits. This is if we want the exact probability of error over all qubits at the end, but what if we are willing to accept an approximation?

4.5.3 Approximating Joint Probability

Calculating the full joint probability state over all the qubits in a non-trivial circuit requires space and time exponential in the number of qubits. We now focus on some of our own methods to reduce both the space and time resources we need to estimate the overall failure probabilities. We will find that while these methods seem attractive, they inevitably lead to unacceptable loss of accuracy.

Cached QubitSets

We call our first modification *cached QubitSets*. Many quantum circuits possess a significant amount of redundancy, especially in fault tolerant subcircuits. For example, preparing the verified ancillary state that is then interacted with data for error correction is always the same for a given code (at least each verified round). Instead of recomputing the QubitSets for the identical ancilla states each time, we can just cache the QubitSets that are output from the ancilla subcircuits. This approach is easily implementable in the context of the Quadence CAD flow since we represent our quantum circuits in as a modular hierarchy as described in Section 2.1. We can substitute the simulation of a circuit module with the cached QubitSet.

There are several ways we can perform this caching:

Explicit We can explicitly mark a module as "cached" in the qasm specification and the program then will compute the QubitSet on the first invocation and use it throughout the rest of the program.

Source-Only If we have a module that is a source in the dataflow graph, i.e. an ancilla factory, we can automatically cache the resulting QubitSet(s), since there is no input QubitSets that could affect the output.

Conditional For non-source modules, we can cache QubitSets predicated on the QubitSet(s) of the input qubits to the module.

The third method for caching sets would require an even more prohibitive amount of memory needed to cache both the input and output QubitSets for modules. We consider only the first two as viable options.

Low Probability Hamming Distance

Instead of using the techniques outlined in Section 4.5.1 to throw out low probability events, we can make a more conservative approximation and merge low probability error events with the event of the closest Hamming distance. Additionally, we can prefer to merge with events with a larger Hamming weight to make our estimate pessimistic.

In practice, we find that this method does not sufficiently combat the exponential increase in the state space without spending exponential time to investigate and merge error events based on Hamming distance.

Big Set Splitting

Instead of allowing the joint error probability model to grow as more qubits interact and become dependent on one another, we periodically factor large joint distributions into several marginal distributions. We decide the sets of qubits to factor into based on an LRU policy, grouping qubits that have interacted most recently into the same set.

Cached Hierarchical Fidelities

The last approach potentially lacks the most accuracy. Instead of computing QubitSets for all the physical qubits in a system, we can compute QubitSets for lower level modules but instead of caching the output as QubitSets, we can compute a simple fidelity of the operation and then just compute QubitSets of logical qubits using the module fidelities to compute logical gate errors at a higher level.

This particular approach has the problem in that a correction operation in a fault tolerant circuit effectively has a fidelity greater than 1. Setting the fidelity to such a value, give us some sort of approximation although it is not immediately obvious how much accuracy is sacrificed since the amount that a correction operation reduces an encoded qubits failure probability is not necessary linear as a greater-than-one fidelity implies.

In practice, the use of QubitSets can only be confined to use in situations of small numbers of qubits. We have implemented a very efficient bit vector based version of the

QubitSet calculation with the promising Big set splitting optimization and have witnessed the following problem: the number of qubit dependencies that must be preserved for the bare minimum level of accuracy is greater than the space available in most cases.

Consider the follow case: a single logical cnot gate encoded in the [[7,1,3]] Steane code, followed by an error correction on the two logical qubits. First, the two QubitSets for the logical qubits are merged due to the interacting encoded cnot, this gives us a set of size 14. Next, we perform error correction on each logical qubit, requiring at least one encoded ancilla qubit to become dependent on each of the logical qubits to be corrected. This comes out to 28 qubits. Since each operation introduces some amount of error to all qubits involved, it is not unreasonable to expect the number of error vectors to approach 2^{28}. In addition to the 28×2 bits (7 bytes) needed to represent error state, we also need a 8 byte floating point number for the probability. This comes out to about 2^{32} bytes, already a gigabyte to represent 2 entangled, corrected, logical bits. This means we are extremely limited in representing multiple logical entangled qubits. Also, this example used a code that is almost the smallest, if we were to calculate the joint probability for a qubit encoded in the [[81,1,9]] code, it would be prohibitive to represent even one encoded qubit. Additionally, more than one encoded ancilla qubit is used in the correction steps, thus the above minimal estimate is probably too conservative. In one measurement, our exact joint probability calculation revealed an error probability of 0.75, whereas, with big-set splitting, the error probability was approximately 10^{-20}! Another possible argument is that our splitting heuristic is not optimal, but in light of the previous example, we believe that any splitting technique aggressive enough to combat the exponential state space growth will sacrifice too much accuracy and not be a very useful estimate.

For this reason, we opt to use only the exact QubitSets without our investigated optimizations, and then only to validate our other error simulation methods for correctness on small example circuits. Later in this chapter we will investigate other improvements in error simulation runtime that do not sacrifice as much accuracy.

4.6 Monte Carlo Simulation

Using our error event stream from Section 4.4, another failure probability estimation technique is to perform a Monte Carlo simulation over this event stream. Figure 4.6 shows an example simulation process through a circuit. We run through the error event

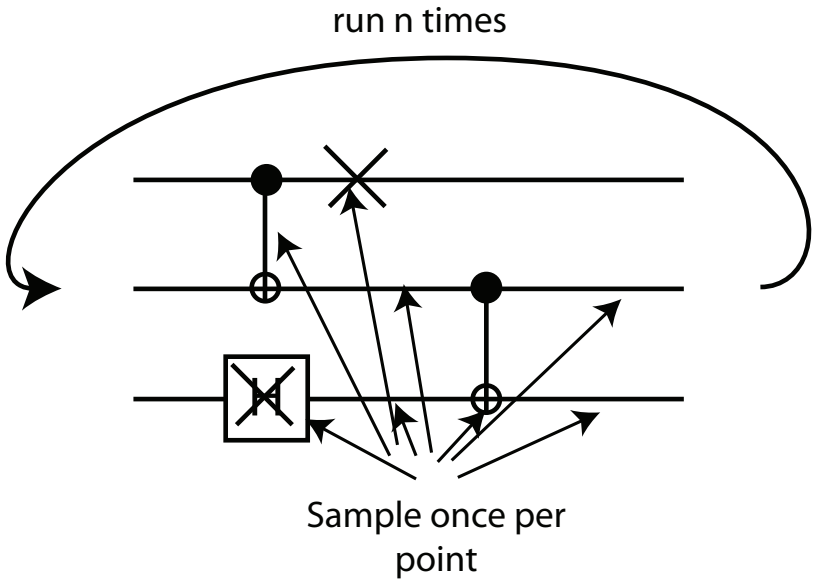


Figure 4.6: Simulating error propagation in quantum circuits. Our Monte Carlo simulation method traverses the circuit n times in order to measure n successes or failures to compute overall success probability.

stream n times, sampling errors at all fault points. For each iteration, we check the output encoded qubits for a set of errors that cannot be corrected by the error correcting code used. If such a set is detected, we have a failure, if not, a success. The final probability of success the then just: $p_{success} = successes/(failures + successes)$. For each *generate* event, we sample errors on qubits. Errors are propagated and removed as well from their respective events. The basic algorithm our Monte Carlo simulator takes is the following:

We assume our errors to be all of the unbiased depolarizing channel type, as outlined in Section 4.2. Random numbers are sampled with a Mersenne prime random number generator rngpack from the Colt Java library [46] which is known for its long period and thus makes it good for sampling large numbers or errors when simulating a large circuit. Error propagate through a circuit according to the rules for simulating circuits given in [1] and from Section 4.2.

Table 4.5 give a summary of the necessary actions our Monte Carlo simulator must take for each error event type. In our implementation, each qubit in the system maintains state as to whether it has an X and/or Z error or not. The Monte Carlo simulator just updates these qubit states according to Table 4.5. While this simulation method is not

Algorithm 3 Monte Carlo error simulation

```
events ⇐ extracted error operators from layout
initialize qubitStates to no errors
failures ⇐ 0
successes ⇐ 0
while not enough runs for statistical significance do
  for event in events do
    if event is gate then
      check to see if errors need propagating
    else if event is correction then
      if encoded qubit can be corrected then
        update qubitStates to remove errors from correct  qubits
      else if qubit is output data then
        failures++
        halt iteration
      end if
    end if
    error ← randomly select error (or no error)
    if error indicates an error then
      add error to qubit(s)
    end if
  end for
  if finished all events then
    successes++
  end if
end while
return failures/(failures + successes)
```

Event type	Action
generate: $q1 \ldots qn\ p$	Use a random number generator to sample whether any error occurs on qubit q, according to error probability p. If so, use another random number to determine the Pauli error type(s) on the qubit(s).
transform:*gate* q	Transfer X error state to Z and vice versa, based on gate type
transfer:*gate* $q\ c$	If X or Z error state on q is positive, set the classical value on c to one, according to rules for gate *gate* (zmeasure transfers X errors, xmeasure transfers Z errors)
propagate:*gate* $q1\ q2$	Propagate and transform X and Z error states between qubits $q1$ and $q2$ based on the error propagation rules for gate *gate*
xcorrect: $q1 \ldots qn, c1 \ldots cn, t$	Reset X error state on each qubit $q1 - qn$ based on its associated syndrome bit $c1 - cn$ as long as there are fewer than t X errors on all the qubit/syndrome pairs
zcorrect: $q1 \ldots qn, c1 \ldots cn, t$	Reset X error state on each qubit $q1 - qn$ based on its associated syndrome bit $c1 - cn$ as long as there are fewer than t Z errors on all the qubit/syndrome pairs

Table 4.5: Actions taken by Monte Carlo simulator for each error event type.

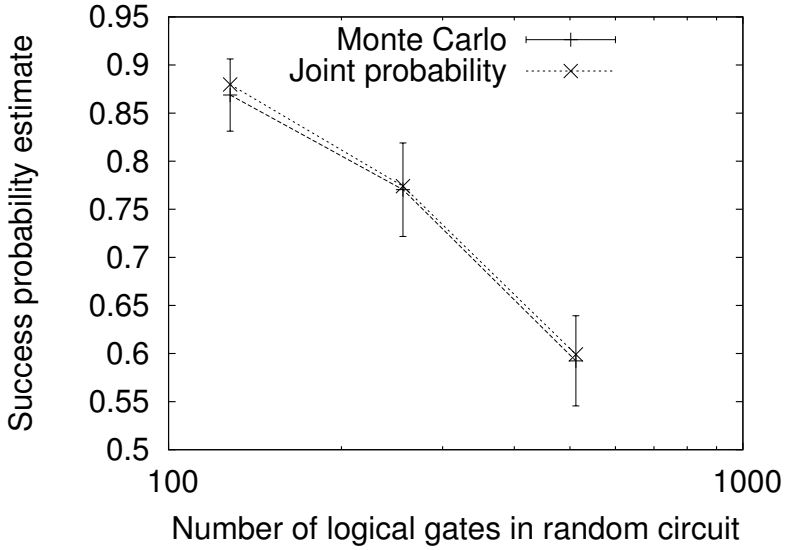


Figure 4.7: Comparison of success probability estimates from the Monte Carlo simulation and the full joint probability calculation estimation methods. 95% confidence intervals are included for the non-deterministic MC simulation. Each random circuit is encoded with the 7 bit Steane code.

novel in itself, we have used it to more completely compare a set of quantum error correcting codes than the studies in [90] and [25]. Using the communication estimation techniques in Chapter 3, we sample not only gate errors but also qubit movement and idle from various communication scenarios. We will also discuss in Section 4.7 our novel enhancements on this technique.

4.6.1 Performance

Having two very different ways of modeling errors allows us to validate them against each other to give us more confidence the results are correct.

Figure 4.7 shows that the estimated success probabilities for the two methods closely match for a variety of random circuits. The Monte Carlo method used 100, 200, and 300 iterations respectively for each point to generate the probability estimates. There are no confidence intervals for the joint probability calculations since the calculation is deterministic.

Implementing these two error estimation techniques exposed mistakes in both im-

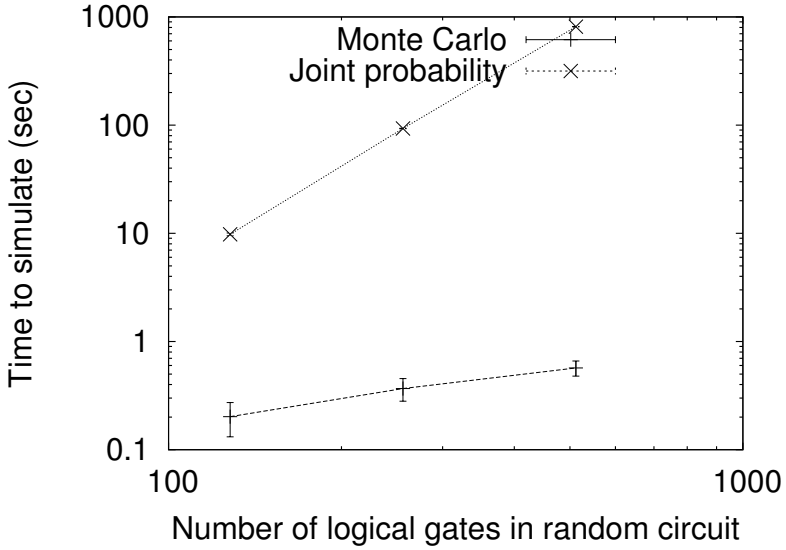


Figure 4.8: Runtime comparison of our Monte Carlo simulation and full joint probability calculation estimation methods. Each random circuit is encoded with the 7 bit Steane code.

plementations. As these bugs were fixed, the two techniques converged on the same probability. We have also compared the estimates generated from our Monte Carlo simulation to the results in Cross et al. [25] and found our pseudo-threshold estimates to be within 2x of theirs for a variety of codes. This is an especially appropriate comparison since we use Cross' ftqctools suite [24] to generate our encoding and correction circuits. We are not expecting to exactly match their estimates, since their use an adversarial noise model but the relative error pseudo-thresholds from our Monte Carlo simulator have very similar relative differences between codes as their results report.

Figure 4.8 shows the exponential time complexity in the number of gates/qubits simulated [1] of our implementation of the QubitSet joint probability calculation. We compare this to our relatively efficient Monte Carlo simulator which shows only a polynomial runtime increase. Due to the prohibitive runtimes of the QubitSet method, we opt to focus on using and comparing Monte Carlo methods for the rest of this work.

[1] Qubits increase proportionately to gates in these random circuits

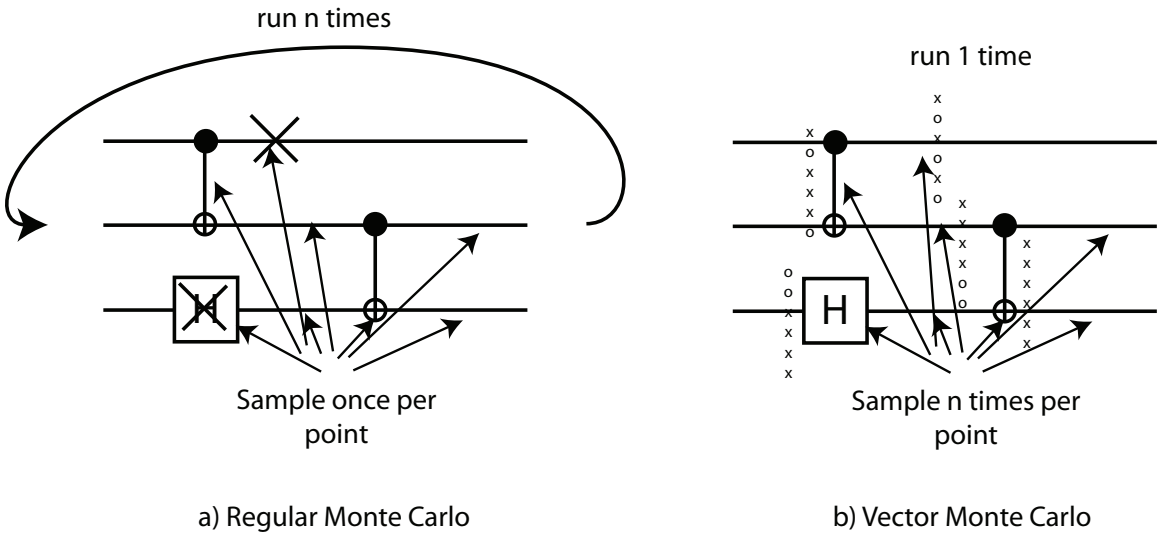


Figure 4.9: Simulating error propagation in quantum circuits. Our regular Monte Carlo simulation method traverses the circuit n times in order to measure n successes or failures to compute overall success probability. Vector Monte Carlo traverses the circuit once but for each error event, it generates a vector of n error scenarios.

4.7 Vectorized Monte Carlo

In profiling our Monte Carlo implementation from Section 4.6, we notice that the majority of the time spent simulating fault incidence and propagation is on traversal of the error event list and accessing qubit error state. Sampling random numbers takes up about 0.7% of the runtime and the actual error calculation logic is approximately 5% of the runtime. So about 95% of the runtime is "wasted" on traversal of the error event list, retrieving error information on qubits, etc.

Furthermore, we note that since our control flow model is just predicated execution, we evaluate each instruction in the same sequence. Instead of repeated traversal of the same instruction sequence, we can traverse it once and do all the sampling of error events in one pass.

We call this technique *vector Monte Carlo* or *vectorMC*. Figure 4.9 illustrates the differences between our two different Monte Carlo simulation methods. Instead of keeping information of a single occurrence of X and Z errors for each qubit, we keep a vector of error occurrences. Each error type is represented by a bit vector and each gate, move, and

Event type	Operation	Operator
Generate	(Pred)Err(Q1)	$X_{Q1} = (X_{Q1} \cdot Pred) + ((X_{Q1} + Err_X) \cdot Pred)$ $Z_{Q1} = (Z_{Q1} \cdot \bar{Pred}) + ((Z_{Q1} + Err_Z) \cdot Pred)$
	(Pred)Err(Q1, Q2)	$(Err_X, Err_Z) \in (I, X, Y, Z \otimes I, X, Y, Z) - II$ $X_{Q1} = (X_{Q1} \cdot \bar{Pred}) + ((X_{Q1} + Err_X) \cdot Pred)$ $Z_{Q1} = (Z_{Q1} \cdot \bar{Pred}) + ((Z_{Q1} + Err_Z) \cdot Pred)$
Propagate	(Pred)CX(QC, QT)	$X_{QT} = X_{QT} + (X_{QC} \cdot Pred)$ $Z_{QC} = Z_{QC} + (Z_{QT} \cdot Pred)$
	(Pred)CZ(QC, QT)	$Z_{QT} = Z_{QT} + (X_{QC} \cdot Pred)$ $Z_{QC} = Z_{QC} + (X_{QT} \cdot Pred)$
	(Pred)H(Q1)	$Z_{Q1} = (Z_{Q1} \cdot \bar{Pred}) + (X_{Q1} \cdot Pred)$ $X_{Q1} = (X_{Q1} \cdot \bar{Pred}) + (Z_{Q1} \cdot Pred)$
Transfer	(Pred)XMEASURE(Q1,C1)	$D_{C1} = (Z_{Q1} xor Err_Z) \cdot Pred$
	(Pred)ZMEASURE(Q1,C1)	$D_{C1} = (X_{Q1} xor Err_X) \cdot Pred$
Correct	(Pred)XCORRECT(C1, ..., CN, Q1, ..., QN)	$X_{Qi} = (X_{Qi} \cdot Pred) + ((Ci \cdot (Count(X_{Q1} ... X_{QN}) < \lfloor dist/2 \rfloor)) \cdot Pred)$
	(Pred)ZCORRECT(C1, ..., CN, Q1, ..., QN)	$Z_{Qi} = (Z_{Qi} \cdot Pred) + ((Ci \cdot (Count(Z_{Q1} ... Z_{QN}) < \lfloor dist/2 \rfloor)) \cdot Pred)$

Table 4.6: Boolean arithmetic rules over data and error vectors in the vectorMC error simulator. Note that the CORRECT operations are not strictly boolean arithmetic but instead require the counting of the number of true bits over a set and checking the sum against the distance of the code.

idle error incidence or propagation is implemented by bit vector logical operations. Table 4.7 shows a listing of all the necessary operations to implement the Monte Carlo simulation and the logical bit vector operations we use to implement them.

Figure 4.10 shows the accuracy and runtime for the vectorMC and standard Monte Carlo simulation methods as a function of the number of effective circuit simulation trials performed. Accuracy is plotted in terms of the confidence interval for the circuit success probability for a simulation with the given number of trials. The smaller the confidence interval, the more accurate the simulation is. As expected, the more trials we do for either method, the more accurate the result. In fact, the accuracy for a given number of trials is approximately the same for either method. This matches our intuition, since the error propagation and sampling rules for MC and vectorMC are identical.

If we look at the runtime, vectorMC is substantially faster than the standard Monte Carlo method of simulation for any number of iterations. For example, to simulate a circuit made up of 6 million gates, the our standard Monte Carlo technique takes 405 minutes while our vector simulation technique takes 32 minutes, more than a 10x improvement. This is as expected since the overhead of the circuit traversal is amortized as only one traversal is now needed. Looking at the output of a profiler, we see that doing a simulation with 1000

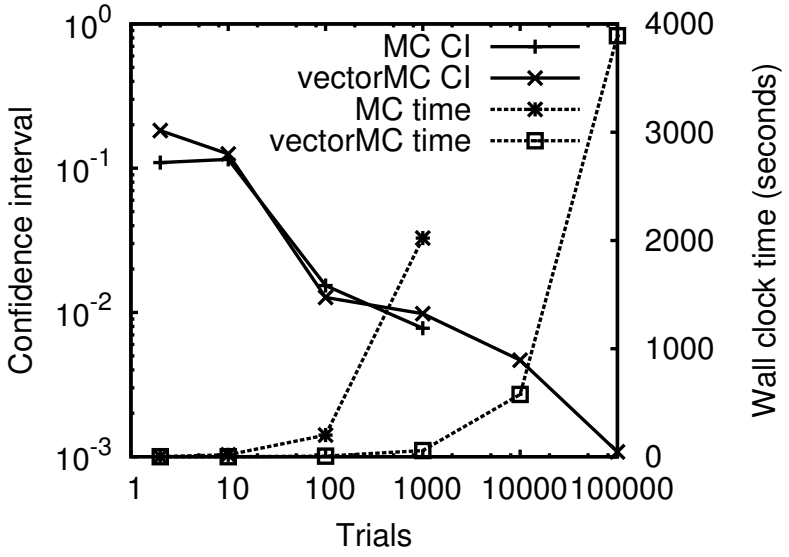


Figure 4.10: Runtime and accuracy of the vectorMC versus regular MC success probability estimation techniques. Accuracy is measured in terms of the average confidence interval for 10 random circuits simulated 20 times with n trials (n on the x-axis). The last two data points for the Monte Carlo simulation were not completed due to the runtime limit on the shared compute cluster used.

simultaneous trials, about 45% of the runtime is spent in the random number generator, compared to the less than 1% of the time used by the regular Monte Carlo method.

We can also look at the performance of the two Monte Carlo methods as a function of the number of error events processed. Figure 4.11 uses a constant number of trials to show accuracy and runtime variation with problem size. As expected, the confidence intervals increase for larger circuits, indicating decreasing accuracy as the number of different fault paths increases.

4.8 Ion Trap Movement and Idle Error Models

We discussed a fault point generator in Section 4.4 and defined a set of possible error events but did not specify how we determine those events from the layout and circuit information that is input to the fault simulator. Since we already introduced our gate error model in Sections 4.2 and 4.3, we do not need to add much more on this. We assume a

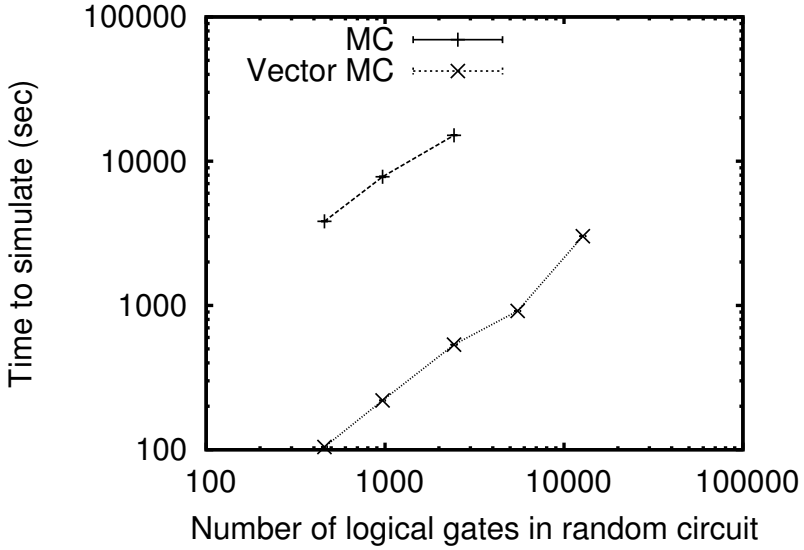


Figure 4.11: Runtime of the vector Monte Carlo error propagation simulation versus the original Monte Carlo implementation. The last two data points for the Monte Carlo simulation were not completed due to the runtime limit on the shared compute cluster used.

particular probability of a gate error, p_{gate} and that is all we need to specify for the *generate* events. The rules for error propagation, transformation and transfer in gates in also detailed in the same sections. Correction operations are also specified there. The remaining details we need to specify now concern the error probabilities for the generate events from qubit movement and idling. Our qubit movement error model is ion trap-dependent because there are many geometric specific details that impact ion trap movement.

4.8.1 Movement Error

In the case of communication (movement) error, we derive an error probability for each path an ion must travel between two successive gate locations. As shown in Figure 4.12, we break up probability events based on path *segment*. Segments are parts of the path in which the ion goes straight, though the same type of trap, or changes direction through a turn or intersection trap. The total path error probability is an aggregate of all the error probabilities along these segments. These probabilities are aggregated similar to what was done in Section 4.1.2:

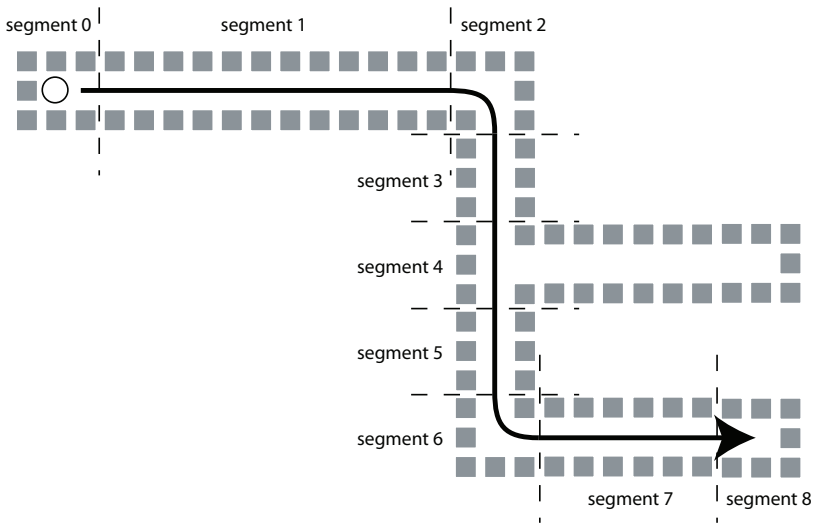


Figure 4.12: This shows a path of ion movement through ion traps that is broken into segments. We get the probability of error of the whole path, p_{path} by aggregating error probabilities along each segment.

$$(1 - p_{path}) = \prod_{i=1}^{N} (1 - p_i) \tag{4.10}$$

Where p_{path} is the error probability for the entire path and p_i is the error probability for just segment i.

$$(1 - p_{path}) = (1 - p_{start})(1 - p_{stop})(1 - p_{memory})^{t(0)}(1 - p_{memory})^{t(S)} \tag{4.11a}$$

$$\times \prod_{i=1}^{S-1} ((1 - p_{start})(1 - p_{stop})(1 - p_{memory}))^{t(i)} \tag{4.11b}$$

$$\times (1 - \bar{c}(i)p_{straight}(w(i), d(i), v(i), v(i+1))) \tag{4.11c}$$

$$\times (1 - c(i)p_{turn}(w(i))) \tag{4.11d}$$

(4.11a) corresponds to the starting and stopping of the ion at the source and destination gate locations, as well as the time they spend waiting around to do their operations. The product is then over the remaining segments. (4.11b) covers the intermediate starts and stops along the movement path. (4.11c) accounts for the movement of the ion along straight channels and (4.11d) accounts for ion turns. Note that the use of c and $\bar{c}$ assures that, for each segment, we only count error from either straight line or turn movement.

Variable name	Description
p_{path}	error probability along the whole path of movement
p_{start}	error probability when a qubit starts moving
p_{stop}	error probability when a qubit stops moving
p_{memory}	error probability of a stationary ion per unit time
$p_{turn}(w)$	error probability during an ion turning through a w-way intersection
$p_{straight}(w, d, v_i, v_f)$	error probability for an ion moving straight through d w-way intersections with a starting velocity of v_i and final velocity of v_f
S	number of segments in the ion path
$w(i)$	number of openings in segment i
$d(i)$	length of segment i
$t(i)$	time the ion is stationary in segment i, note that this time is in the same units as p_{memory}.
$v(i)$	velocity of the ion entering segment i
$c(i), \bar{c}(i)$	c is 1 or 0 depending on whether the ion changes direction or not in segment i. $\bar{c}$ is the opposite

Table 4.7: Description of all the parameters in Equation (4.11)

Assumptions:

- This model assumes that for a given segment, the ion acceleration is constant so the initial and final velocities completely describe the ion dynamics.
- We assume that memory errors from stationary ions are independent of the ion trap geometrical configuration.
- We assume that the ion always has the same velocity magnitude moving into and out of a turn.
- p_{path} is the probability that *any* error happens, so we could pick one of $\{X, Y, Z\}$ errors with some other probability distribution.

Additionally, we do not need to account for any additional memory errors that what was already mentioned for the movement path because it covers all the memory error during the waiting times before and after gates as well as in the middle of a qubit movement.

4.8.2 Idle Error

All qubit idling that is during movement along a path is already specified in the above movement model. We must next consider the idle error of qubits that are sitting around in memory or awaiting another qubit at a gate location in order to perform a gate. To get this information, we can look at the schedule of gate and movement events for each qubit, for any period it is not undergoing either of these operations, we assume it is subject to idle errors. Therefore we just have to go through the dependency graph of movement and gate operations and then extract an idle event for each time period between a source's end time and a sink's start time:

$$t_{idle} = t_{sink}^{start} - t_{source}^{stop} \tag{4.12}$$

The idle error probabilities are defined similarly to the path movement error:

$$(1 - p_{idle}) = (1 - p_{idle\ per\ \mu s})^t \tag{4.13}$$

p_{idle} is the total idle error probability for that time, $p_{idle\ per\ \mu s}$ is the probability of idle error per microsecond and t is the time in microseconds.

4.8.3 Example Movement/Idle Models

The above model is intended to provide maximum flexibility in describing variation in error probabilities during ion motion. There are a number of specific error model parameters that correspond to previous work:

Distance based

$$p_{straight}(w, d, v_i, v_f) = dp_{base\ move}$$
$$p_{turn}(w) = p_{base\ move}$$
$$p_{memory} = p_{start} = p_{stop} = 0$$

Physical Operation	Error Set 1 [32]	Error Set 2 [91]	Latency in (μs) [73]
One-Qubit Gate	10^{-6}	10^{-4}	10
Two-Qubit Gate	10^{-6}	10^{-4}	100
Measurement	10^{-6}	10^{-4}	500
Zero Prepare	10^{-6}	10^{-4}	510
Straight Move ($\sim$30 μm)	10^{-8}	10^{-6}	10
90 Degree Turn	10^{-8}	10^{-6}	100
Idle (per μs)	10^{-10}	10^{-8}	N/A

Table 4.8: Error probabilities and latency values used by our CAD flow for basic physical operations. This table is a reproduction of Table 1.1.

In this model, error probability is linearly proportionate to distance and distance only. This is arguably the simplest model and is the basis of analysis in [37, 96, 97]. This model was particularly suitable for these analysis because they all assumed a isotropic swap-based movement model.

Turn based

$$p_{straight} = p_{memory} = 0$$
$$p_{turn}(w) = p_{start} = p_{stop} = p_{base\ move}$$

This model is more specifically targeted for trapped ion technologies since it is generally believed that it is easier to move an ion in a controlled manner in a straight line rather than a accelerating it through a corner turning trajectory. This is the assumption made in [48].

Memory-centric

$$p_{turn}(w) = p_{base\ move}$$
$$p_{straight}(w, d, v_i, v_f) = dp_{base\ move}/2$$
$$p_{start} = p_{base\ move}/200$$
$$p_{memory} = p_{base\ move}/140$$

where $t(i)$ is in units of single qubit operation latencies. Even though the base p_{memory} is relatively low, $t(i)$ can potentially be very high in realistic circuit schedules. This model is featured in [56, 8].

4.9 Hybrid Error Modeling

In many cases, it is not necessary to have a fully-specified, detailed layout for the entire design to check properties or optimize. For a large design, it is not practical to construct a full, hierarchically flattened layout specification, all at the lowest macroblock level. Instead, we may extract error properties of reused modules once and apply them throughout the circuit simulation. We may estimate qubit movement in some situations instead of computing exact macroblock distance, like for long wires between modules at the coarse-mapping level. These all require a hybrid error model in which exact gate and movement errors from portions of the layout with macroblock-level detail are combined with inter-block communication estimates.

4.9.1 Error Streams from Layout Pieces

The basic behavior of a qubit in a layout is the following:

Pre-gate wait Wait in current location until time to do move to a gate location

Pre-gate move Move to a gate location to perform a gate

Operand wait Wait for additional operand qubits if gate operates on more than one qubit

Gate Perform gate operation

Post-gate move Move to another location, possibly memory, to wait for next gate

This process is repeated over and over again for each qubit throughout the run of the circuit. Our hybrid error stream generator extracts gate, movement and idle errors in different ways depending on the type of qubit and the type of communication. We will break down the error components on a qubit type basis and how they are derived for each situation. For all operations, we will separately specify the *physical processes* that qubit undergoes and the *mechanism by which we estimate the error for each process.* In the most general case of a LQLA or Qalypso datapath from Section 3.4.2 with macroblock-level layouts of compute, memory, and teleportation regions, we only have three real cases.

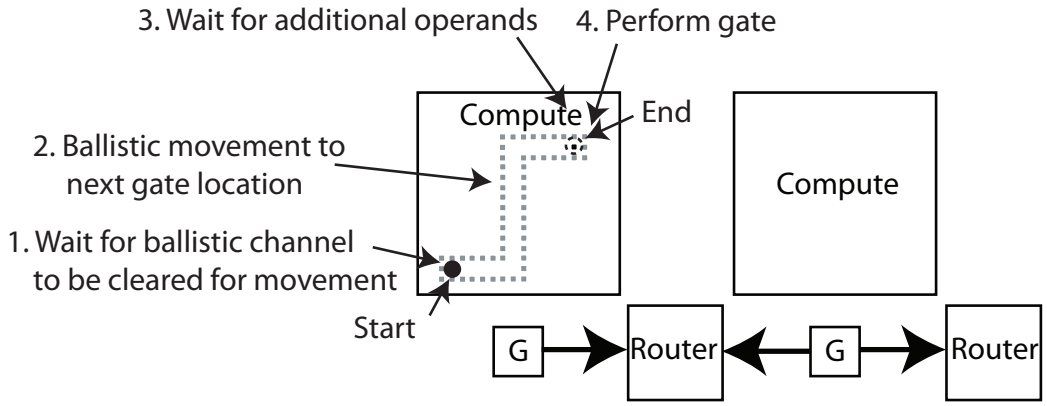


Figure 4.13: Simple sequence of 2 gates within the same compute region.

Intra-Region Gates

The simplest case of the above qubit behavior is shown in Figure 4.13. A qubit has finished a gate at one gate location and is scheduled to be used in another gate. Since everything happens within the same macroblock detailed layout, we can get precise numbers for all associated error events. All the possible stages are as follows:

1. Wait for ballistic channel: Our qubit may have to wait for the next gate to be scheduled or for the ballistic channel it will use to clear. Since we have exact scheduling information for gate and movements, we know how long the qubit must idle, thus we can get an exact idle error probability.

2. Ballistic movement to next gate: Once it is time, our qubit moves to the destination gate location. Since this is a macroblock layout we can apply the model in 4.8.1 to get the movement error probability.

3. Wait for operands: If our qubit is not the last one to arrive for a particular gate, it must idle there for some time. Since we know the exact arrival time of our qubit and the last qubit in the interaction, we can compute an exact idle error probability.

4. Perform gate: Once all the operands have arrived, the gate can be performed. We can apply the basic gate error operations detailed in Section 4.2.1.

All these macroblock level specified events can be directly input to our general error event simulators (joint probability or Monte Carlos).

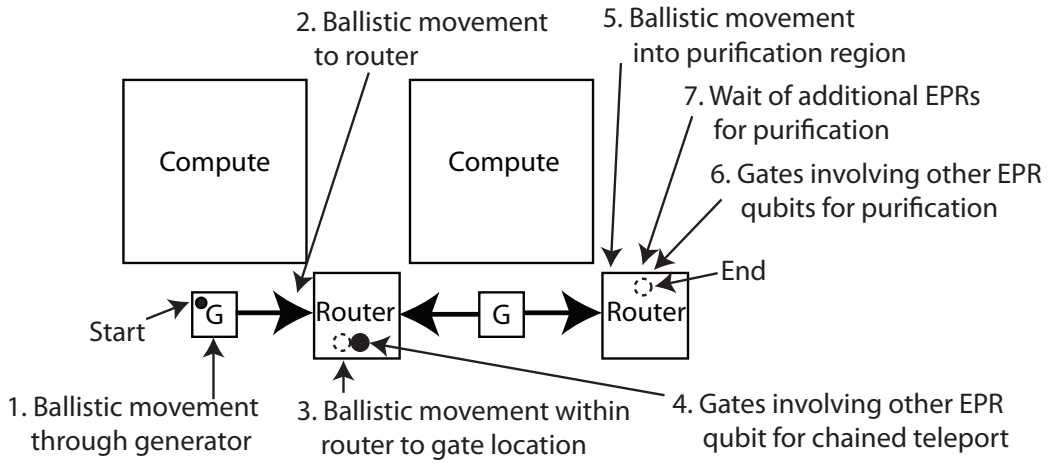


Figure 4.14: Life-cycle of an EPR qubit used in the teleportation based interconnect.

EPR Qubits for Teleportation

The path an EPR qubit takes when setting up a teleportation link for communication is rather complex. Section 1.2.4 gives an overview of the teleportation mechanism itself. Figure 4.14 shows all the error-prone stages that an EPR qubit undergoes when setting up this link. For EPR qubit distribution, we use the simpler teleportation fidelity model that we introduced in Section 4.1. Since our EPR qubits are not encoded and do not undergo complicated error correction procedures, we do not have to worry about correlated errors in these qubits. We extract error probabilities from each of these stages as follows:

1. Ballistic movement through generator: Since we have a macroblock detailed layout of the generator, we can get the precise movement error probability using the model we developed in Section 4.8.1.

2. Ballistic movement to router: We do not explicitly lay out the EPR distribution channels but given the sizing of a compute region and generator, we know how far a qubit moves to get to the router. Therefore we estimate the movement error probability as a function of this straight move distance.

3. Ballistic movement within router: Since we have a macroblock detailed layout of the router (schematic in Section 3.2.4, we know all the ballistic movement that must take place to get from the router input to the gate location used for chained teleportation.

4. Gates for chained teleportation: If our EPR qubit is involved in a multi-hop distribution to the location of the data qubit it will teleport, it must perform chained teleportation to get there. This involves several gates with other EPR qubits. Equation 4.3 gives the formula for our EPR qubit's fidelity after the chaining teleportation. We assume F_{old} is the fidelity of the qubit before the gates were applied and F_{EPR} is the fidelity of the chaining qubit.

5. Ballistic movement into purification region: After teleportation, the EPR qubit undergoes purification before it is interacted with the data qubit. Since we again have a macroblock detailed layout of the router, we know precisely what movement must happen and use that to compute the error probability.

6. Wait for purification: Since we must accumulate a number of identical EPR qubits first in order to do purification, our EPR qubit may have to wait for more to arrive. This is the only point in the whole process that our EPR qubit may idle. Based on which qubit this out of the total number needed to purify, we can compute the idle time and thus the idle error.

7. Gates for purification: We know what sequence of gates must occur to perform purification using the DEJMPS protocol from Section 5.4. Thus we can compute the resulting fidelity from this sequence of gates.

At the end of all this, we end up with an EPR qubit with some fidelity F_{EPR}.

Inter-Region Gates

Another complicated scenario is if our data qubit must move from a one gate in one compute region to the next gate in a different compute region. Figure 4.15 shows this case and the actions are as follows:

1. Wait for ballistic channel: The qubit first waits in its original location for the scheduling of the next gate and for the ballistic channel within the compute region to clear. Since we have detailed scheduling information on this, we can compute an exact idle error probability.

2. Ballistic movement out of compute region: To get to the interconnection router, we

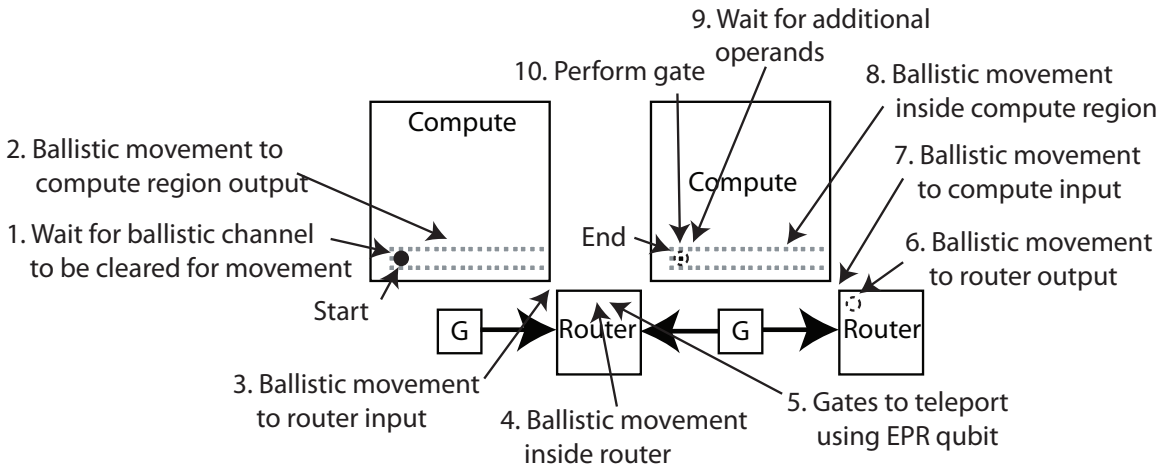


Figure 4.15: More complicated case of 2 gates in different compute regions.

must first move out of the compute region, this movement error can be derived exactly from the compute region macroblock layout and the model 4.8.1.

3. Ballistic movement to router: We do not have macroblock detailed layouts the router-compute region boundary but they are designed to adjoin each other, so the typical movement estimate is just a few straight macroblock moves and a turn macroblock move.

4. Ballistic movement inside router: Using the macroblock router layout and the detailed movement error model, we can derive the exact movement error for this event.

5. Teleportation gates: We must perform a gate with an EPR qubit and additional single qubit gates. The EPR qubit has errors calculated using the previous hybrid model and the gate errors can be calculated using the basic model in 4.2.1. The next section discusses the combination of our general depolarizing error model with the fidelity model.

6. Ballistic movement to router output: After teleportation, our data qubit must move out of the teleported-to router. The movement error is obtained using the macroblock router layout and the movement model in 4.8.1.

7. Ballistic movement to the compute region input: This is essentially the same as step 3 except in reverse.

8. Ballistic movement inside compute region: Same as step 2 in reverse.

9. Wait for operands: Similar to the intra-region case, we may need to wait for additional operands. We can get the wait time from the schedule and compute the idle error.

10. Perform gate: Once all operands have arrived, perform the gate. Errors come from 4.2.1.

In all these steps the data qubit error events are computed using the general error model presented in Sections 4.2 and 4.8. While the above description focused on moving into and out of compute regions, the same exact sequence of error events applies to ancilla qubits moving out of factories to be used in QEC procedures and non-transversal gates. Also, moving a data qubit into and out of a memory region is the same.

4.9.2 Putting the Pieces Together

We use several different types of error models in the above error events for the components of a qubit movement-idle-gate scenario: the gate error model is used from Section 4.2.1, the ion trap specific movement/idle model from Section 4.8.1 and the fidelity communication model from 4.1.2. The general gate and ion-trap specific movement/idle error models are already compatible with one another, since they both rely on the same depolarizing channel error assumption.

The fidelity based EPR error analysis has a different set of assumptions but we can conservatively convert our EPR qubit fidelity to a depolarizing error probability. Since fidelity is a measure of how much the given state varies from some reference state, namely the error-free state, we can take the inverse to get an overall probability of error.

$$p_{EPR\ error} = 1 - F_{EPR} \tag{4.14}$$

We use then propagate this aggregate error probability to the data qubit as discussed in Section 4.2.1. Thus, this final fidelity gets integrated into one of the more general joint or Monte Carlo simulators. With all these error events now compatible, we obtain the error event stream that we introduced in Section 4.4.

With these failure modeling techniques in hand, we are now ready to evaluate circuits, using gate, movement and communication errors that can be extracted from the layout and mapping techniques from Chapter 3. The composition of the various error

models presented in this chapter will allow us to do detailed analysis of circuits that are much larger than have been analyzed before.

Chapter 5

Error Analysis for Codes and Communication

The aim of this chapter is to analyze the error properties of some critical subsystems that will be used in a quantum computer. The error correcting subsystem and the teleportation network present interesting case studies for the tools we have developed so far.

One goal of this section is to identify codes that are particularly well suited for use in the large circuits we will investigate in Chapter 7. "Well suited" can mean a number of things, it could mean the code that delivers the absolute best success probability or the best in an aggregate metric such as expected total latency or ADCR. This is the first comparison of its kind in that it includes exact movement and idle errors in addition to gate errors for many codes. The error correcting code analysis is the first of its kind in that it includes exact idle and movement errors from layouts.

The teleportation network analysis shows off our fidelity based communication error analysis and is useful in not only calculating error probabilities, but also to determine the network resources needed to transport data over a given distance when it must stay below a given error threshold.

5.1 Comparison of Failures in Codes

The relative reliability of quantum error correcting codes has been discussed previously in [90, 25]. Both of these techniques ignored some or all of the impact of commu-

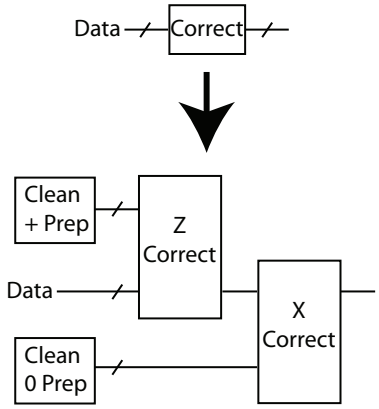


Figure 5.1: The basic error correction circuit for correcting a logical data qubit. Clean encoded ancilla qubits are used to extract the X and Z error syndromes from the data. "+ ancilla" refers to the equivalent zero state in the X-basis.

nication and idle errors. Cross et al [25] attempted to estimate the time during which a qubit was not in a gate and apply a memory error but this technique was inexact since it did not use an explicit schedule and did not include communication distance and time. Using the set of tools we have developed so far, combined with our error model and error simulation techniques, we can provide the first comprehensive study of the 3 error sources: gate, movement and idle errors.

5.1.1 QECC Choices and Benchmark Circuits

Table 3.1 shows the CSS codes we chose to analyze. Among them are the very popular [[7,1,3]] Steane code, the [[23,1,7]] Golay code (which was shown to be one of the best performing in a number of previous studies [25, 90]), the Bacon-Shor codes [4], and surface codes [16].

We generate encoders for the codes under consideration using Andrew Cross's qasm-tools [25]. Following the conventions in many other works [5, 95, 25], we start by encoding a single CNOT gate in the different codes under analysis. The test circuit we use is shown in Figure 5.2. We must now fully specify the correct steps in this figure: the ancilla verification and correction procedures are outlined in [90, 77]. Figure 5.1 shows a high level view of the error correction circuit we use for all fault tolerant designs in this work. The

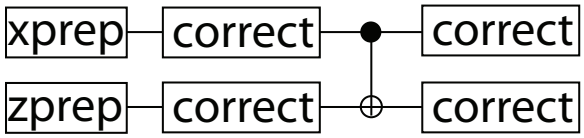


Figure 5.2: In this circuit, each qubit line represents a logical encoded qubit and the gates are logical, encoded gates. The correct blocks correspond to a Steane-type error correction step [87].

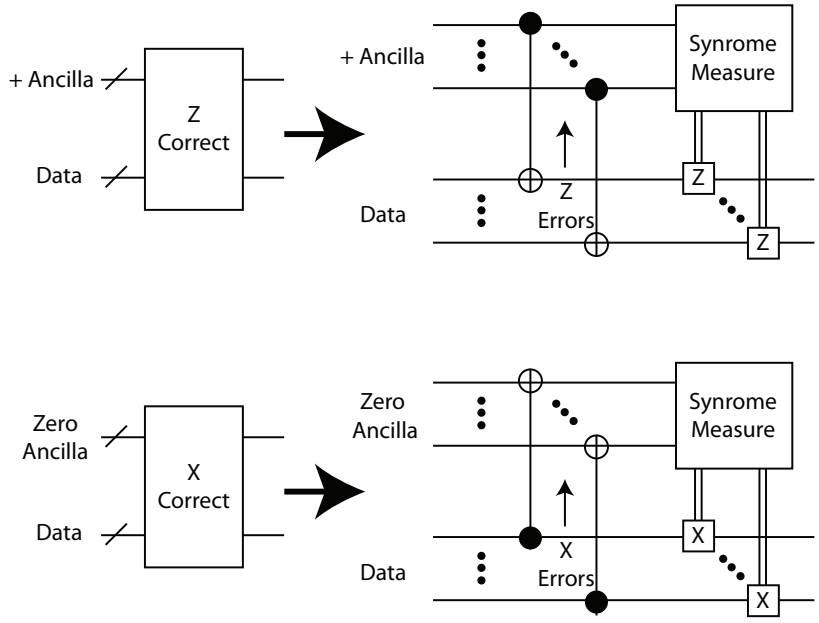


Figure 5.3: Internals of the two types of error correction. The procedures are essentially the same except the CNOT that transfers errors between the encoded data and encoded ancilla is reversed. This circuit only works for CSS codes, or codes that have a transversal implementation of encoded cnot gates. All the codes we investigate here are CSS codes.

key resources in this process are clean encoded ancillae.

Figure 5.3 shows how syndromes are extracted from data for both X and Z errors. The basic idea behind this procedure is that X and Z errors are allowed to flow from data to ancilla qubits through the CNOT gates. The syndromes are then measured from the ancilla to determine which data bits have errors. X errors are corrected with X gates and Z errors with Z gates. Since the ancilla are prepared in states that do not entangle with the data through the cnot, only the errors are entangled. Measurement collapses the potential

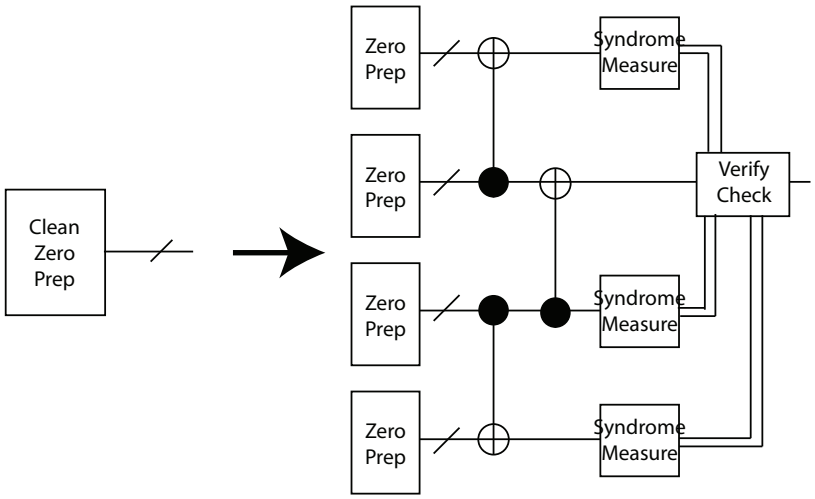


Figure 5.4: Internals of the clean zero preparation module. All gates/qubits shown here are encoded in the given code. Three of the four generated encoded ancilla are used to check for errors and the other ancilla is output as clean if all the checks succeed. The bottom ancilla is used to verify the verification ancilla. Multiple rounds may be necessary.

superposition of errors to a single error corresponding to the syndrome. The syndromes are the same as those from the code's *check matrix* in classical error correcting code terminology.

Clean zero ancilla production is shown in Figure 5.4 and follows the work of Steane [90] and Cross et al [25]. The entire circuit is repeated up to r times or until the verify check passes. The verify checks measure syndromes for error detection in order to determine if there are any errors that could corrupt data if this ancilla was used for correction. If all r rounds fail verification, the last prepared ancilla is passed on for correction anyway. r is set statically on circuit synthesis in our system. Since the probability that there will not be an error-free ancilla decreases exponentially with r, this is an acceptable trade-off, provided r is large enough.

Cross et al cataloged threshold estimates for the above codes in the single encoded CNOT circuit. Their estimates are a function of number of verification rounds [25], so we use these estimates as a starting point for our own search. In general, we choose the minimum number of verification rounds that gets us within 1% of the maximum achievable threshold for each code. By doing this we are looking for the "knee in the curve" where we can get a good threshold without requiring too many gates and qubits. In general, codes with larger block sizes require more verification rounds since there are more qubits

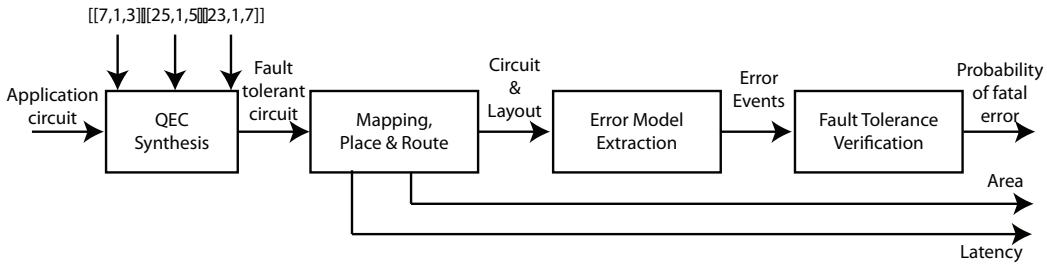


Figure 5.5: Tool flow used for evaluating the relative effectiveness of the error correcting codes under study.

interacting with one another to encode the block, thus more possibilities for error.

At the end of our analysis of the encoded, corrected CNOT circuit, we will have identified a set of "good" codes with low failure probabilities over a wide range of physical error probabilities. Once we have completed our analysis of the CNOT circuit, we will take the top performing codes and turn our attention to analyze code performance on the random circuits we developed in Section 2.2.3. We point out that the only other comparative code study on large (more than one logical gate) circuits was by Steane in [90] and large scale circuit results were computed by his analytical model, not through actual simulation of error propagation.

5.1.2 Evaluation Flow

To determine the probability of failure for each code, we layout the CNOT circuit, then the random circuits using our tools introduced so far. Figure 5.5 shows the tool flow used to do this analysis. Once the library file for the associated code is created using ftqctools, it is used by the QEC synthesis tool to encode the test circuit. It is laid out in the mapping/place and route phase, followed by extraction of error events and then the verification via error simulation using the vectorMC simulation from Section 4.7. We use two different movement error models, mentioned in Section 4.8.3: *distance based* and *turn based*.

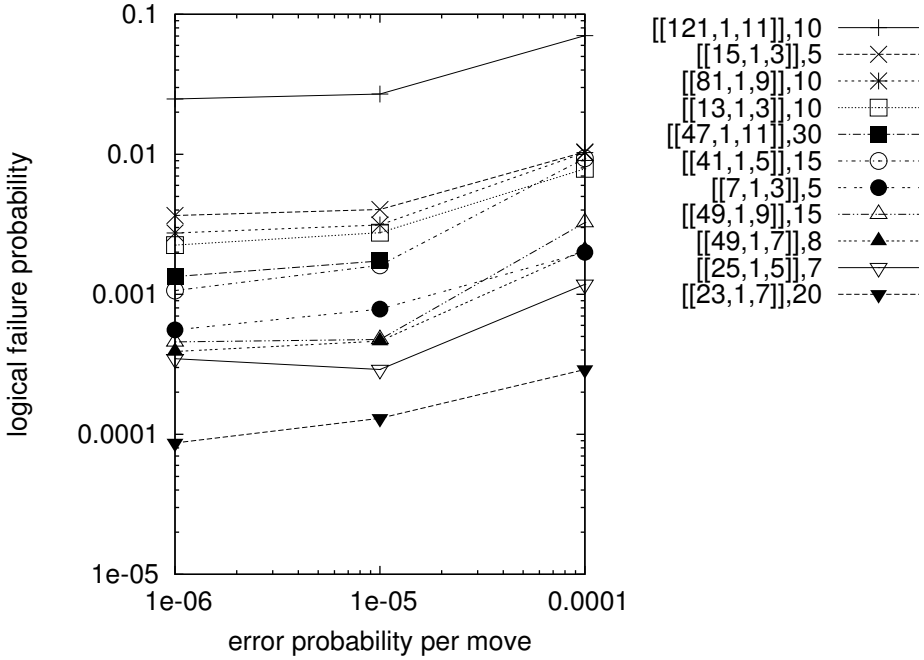


Figure 5.6: *Turn based movement error:* Logical failure probability as a function of the error probability of each qubit turn movement for the single encoded cnot circuit in Figure 3.19. The gate error probability is fixed at $p_{gate} = 10^{-3}$ and idle error was set to zero.

5.2 Comparing Code Pseudo-Thresholds

We begin our comparative study by looking that the probability of failure for each code when we lay out an encoded version of the circuit shown in Figure 3.19. This is the same circuit used in [25] for his code comparison. Since these circuits are small enough, we can lay them out in their entirety using just the macroblock level layout and directly extracting the error events.

Figure 5.6 shows code performance with the turn based error model as we vary the base movement error rate. In this case, the $[[23, 1, 7]]$ Golay code is consistently the best code, regardless of the movement error rate. The low movement error case is consistent with the results in [25, 90], since they do not account for movement. Since the turn based model imposes less total error compared to the distance model, it makes sense that these results match those for gate-only errors. The next best in performance is the $[[25, 1, 5]]$ Bacon-Shor code, which has been previously been shown to perform well due to its simple

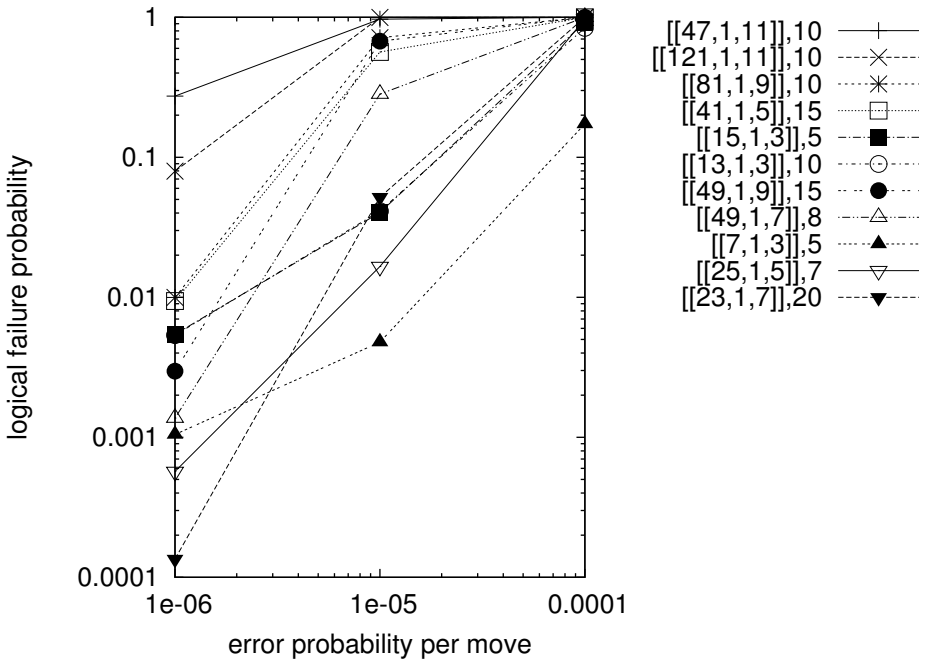


Figure 5.7: *Distance based movement error:* Logical failure probability as a function of the error probability of each macroblock movement. Just as in Figure 5.7, gate error probability is constant, $p_{gate} = 10^{-3}$, and idle error is set to zero.

encoder structure [7, 4]. Another perennial favorite, the Steane $[[7, 1, 3]]$ code begins only as 5th best but as the movement errors increase, it moves up to the number 3 slot, due to the fact that it as well has a simple encoding structure because the code itself is relatively small.

The curves for all the codes are relatively flat, at least compared to the distance based model we will show next. This is due to the fact that our dataflow heuristic explicitly tries to locate dependent gates in the same horizontal positions in different columns to make qubits movement just a straight shot through the gate locations, without turns.

Figure 5.7 shows code performance under distance based movement errors. While the $[[23, 1, 7]]$ Golay code again does well for low error rates, it is quickly overtaken by a number of other codes as movement error becomes more significant. Both the $[[25, 1, 5]]$ Bacon-Shor and $[[7, 1, 3]]$ Steane codes have lower failure probability by 3x and 10x respectively. In fact, in the high error case $p_{move} = 10^{-4}$ per macroblock, $[[7, 1, 3]]$ is the only code that keeps its head above water, with a failure probability under 0.2 while the rest of

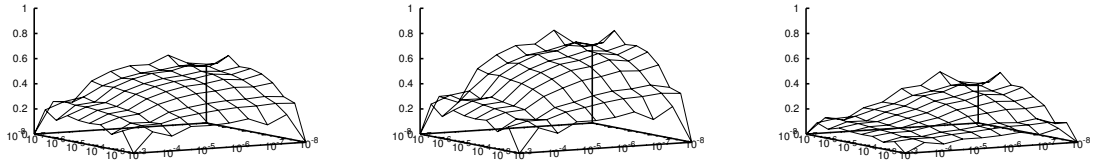
the codes converge to total failure (near 1.0 probability). The $[[7,1,3]]$ code fares better because it is more compact so the qubits simply do not have to move as far in the encoder to get to the logical cnot gate. The Golay code's gate-error efficient structure work to its advantage at first but finally the larger block size overcome any gate error efficiencies. The $[[25,1,5]]$ code overcomes $[[23,1,7]]$ in the mid-level error range, despite the slightly larger code size, because it has a simpler encoder. Thus there are fewer stages in the encoder dataflow graph that qubits must flow through.

Given the relative success of the $[[7,1,3]]$ code with more movement error, it is surprising that the 2 level concatenated version of this code, $[[49,1,9]]$, does not perform better than it does. It quickly rises to the top in failure probability. At a movement error probability of 10^{-5}, this code is already near the top of the worst performers. One explanation for this is that our layout or circuit does not take advantage of the locality available in the concatenated structure and introduces unnecessary movement. Future work will be to more closely investigate this issue.

In terms of how well these codes will perform on a real ion-trap based system, we consider 10^{-5} a reasonable target error rate. We are interested in the relative error rates between gate and movement errors. Projected error rates from [91] indicate a gate to movement error ratio of 100x so movement error of 10^{-5} vs. gate error of 10^{-3} is consistent. In the case of ion traps, the actual error rates will probably be somewhere between the distance and turn-based models, since changing momentum of the ions when moving around the corner will probably be more complex and noisy than straight line movement. Since our movement error model is fully parameterizable, we can perform new comparisons as physical data about the error processes becomes more plentiful.

5.2.1 Code Performance on Random Circuits

From these analyses, we single out the $[[23,1,7]]$, $[[25,1,5]]$, and $[[7,1,3]]$ codes as the promising ones. $[[23,1,7]]$ for its usefulness when movement error is low, and $[[25,1,5]]$ and $[[7,1,3]]$ for their relative robustness against movement error, especially in the 10^{-5} error rate region and beyond. We move on to study the performance of these codes under various error scenarios with a larger circuit. Our next set of benchmarks are from the random circuit generator introduced in Section 2.2.3. We have generated 5 random circuits with an average of 500 logical gates, using a splitting fraction of 0.5. This circuit size has



(a) [[7,1,3]] Steane code (b) [[23,1,7]] Golay code (c) [[25,1,5]] Bacon-Shor code

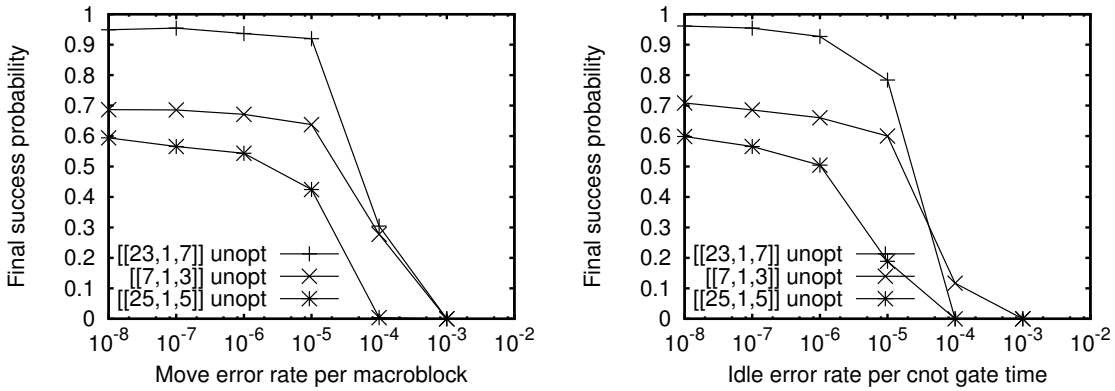
Figure 5.8: Error surface plots for 3 good codes. The z-axis is the final success probability and the x and y axes are the movement and idle error probability. The movement error probability is per macroblock moved (distance-based model from Section 4.8.3) and the idle error rates is per CNOT gate latency. The gate error rate in this example is fixed at 10^{-3}.

enough gates for non-trivial amounts of logical gate communication but still is small enough to allow extensive simulation with a variety of error parameters.

We encode each random circuit in each of the 3 QECCs, Each circuit is mapped to the Qalypso layout from Section 3.4.2. We extract error event streams for each using our hybrid error model from Section 4.9.

Since we are down to only 3 codes, we choose to map out a more complete picture of the associated error spaces of each code. We again fix the gate error rate at 10^{-3} and vary the movement and idle error rates across a range of values. For each error rate set, we apply the vectorMC success probability estimator from Section 4.7. The result is the set of *error surfaces* shown in Figures 5.8a-c. While the actual error values are difficult to compare on these graphs, they give us a global view of general sensitivities to error across the codes.

From the surfaces, we again verify some of the points mentioned before: the $[[23, 1, 7]]$ code exhibits the best performance with low movement and idle error, exhibited by it having the highest success probability in the back corner of the surfaces. All the codes are more sensitive to idle errors compared to movement errors, evidenced by a lesser slope in the success probability surface as movement error increases compared to the idle error slope. The $[[25, 1, 5]]$ code shows the lowest overall success probabilities for all the base error rates, including the minimal error case (back corner) and the maximal error edges in the front. Although we do not compute the exact amount, it appears that the $[[23, 1, 7]]$ code encloses the greatest volume out of the three, leading us to believe it is the



(a) Move error variation (b) Idle error variation

Figure 5.9: Success probability as a function of movement and idle errors.

most versatile code over a wide range of error rates.

The error surfaces give us some insight into a particular code's behavior over a range of error rates but they also make it difficult to compare codes to determine the best one in a given error scenario. Figures 5.9a and b give us a better opportunity to compare codes, plotting the same data as shown in the error surface series. These comparisons are similar to what we did for the single encoded CNOT circuit but it is interesting to see if anything changes when we start to deal with issues such as longer range communication and increased idle times in larger circuits. We see once again that all the success probabilities fall off faster with increasing idle error compared to movement error. All the observations made above are confirmed by these figures but we also notice that in the high idle error regime in Figure 5.9b, the $[[23, 1, 7]]$ code success probability falls off faster than the others and the $[[7, 1, 3]]$ ends up being a better performing code 10^{-4} to 10^{-3} error rate range.

5.2.2 Which code is best?

From our initial comparison of 11 quantum error correcting codes, we picked out 3 codes that seemed to have the best success probability over a wide range of error rates. The $[[7, 1, 3]]$, $[[23, 1, 7]]$, and $[[25, 1, 5]]$ codes were then subjected to further analysis on larger random circuits. We found that the $[[23, 1, 7]]$ code is a top performer for many different error rates due to its relatively good distance to block size ratio. $[[7, 1, 3]]$ outperforms the other two codes in the high idle error regime.

Now that we have identified some good codes for use in the quantum error cor-

rection subsystem of a fault tolerant architecture, we are going to switch gears and talk about another important component: the error analysis of the long-distance teleportation interconnect.

5.3 Analysis of Teleportation Interconnect

The teleportation-based interconnect plays an integral part in our tiled microarchitectures. Since this communication is so ubiquitous, we must carefully track the amount of error that EPR qubits used for teleportation can introduce in the data. As mentioned in Section 4.1 we leverage a simple fidelity based analysis of errors in the EPR transport and purification processes. We translate this conservative estimate of error to real fault probabilities when interacting with the data in a more detailed error simulation.

5.4 EPR Purification Model

As shown by Equation 4.3, the fidelity of the EPR pairs utilized in teleportation (F_{EPR}) has a direct impact on the fidelity of information transmitted through the teleportation channel. Since EPR pairs accrue errors during ballistic movement, teleportation by itself is not an improvement over direct ballistic movement of data qubits unless some method can be utilized to improve the fidelity of EPR pairs.

Purification combines two lower-fidelity EPR pairs with local operations at either endpoint to produce one pair of higher fidelity; the remaining pair is discarded after being measured. Figure 5.10 illustrates this process, which must be synchronized between the two endpoints since classical information is exchanged between them. On occasion both qubits will be discarded (with low probability).

The purification process can be repeated in a tree structure to obtain higher fidelity EPR pairs. Each *round* of purification corresponds to a level of the tree in which all EPR pairs have the same fidelity. Since one round consumes slightly more than half of the remaining pairs, resource usage is exponential in the number of rounds. There are two similar tree purification protocols, the DEJMPS protocol [26] and the BBPSSW protocol [11]. The analysis of the DEJMPS protocol provides tighter bounds which assures faster, higher fidelity-producing operation compared to the BBPSSW protocol. The effects

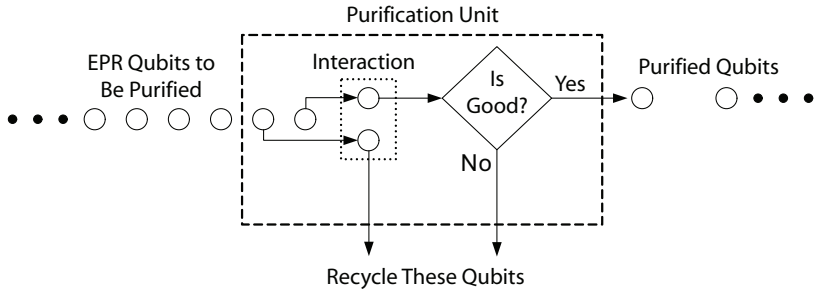


Figure 5.10: Simple Purification: pairs of EPR qubits undergo local operations, yielding a classical bit that is exchanged with the partner unit. Purification succeeds if classical bits are equal.

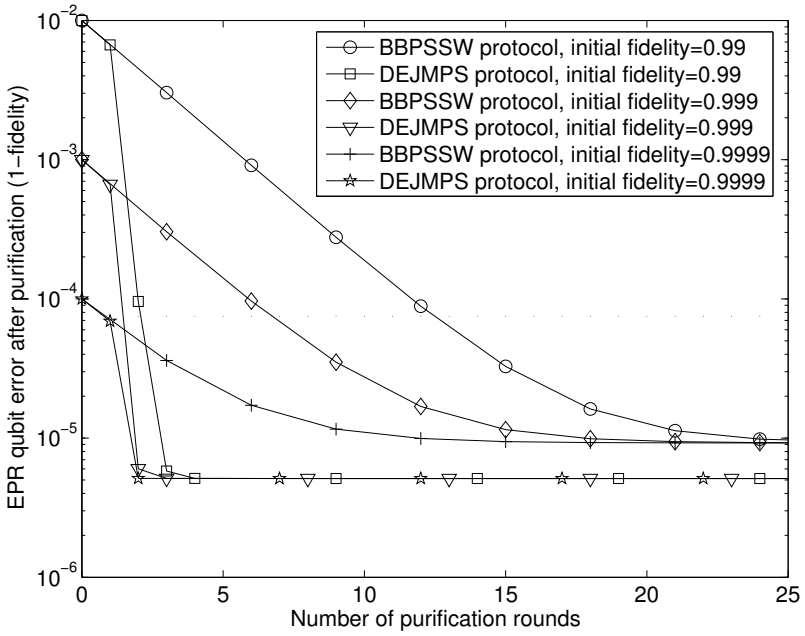


Figure 5.11: Error rate (1-fidelity) for surviving EPR pairs as a function of the number of purification rounds (tree levels) performed by the DEJMPS or BBPSSW protocol. Lower is better.

are significant, implying that purification mechanisms must be considered carefully[1].

Figure 5.11 shows error rate as a function of number of purification rounds. The BBPSSW protocol takes 5-10 times more rounds to converge to its maximum value as the

[1]Dur also proposes a linear approach to purification [31]; unfortunately, it appears to be sensitive to the error profile. We will not analyze it here.

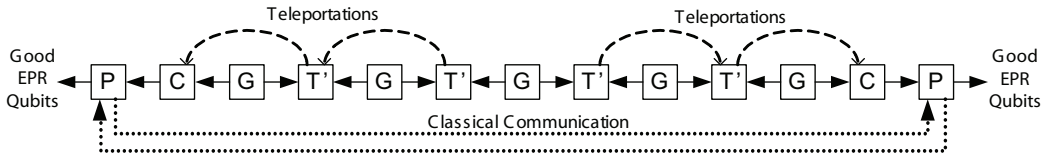


Figure 5.12: Chained Teleportation Distribution Methodology: EPR qubits generated at the midpoint generator are successively teleported until they reach the endpoint teleporter nodes before being ballistically moved to corrector nodes and then purifier nodes.

DEJMPS protocol. Since EPR pair consumption is exponential in number of rounds, the purification protocol has a large impact on total EPR resources needed for communication. Other features of Figure 5.11 to note are that DEJMPS has higher maximum fidelity and converges to maximum fidelity faster than BBPSSW (possibly because BBPSSW partially randomizes its state after every round).

Finally, the time to purify a set of EPR qubits is dependent on the initial and desired fidelity. The time to complete one round of purification is $1210\mu s$ from Table 4.8:

$$t_{purify\ round} = t_{2q} + t_{ms} + t_{classical\ bit} \tag{5.1}$$

5.5 Teleportation Network Fidelity Analysis

All of the tiled microarchitectures from Section 3.4.1 and 3.4.2 use teleportation based interconnect for long range qubit communication. In long range communication, the preservation of data qubit fidelity, is our highest priority. Therefore, we choose to transport all data by way of single teleports, since this introduces the minimum error from ballistic movement. This necessitates the distribution of EPR pair qubits to communication endpoints. Since data qubits interact with these EPR pairs, the above threshold must be imposed on them to avoid tainting the data.

Two options present themselves for distributing high-quality EPR pairs to channel endpoints. First, one could ballistically move the EPR pairs to the endpoints, which is preferable to moving data ballistically because EPR pairs can be sacrificed if they accumulate too much error. Second, one could route EPR pairs through a series of teleporters, as shown in Figure 5.12. While preserving fidelity of our data states is top priority, when dealing with less precious EPR pairs, we do not have to adhere to strict maximal fidelity

preserving distribution methods. In the rest of this section, we will investigate the tradeoffs between ballistic distribution and chained teleportation distribution of EPR pairs.

Fidelity Difference The final fidelity of these two techniques is approximately the same. Conceptually, the final EPR pair either directly accumulates movement error (through ballistic movement) or is interacted with several other EPR pairs to teleport it to the endpoints and these intermediate EPR pairs have accumulated the same distance ballistically. By interacting with intermediate pairs, the final pair accumulates all this error. This statement assumes that the fidelity loss from gate error is much less than the loss due to ballistic movement, which is the case for ion traps, as shown in Table 4.8 (for two teleporters spaced 100 cells apart, ballistic movement error equals $1-(1-10^{-6})^{100} \approx 10^{-4}$ compared to 10^{-7} for a two-qubit gate error).

Long-distance distribution of EPR pairs can severely reduce the fidelity of the EPR pairs arriving at a functional unit for data teleportation, as shown in Figure 5.13. In order to process 1024 qubits, we could imagine arranging them on a square 32x32 grid, in which the longest possible Manhattan distance is 64 logical qubit lengths. If we assume that we have teleporter units at every logical qubit, EPR pair distribution could require up to 64 teleports. From the figure, teleporting 64 times could increase EPR pair qubit error by a factor of 100. The dotted line represents the threshold at which the EPR pairs must be in order to not corrupt the data qubit when teleporting it. In order to preserve data fidelity, we must use EPR pair purification. One way to think about this process is to stitch Figures 5.11 and 5.13 side-by-side, so that EPR pairs accumulate error (degrade in fidelity) as they are teleported and then purified back to a higher fidelity at the endpoints before being used with data.

Latency Difference If teleportation is considered performed in near constant time, then we would like to know the distance crossover point where teleportation becomes faster than the equivalent ballistic transport. From Table 4.8, teleportation takes about $122\mu s$ while ballistic movement takes $0.2\mu s$ per ion trap cell. Thus for a distance of about 600 cells, teleportation is faster than ballistic movement. We assume our communications fabric to be a 2-D mesh of teleporter nodes and use 600 cells as the distance that each teleportation "hop" travels. Allowing teleportations of longer distances would further reduce communication latency in some cases but would then require more local ballistic movement to get

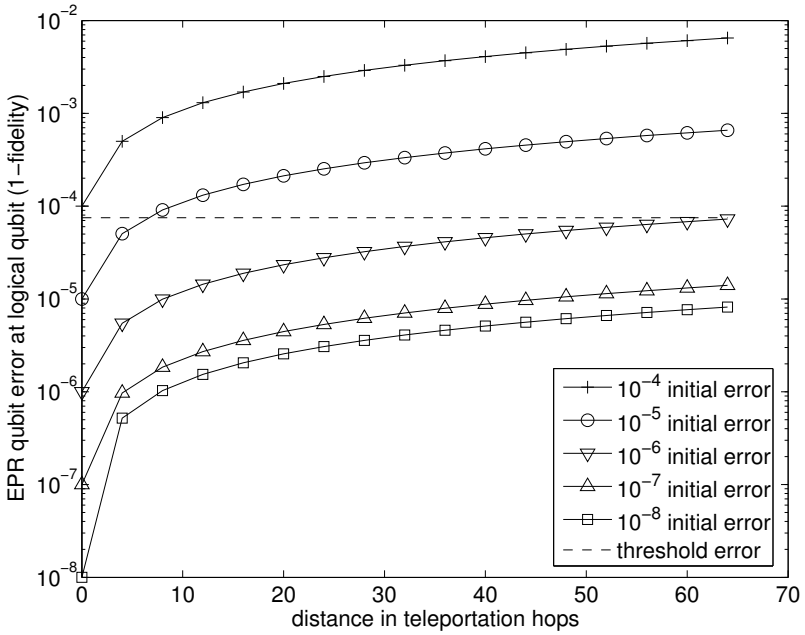


Figure 5.13: Final EPR error (1-fidelity) as a function of number of teleportations performed, for various initial EPR fidelities. The horizontal line represents the minimum fidelity the EPR pair must be at to be suitable for teleportation of data qubits, $1 - 7.5 * 10^{-5}$

an EPR pair from the nearest teleporter to its final destination.

5.5.1 Purification Resources

Earlier in this section, we noted that when we purify a set of EPR pairs, we measure and discard at least half of them for every iteration. This means that to perform x rounds, we need more than 2^x EPR pairs to produce a single good pair.

To measure EPR resource usage, we count the total number of pairs used *over time* to move a level 2 [86] error corrected logical data qubit between endpoints. This means we are transporting 49 physical data qubits some distance by way of teleportation. We find that the total number of EPR qubits necessary to move a datum critically affects the data bandwidth that our network can support. This metric differs from that used in a number of proposals for quantum repeaters which focus on the layout of a quantum teleporter and are most concerned with spatial EPR resources, i.e. how much buffering is necessary for a particular teleporter in the network [22]. We will show that our design is fully pipelined,

and therefore only a small number of qubits must be stored at any place in the network at any time.

We saw in Figure 5.11 that if we start at a relatively low fidelity and try to obtain a relatively high fidelity, we could need more than a million EPR pairs to produce a single high fidelity pair using the BBPSSW protocol. Therefore we use the DEJMPS protocol in all further analysis. Even though the DEJMPS protocol converges to good fidelity values much quicker, the exponential increase in resources for each additional round performed means we must be careful about how much error we accumulate when distributing EPR pairs. We will also show that the point in the datapath at which purification is performed can have a dramatic impact on total EPR pairs consumed. We have 3 options:

Endpoints only: Purify only at the endpoints, immediately before using EPR pairs to teleport data.

Virtual wire: Purify EPR pairs which create the links between teleporters, namely the constant stream of pairs from a G node to adjacent T' nodes. The result is higher fidelity qubits used for chained teleportation.

Between teleports: Purify EPR pairs *after* every teleportation; this purifies qubits that are being chain teleported rather than qubits assisting the chained teleportation.

We now model the error present in our entire communication path. Assuming the EPR pairs at the logical qubit endpoints must be of fidelity above some given threshold, we determine the number of EPR pairs needed to move through different parts of the network per logical qubit communication.

Total EPR Resources Figure 5.14 shows that the Endpoints Only scheme uses the fewest total EPR resources. This conclusion is evident if we refer back to Figure 5.11, where purification efficiency asymptotes at high fidelity; thus, purifying EPR pairs of lower fidelity shows a larger percentage gain in fidelity than purifying EPR pairs of high fidelity. From this, we can see that to minimize total EPR pairs used in the whole system, it makes sense to correct all the fidelity degradation in one shot, just before use.

Non-local EPR Pairs Another metric of interest is to focus only on those EPR pairs that are transmitted to endpoints during channel setup (i.e. those that are teleported through the path). This resource usage is critical for several reasons: First, every EPR pair moved

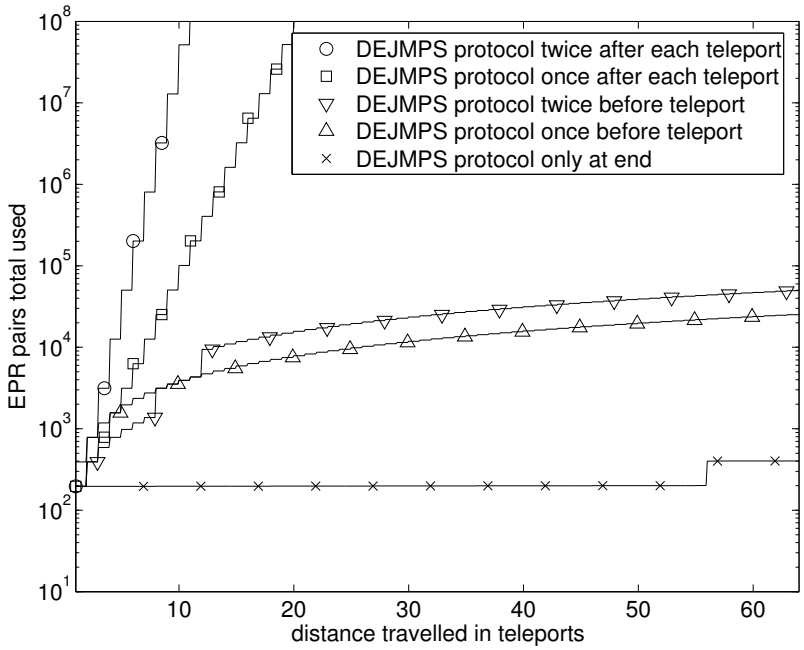


Figure 5.14: Total EPR pairs consumed as a function of distance and point at which purification scheme DEJMPS is performed.

through the network consumes the slow and potentially scarce resource of teleporters; in contrast, the EPR pairs consumed in the process of producing virtual wires are purely local and thus less costly. Second, because of contention in the network, EPR pairs communicated over longer distances (multiple hops) place a greater strain on the network than those that are transmitted only one hop. The channel setup process can be considered to consume bandwidth on every virtual wire that it traverses. Third, the total EPR pairs transmitted to endpoints during channel setup consumes purification resources at the endpoints—a potentially slow, serial process.

Figure 5.15 shows that purifying EPR pairs after each teleport transmits many more EPR pairs than purifying at the endpoints (either with or without purifying the virtual wires). From this figure, we see that over-purifying bits leads to additional exponential resource requirements without providing improved final EPR fidelity[2]. Virtual wire purification improves the underlying channel fidelity for everything moving through the teleporters, thereby allowing less error to be introduced into qubits traveling through the channel. For

[2]The authors of [22] claim that this nested purification technique (after every teleport) has small resource requirements; however, they count spatial resources rather than total resources over time.

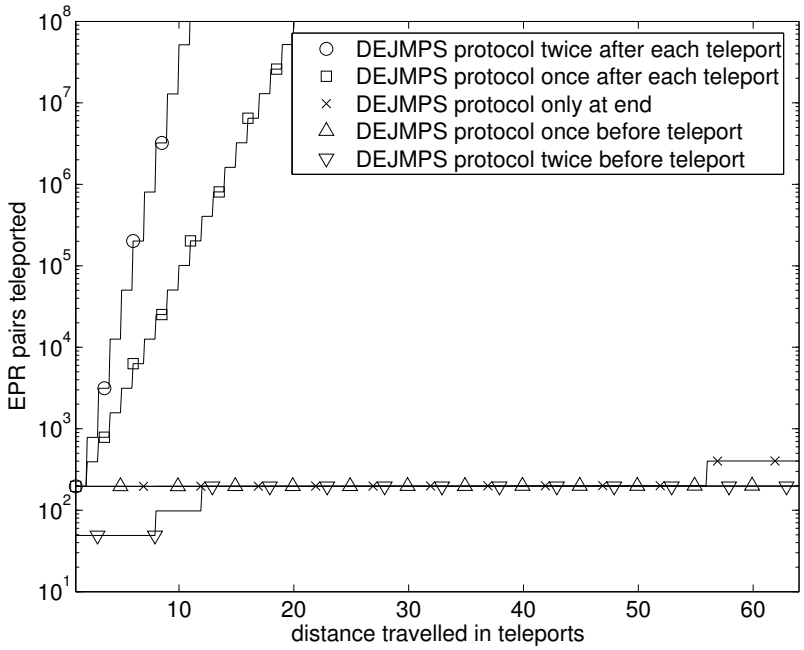


Figure 5.15: Total EPR pairs in teleportation channel as a function of distance and point in transport in which purification scheme DEJMPS is performed. The only 2 lines that change from Figure 5.14 are the purify before teleport cases.

a given target fidelity at the endpoints, virtual wire purification reduces the number of EPR pairs that need to move through the teleporters and also reduces the strain on the endpoint purifiers.

To summarize, we have made the following design decisions based on fidelity and latency concerns:

Teleport data always: Data qubits sent to destination with single teleportation to minimize ballistic error.

Teleport EPR pairs: EPR pairs distributed to endpoints with teleportation, allowing pre-purification to increase the overall fidelity of the network.

Purification before teleport and at endpoints: Purify intermediate EPR pairs before they are used for teleportation as well as EPR pairs at the channel endpoints.

Finally, Figure 5.16 shows the sensitivity of the EPR resources necessary to sustain our previous error threshold goals as a function of the error of the individual operations like quantum gates, ballistic movement, and quantum measurement. The first thing to

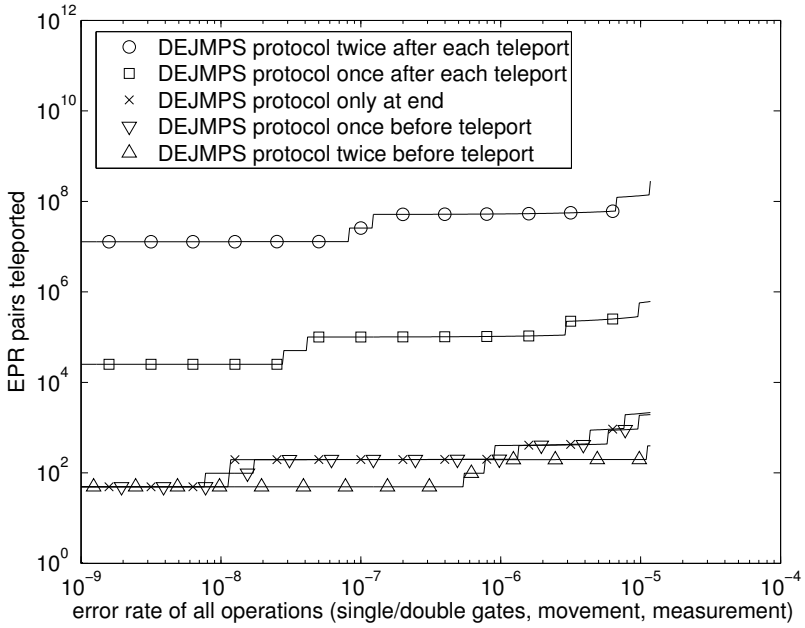


Figure 5.16: Number of EPR pairs that need to be teleported to support a data communication within the error threshold. All error rates are set to the rate specified on the x-axis.

note are the abrupt ends of all the plots near 10^{-5}. This is the point at which our whole distribution network breaks down, and purification can no longer give us EPR pairs that are of suitably high fidelity (above $1-7.5*10^{-5}$). The fact that all the purification configurations stop working for the same error rate is due to the fact that the purification schemes we investigated are limited in maximum achievable fidelity by operation error rate and not the fidelity of incoming EPR pairs (unless the fidelity is *really* bad). Throughout the regime at which our system *does* work however, the total network resources only differ by a factor of up to 100 for a 10,000 times difference in operation error rate.

In this chapter, we have applied the tools developed in Chapter 3 and 4 to provide new analyses of some key components of any fault tolerant quantum computer design. The work done in this chapter will be used to make important design decisions later on for the support infrastructure of large circuits.

Our analysis is enabled by the tools described in earlier chapters. From the layout that we designed using methods in Chapter 3, a stream of error events can be extracted using the process described in Section 4.2. This event stream is then fed into the accelerated

vector Monte Carlo error simulator described in Section 4.7.

Chapter 6

Optimization of Fault Tolerant Circuits

We mentioned earlier that quantum error correction has been estimated to contribute a very large amount of overhead to a fault tolerant circuit. Previous rough estimates have been that 95% of all the resource will go towards quantum error correction. In order to verify this, we took 3 application circuits and encoded them in the $[[7, 1, 3]]$ code using our QEC synthesis tool and looked at the breakdown in gate count for the encoded and unencoded circuits. We look at a randomly generated circuit from the generator described in Section 2.2.3, a quantum carry-lookahead adder (to be described in Section 7.1.1), and a Shor's factorization implementation from Section 7.2.1. Table 6.1 shows the breakdown in gate counts of these circuits. We note that all 3 circuits experience a thousand-fold increase in gates from the logical circuit to the encoded, corrected version. This is because all the logical gates are being expanded to many physical gates to implement the encoded version, and we have added correction stages after every gate. If we look at the fraction of all these

Application	Logical gates	Total gate operations	Percent QEC
Random	4.7×10^3	5.73×10^6	99.3%
Adder	6.75×10^4	7.98×10^7	95%
Shor's	2.14×10^{12}	1.12×10^{15}	95%

Table 6.1: Gate counts for 3 circuits encoded in the Steane $[[7, 1, 3]]$ code.

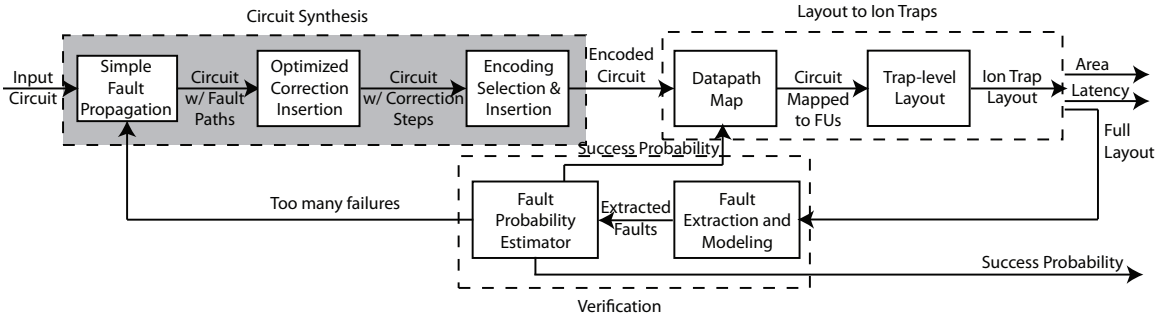


Figure 6.1: The optimization stage in our design flow takes in a circuit with encoding information and outputs an encoded circuit with correction stages inserted.

physical gates in the encoded circuit that go towards the correction stages, it is between 95% and 99%. This verifies the earlier estimates from others that quantum error correction will take up 95% of all quantum circuits.

The purpose of this chapter is to devise a method to reduce this prohibitive overhead, without sacrificing the necessary gains in reliability that error correction provides.

6.1 Error Correction Placement

Quantum error correction is a critical component to any quantum circuit due to high qubit operation error rates. Figure 6.1 shows the process we use to create a fault tolerant design. To produce there fault tolerant quantum circuits, we automatically synthesize circuits with error correction procedures inserted. We introduce a novel technique for optimizing the insertion of these error correction procedures. We then validate the fault tolerance of the final design to make sure the error correction used adequately protects the data.

Figure 6.2 shows our basic approach to reducing error correction overhead. Using the standard correct-after-every-gate scheme as a starting point, we would like to remove the correction stages that have the "least impact" on overall circuit reliability. The question becomes: which correction stages are the least important? We assume that any circuit of interest will not have a completely homogeneous topology with respect to each qubit. Thus, some qubits are likely to be subjected to more error opportunities than other qubits at certain points. We reason that it makes sense to correct these qubits more often. We call these points in the circuit *error critical* and the circuit structure determines a *error*

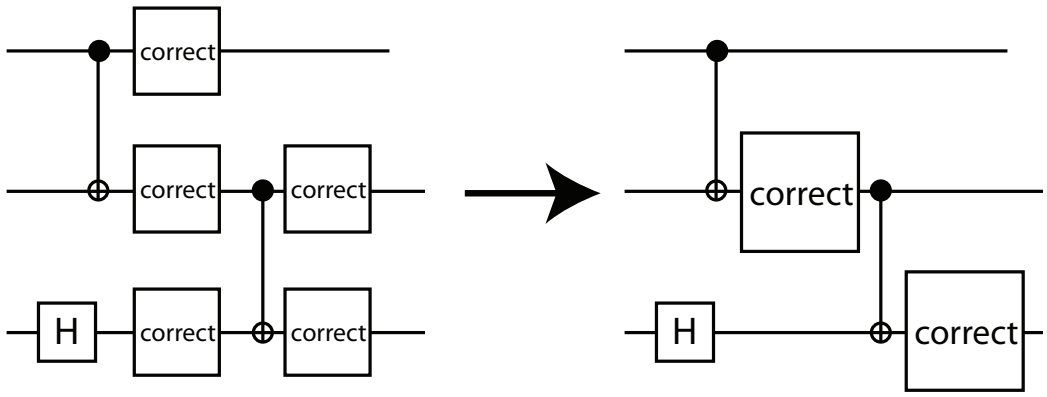


Figure 6.2: Instead of correcting after every single gate in the circuit as has been suggested before, our technique only places error correction circuits where they are most needed.

criticality spectrum across all the qubits at all points in the circuit. A qubit undergoing a lot of operations at one point in the circuit might be error critical but later when the qubit is no longer used in the computation and is just idling, it would not longer be error critical.

With respect to the fault propagation modeling techniques developed in Chapter 4, we consider the error critical parts of the circuit to be the ones that are most likely to have an unrecoverable error. For each modeling technique, the points that are most likely to have unrecoverable errors are:

- Communication with the lowest qubit fidelity in the case of the fidelity based network reliability analysis.
- Highest error probabilities in the joint probability model.
- Sampled unrecoverable errors in the case of the Monte Carlo methods.

We must estimate the effect of various fault paths through a circuit, We propose to do this in the simplest manner to start with: model the gate networks as a error generating/passing network. Each gate contributes one error unit to each qubit and it also propagates the maximum error on the input qubits to all output qubits.

6.2 Fault Counting Model

Figure 6.3 shows an example of a circuit and how our fault counting model works on it. This model assumes that each gate contributes one fault unit to each qubit in the

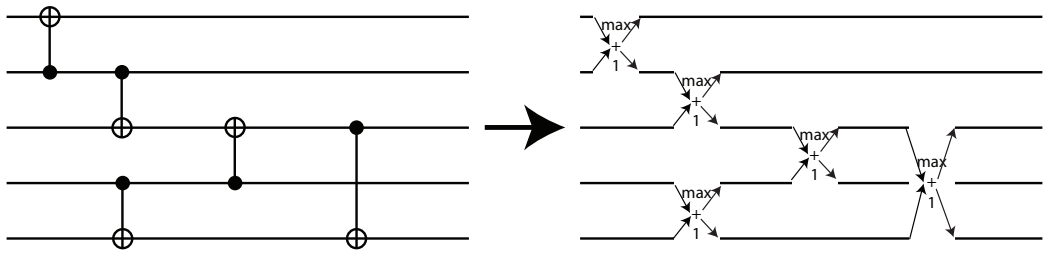


Figure 6.3: This simple model is used to estimate error propagation on the circuit to be optimized. Each gate effectively takes the maximum current error count out of all the input qubits, adds one count for the gate error itself, then sets the output qubits to this value. This models the fact that in general, the fault tolerance of a gate is limited by the most error prone input qubit.

interaction and that errors are propagated from qubits with higher counts to qubits with lower counts. Furthermore, we assume that a correction operation zeros this count. This model tries to balance reality and simplicity, it exhibits some realism because:

- Gates are believed to be the primary source of errors on qubits so their role as generators of error units is appropriate.
- While different 2-qubit gates have different rules for propagating different errors between the qubits interacted, they all pass some types of errors between them. This model assumes that all errors propagate to all qubits, which is pessimistic.
- Correction stages correct errors, so we would expect them to remove error units from the qubits.

We will use this model to drive the optimization technique we will introduce in the next section. However, it is good to be aware of this model's limitations:

- No differentiation between phase (Z) and bit (X) errors in error propagation. For example, a CNOT should propagate X errors from control to target and Z errors the opposite way.
- The real combination of errors on qubit inputs to a gate is a more complicated function than a simple maximum.

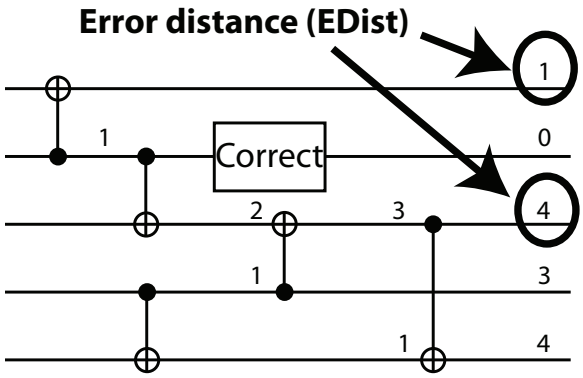


Figure 6.4: An example of our simple error model on a circuit.

- Each bit line in represents an encoded qubit so there are potentially complex interactions between qubits in the encoded block not encoded by this counter.
- Correction operation success is not really dependent on such a simple logical-level counter, but a complicated error distribution over the qubits that make up the encoded logical qubit.
- We take the maximum of all the input error counts instead of some other aggregate, such as minimum or sum. The max is not necessarily the most accurate way to model real error propagation but we choose it out of convenience, since it is the easiest to optimize.

Figure 6.4 shows an example of applying the rules from Figure 6.3 to a circuit. We call the error counts on the qubits the "error distance" or $EDist$. For a given circuit, we define the $EDist_{max}$ to be the largest EDist in the circuit.

One reason why this particular model is interesting is that it closely matches the model for estimating the latency of a classical circuit. In a classical circuit a gate's output is not stable or usable until the latest classical bit value reaches it. Thus, the total circuit latency could be calculated as finding the global maximum of the output bit times as a function of this local maximum calculation. We will discuss this more in the next section. This global max is the same as the $EDist_{max}$ we just defined.

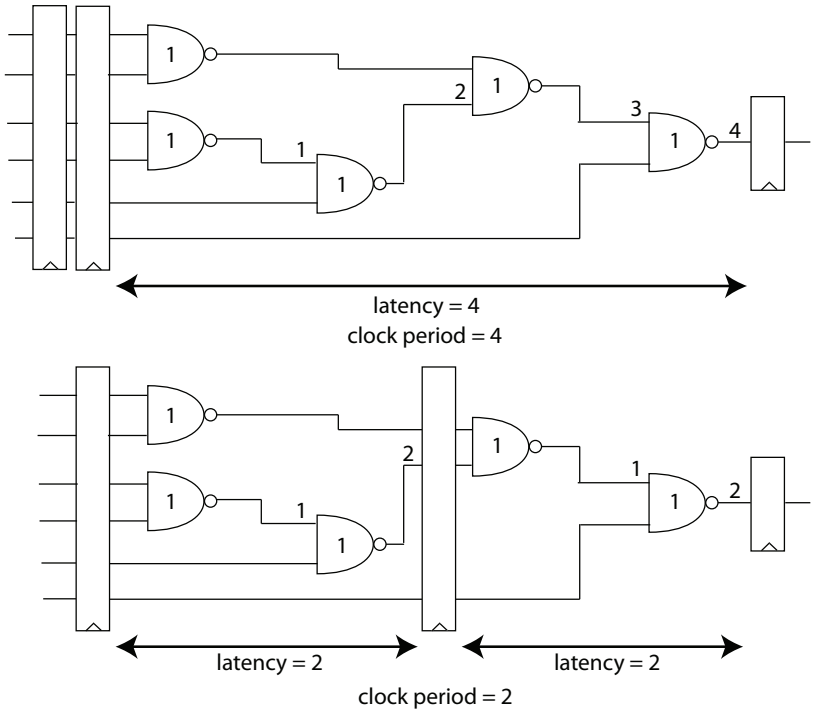


Figure 6.5: Classical circuit retiming example: in order to minimize the latency through this circuit, we move the extra register from the left to the point where the subcircuits on either side of the partition have equal latency.

6.3 Retiming for EC Placement in Circuits

In our optimized process, we try to add error correction steps only after the qubits with the highest error counts, as long as we keep all qubits below a correctable threshold. As mentioned before, this has an analog to minimizing the critical path latency of a circuit and we have formulated this problem as a case of circuit retiming [59] from the classical CAD literature. In this section, we will review classical retiming and give a formal definition of our own quantum recorrection process.

6.3.1 Classical Circuit Retiming

Figure 6.5 shows an example of a classical CAD technique called circuit retiming, as presented by Leiserson and Saxe [59]. Through this technique, we redistribute the registers in a synchronous circuit in order to balance the latency of the between-register circuit stages.

The idea is that the overall clock period of the circuit is determined by the maximum latency stage, so we want to minimize the maximum latency stage.

This technique is performed by framing the register moving problem as a linear program. A simplified version of the solution from [59] is given in Algorithm 4.

Algorithm 4 Retiming: Input: circuit graph G with vertices V and edges E

Compute minimum register counts between all vertex pairs in V

Compute maximum delay for all minimum register paths for all vertex pairs in V

Solve for a retiming r (a system of equations operating on register placements) so that registers are not created or deleted from the circuit and so high delay paths include at least one register

Modify register placements based on original positions and retiming r

Many interesting details are left out of this algorithm, they can be found in [59]. The intention of the above is to show the key portions of the algorithm. The first two steps consist of running two all-pairs shortest-path algorithms, one for the paths with edge weights and one for the paths with vertex weights. The next step involves solving a system of linear equations to find a retiming, if one exists. The last step takes this retiming and moves the registers around in the circuit. This simple version of their retiming algorithm takes $O(|V|^3 \log |V|)$ time, due to the time it takes to solve for r. A more complex but faster algorithm can complete the task in $O(|V||E| \log |V|)$ time [59]. We will go over many of these details in Section 6.3.3.

6.3.2 Transforming Latency Retiming to Error Recorrecting

We want our circuit have enough correction steps to have a high output success probability, but not too many steps so as to incur prohibitive overhead. We therefore have two constraints if we would like to place error correction steps in our quantum circuit:

- Placing too many error correction steps does not necessarily protect our data better. If the encoded qubits have a very low probability of any error, it is more resource efficient to put off correction until it is more likely there are errors to correct.
- Placing too few error correction steps can be much worse because qubits will accumulate more errors and encoded blocks will have a higher probability of having uncorrectable errors which will ultimately lead to corrupted output.

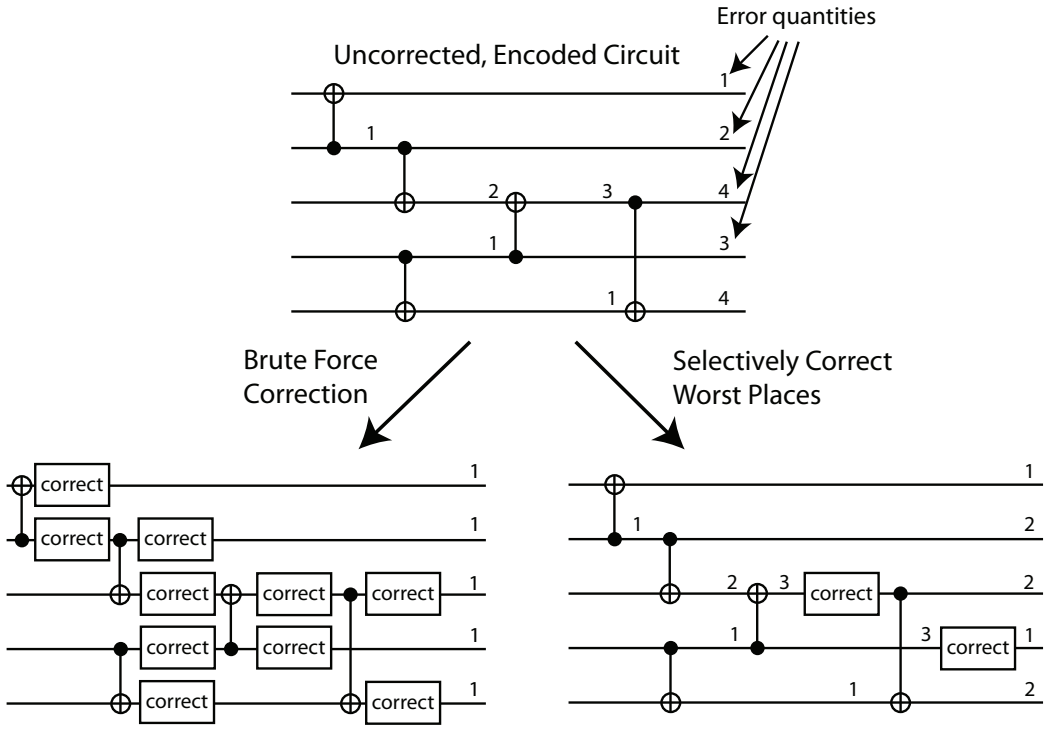


Figure 6.6: The top circuit shows a simple model of counting errors introduced to qubits through gates. Each gate adds one error unit to each qubit involved. Additionally, when qubits interact, they propagate their error counts to each other. The left circuit shows the same circuit with the standard conservative placement of error correction procedures (one after every gate).

Furthermore, the gates in the error correction procedure itself also contribute error, so if our encoded qubits have zero error on input to the error correction step, they could come out with more error than they went in with. *Over-corrected qubits might have a higher fault probability than "optimally" corrected qubits.* We will return to this hypothesis later in the section.

In the end, our goal is the following: *minimize the number of resources used for error correction in a quantum circuit while minimizing the probability of errors on the output.*

In order to minimize errors in our circuit overall, we would like to make sure all the qubits going into error correction steps have not already acquired an uncorrectable number of errors. Under our fault counting model from Section 6.2, our goal of minimizing

the maximum error count ($EDist_{max}$) is very similar to minimizing classical circuit stage latency. If we assume that an error correction step resets a qubit's error count to zero, much like a register resets a classical bit latency to zero, we just have to exchange register placement with error correction step placement. We call our correction step placement optimization "recorrection" and will now show its close relationship to retiming.

6.3.3 Formal Definition of Recorrection

To be precise we now present the formal framework for our recorrection optimization. We start with an algorithm that takes a circuit with some correction steps already placed in it (not after every gate) and positions them to minimize the maximum error count, or $EDist_{max}$. The next step is to determine how many correction steps should be placed. To do this, we define an input parameter: $EDist_{threshold}$ to be the maximum tolerable $EDist_{max}$ for the circuit. We continue by showing how to determine the minimal number of corrections necessary to satisfy a given $EDist_{threshold}$. Since the recorrection technique very closely mirrors classical circuit retiming, we will follow the derivation of retiming given by Leiserson and Saxe [59] and point out how our optimization differs. We stress that most of the reasoning in this derivation is from the previous work and that our focus is on the connection between latency in classical circuits and error propagation in quantum circuits. We preserve the labels of the algorithms and theorems from [59] and omit the proofs except where the recorrection case differs.

To restate the problem, we are given as input to our recorrection algorithm an $EDist_{threshold}$ and a corrected or uncorrected, encoded circuit. Compared to the retiming case, this is solving the reverse problem: we may initially not have any correction steps in our circuit and will have to insert some to meet the required $EDist_{threshold}$. In the retiming case, the number of registers is fixed and they are just moved to achieve some initially unknown optimal clock period. We represent our input circuit as a weighted multigraph:

$$G = (V, E, d, w) \tag{6.1}$$

V is the set of vertices and E is the set of edges. Each vertex represents a gate in the circuit and each edge $e : u \xrightarrow{e} v$ represents a qubit communication from gate u to gate v. If $v \in V$, $d(v)$ is the error unit count on the gate v. If $e \in E$, $w(e)$ is the number of correction steps along that qubit communication link. An initial uncorrected circuit input

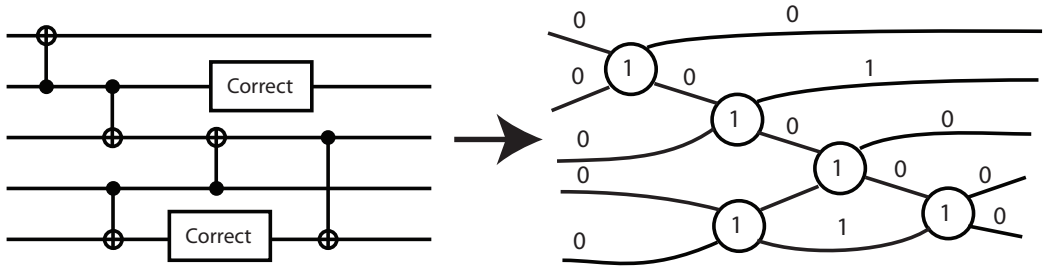


Figure 6.7: Equivalence between a corrected quantum circuit and a multigraph G. The edges are labeled with weights $w(e)$ and the vertices with $d(v)$.

will have $w(e) = 0, \forall e \in E$. An example of such a equivalence is shown in Figure 6.7. Here, vertices are labeled by d and edges by w. We can define a path through this graph from v_0 to v_k as:

$$v_0 \xrightarrow{p} v_k = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} \ldots \xrightarrow{e_{k-1}} v_k \tag{6.2}$$

w and d can be defined over this path p:

$$w(p) = \sum_{i=0}^{k-1} w(e_i) \tag{6.3}$$

$$d(p) = \sum_{i=0}^{k} d(v_i) \tag{6.4}$$

Furthermore, we can define the set P of all paths in G as:

$$p \in P : u \xrightarrow{p} v, \forall (u, v) \in V \times V \ \textit{where } u, v \textit{ are connected} \tag{6.5}$$

There are 2 constraints that are placed on this representation:

D1: $d(v) \geq 0, \forall v \in V$, this means each gate must contribute 0 or more error units

W1: $w(e) \geq 0, \forall e \in E$, this means each edge cannot have a negative number of correction steps

Missing from this list is the original retiming constraint *W2*, which states that for any cyclical path $p \in P, w(p) > 0$ (i.e. all cycles must be "broken" by registers). We can ignore this constraint because we assume all the quantum circuits we optimize are acyclic.

We make this assumption because we unroll all logical circuits with loops in them. All of our quantum circuits that have perceived loops (Figure 7.2 for example), actually use classical values to determine the number of times looped. We assume that a circuit is re-synthesized for each new set of classical loop parameters.

We define a function over G that is equivalent to $EDist_{max}$:

$$\Phi(G) = \max_{p \in P} (d(p) : w(p) = 0) = EDist_{max} \tag{6.6}$$

Algorithm 5 gives a method for computing $\Phi(G)$.

Algorithm 5 CP

G_0 is a subgraph of G with only edges $e \in E$ such that $w(e) = 0$
Go through all $v \in V$ in topological order
if v has no input edges **then**
  Set $\Delta(v) = d(v)$
else
  Set $\Delta(v) = d(v) + \max_{u \in V} \left(\Delta(u) : u \xrightarrow{e} v, w(e) = 0\right)$
end if
$\Phi(G)$ is $\max_{v \in V} \Delta(v)$

Note that $\Delta(v)$ is the EDist for the qubits output from v.

Next, we introduce a recorrection, $r : V \rightarrow \mathbb{Z}$ on the graph G as:

$$G_r = (V, E, d, w_r) \tag{6.7}$$

$$w_r(e) = w(e) + r(v) - r(u), \forall e \in E \text{ and } u \xrightarrow{e} v \tag{6.8}$$

Intuitively, the recorrection (r) map determines how correction steps move through the circuit. $r(u) = n$ means that n corrections move from an output edge to an input edge from the original circuit. We can extend this to paths:

Lemma 1. $w_r(p) = w(p) + r(v) - r(u), \forall p \in P \text{ and } u \xrightarrow{p} v$

Corollary 2 from the retiming work is unnecessary, since our circuits do not have cycles. The corollary states that for any cycle p in G, $w_r(p) = w(p)$.

We define a recorrection r as *legal* if G_r satisfies constraint *W1* and *D1* (*W2* does not apply). The next step is to show the minimal sufficient conditions to r being legal.

Corollary 3. *Let r be a recorrection of G, such that G_r satisfies condition W1, then r is a legal recorrection.*

This is true because condition *D1* is invariant over recorrections and condition *W2* does not apply.

It is not necessary to prove G is equivalent to G_r as is done in the retiming case, since correction steps do not change the functionality of a circuit.

Our goal now is to find a recorrection r of G that minimizes $\Phi(G_r)$. The next step is to define the pairwise aggregates:

$$W(u,v) = \min_{p \in P} \left(w(p) : u \xrightarrow{p} v \right) \tag{6.9}$$

$$D(u,v) = \max_{p \in P} \left(d(p) : u \xrightarrow{p} v, w(p) = W(u,v) \right) \tag{6.10}$$

The next step is to relate Φ and D:

Lemma 4. *The two are equivalent for any integer c:*

$$\Phi(G) < c \tag{6.11}$$

$$\forall u, v \in V, if\ D(u,v) > c, then\ W(u,v) \geq 1 \tag{6.12}$$

Algorithm 6 gives a method for computing W and D.

Algorithm 6 WD: compute $W(u,v)$ and $D(u,v)$ for connected verts $u, v \in V$

Weight edge e with source u with the ordered pair: $(w(e), -d(u))$
Compute the all-pairs shortest paths over weighted edges
for Each shortest path (x,y) for (u,v) **do**
  $W(u,v)$ set to x
  $D(u,v)$ set to $d(v) - y$
end for

Next, we note W and D are useful under recorrection:

Lemma 5. *Given W and D for G, if r is a legal recorrection of G, and if W_r and D_r are analogous for G_r, then:*

1. *The critical paths of G_r and G are the same*

2. $W_r(u,v) = W(u,v) + r(v) - r(u), \forall u,v \in V$

3. $D_r(u,v) = D(u,v), \forall u,v \in V$

Since D is invariant under recorrection, we can make the following observation:

Corollary 6. *If r is a recorrection of G, the $EDist_{threshold}$, $\Phi(G_r)$, is $D(u,v)$ for some $u,v \in V$.*

Next we give the conditions under which a recorrection produces a circuit whose $\Phi(G_r) \leq c$

Theorem 7. *r is a legal recorrection of G such that $\Phi(G_r) \leq c$ iff*

1. $r(u) - r(v) \leq w(e), \forall e \in E$

2. $r(u) - r(v) \leq W(u,v) - 1, \forall u,v \in V$ *such that* $D(u,v) > c$

Algorithm 7 FEAS: given G and $EDist_{threshold}$ c

for $v \in V$ **do**
  Set $r(v) = 0$
end for
for $|V| - 1$ times **do**
  Compute G_r for the current r
  Run Algorithm CP on G_r to get $\Delta(v), \forall v \in V$
  $\forall v : \Delta(v) > c$, set $r(v) = r(v) + 1$
end for
Run Algorithm CP again to make sure $EDist_{max} \leq EDist_{threshold}$, otherwise r is not legal

The constraints above are solvable by a Bellman-Ford shortest path algorithm. One solution to find the best recorrection is to binary search $D(u,v)$s for a minimal $\Phi(G_r)$, using Bellman-Ford to check each $D(u,v)$ for a legal r. Instead of using Bellman-Ford to test $D(u,v)$, however, we use a more efficient method, given by Algorithm 7.

Satisfying $EDist_{threshold}$

So far, we have followed the derivation in [59] to get information about a corrected circuits $EDist_{max}$ (or in the classical case, clock period). We will now depart from this to figure out how to insert correction steps to satisfy a $EDist_{max} \leq EDist_{threshold}$.

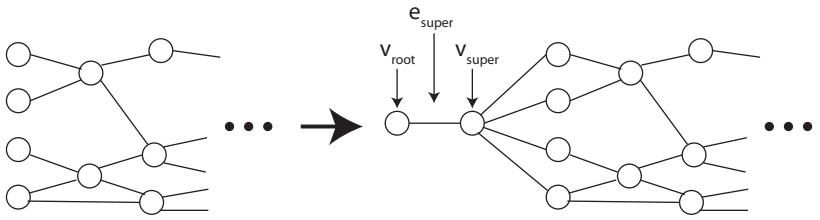


Figure 6.8: In order to create our rooted graph, we add two vertices v_{super} and v_{root} and edge e_{super}. $w(e_{super})$ with then be the number of correction steps on e_{super}.

We first need a place to insert our new correction steps so they have the flexibility to go anywhere in the circuit. We create a new multigraph that we will call a *rooted graph* and designate a place to put the corrections. We define S to be the set of all source vertices in V (vertices with no input edges). We add a new vertex that we call a *supernode*, v_{super}, and connect this vertex to each of the source vertices. For each $v \in S$, we add an edge e, such that:

$$v_{super} \xrightarrow{e} v, and\ w(e) = 0 \tag{6.13}$$

Next we add an additional vertex, v_{root} that connects to v_{super} with edge e_{super} : $v_{root} \xrightarrow{e_{super}} v_{super}$. Figure 6.8 shows this augmented graph. e_{super} is the starting point for all the correction steps we will insert into the circuit, specified by $w(e_{super})$. The idea is to binary search the settings to $w(e_{super})$ until we get the minimal $w(e_{super})$ that gives us the desired $EDist_{threshold}$. We are now in a position to extend Algorithm OPT2 from the original work to do this reverse mapping in Algorithm 8.

We call this process *RECORRECT*. The derivation very closely follows that of [59]. The key insight was simply that the $w(e)$ counters could be mapped to the correction steps on qubits between gates and that $d(v)$ could correspond to gate error instead of delay. From this, the r map moves correction steps around the same way it moved synchronous registers. Equation 6.8 guarantees that the number of correction steps in the system stays constant. The relaxation algorithm, 7, shifts corrections in order to redistribute and minimize $EDist_{max}$.

Since Algorithm OPT2INV has a similar structure to Algorithm OPT2 from [59], the runtime is similar as well. We note that we added an extra binary search for the number of correction stages inserted ($w(e_{super})$), so the runtime increases by a factor of

Algorithm 8 OPT2INV: Input: Graph G and $EDist_{threshold}$

Create rooted graph H from G
while Binary searching $w(e_{super})$ in H for min that satisfies $EDist_{max} \leq EDist_{threshold}$ **do**
  Compute W and D with Algorithm WD
  while Binary searching $D(u, v)$ **do**
    Test $D(u, v)$ for feasibility using Algorithm FEAS
  end while
  Minimal feasible D is $EDist_{max}$
  Save r in $R[w(e_{super})]$
end while
Minimal $w(e_{super})$ that produces $EDist_{max} \leq EDist_{threshold}$ is number of corrections necessary
$R[w(e_{super})]$ is the optimal recorrection

$\log |E|$, giving us a total runtime of $O(|V||E| \log |V| \log |E|)$.

6.3.4 Recorrection and Real Error Probability

There are still a few unknowns in this inner optimization loop that we do not know a priori:

- What final probability of a unrecoverable error on the output do we want?
- There is no exact mapping between the maximum allowed error count $EDist_{threshold}$, and the target failure probability. We could reason that it is more than $1 - (1 - err_{gate})^c$, but there is no reliable mapping.

Since different circuits have different reliability requirements, we rely on the user of our flow to specify the expected output success probability, p_{final}. It is assumed that p_{final} is an achievable value. One way to guarantee this is to set p_{final} to be equal to or less than $p_{success}$ for the unoptimized, correction version of the design. p_{final} determines $EDist_{threshold}$ but due to a complex non-linear relationship, we rely on a full error simulation like one of the methods described in Chapter 4 to pick the $EDist_{threshold}$ that yields the proper p_{final}. Figure 6.9 shows this entire flow. Figure 6.10 shows an example of this

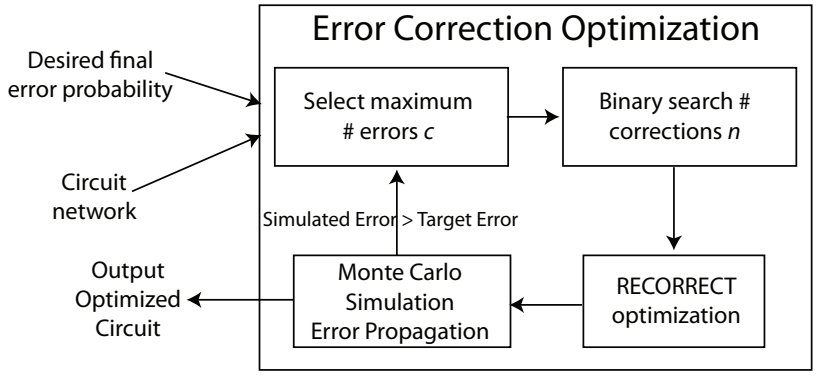


Figure 6.9: The entire circuit recorrection optimization procedure: it takes in the circuit specification and a goal maximum failure probability and tunes the optimization parameters accordingly. $EDist_{threshold}$ and n are selected by binary search.

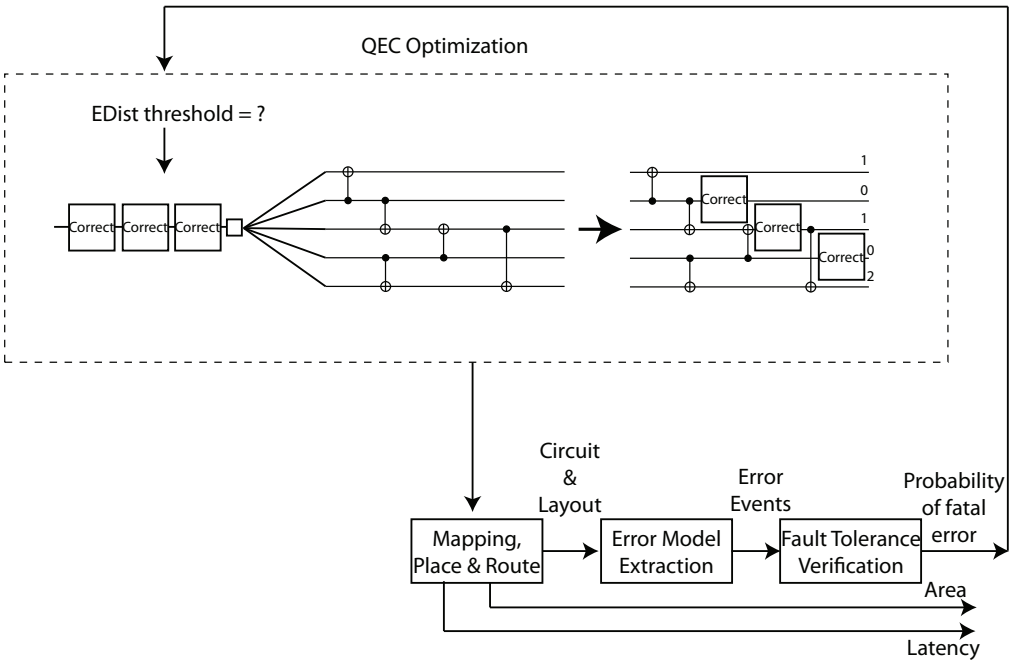


Figure 6.10: The entire circuit recorrection optimization procedure: it takes in the circuit specification and a goal maximum failure probability and tunes the optimization parameters accordingly. $EDist_{threshold}$ and n are selected by binary search.

process of an "outer loop" selecting a target success probability or fatal error probability and searching for the best $EDist_{threshold}$. The "inner loop" searches for the minimal number of correction steps to achieve that threshold.

The complexity of our flow consists of a binary search over the $EDist_{threshold}$ parameter, doing a run of the RECORRECT algorithm each time. Furthermore, for each $EDist_{threshold}$, we perform a full error simulation. If the number of gates in the circuit is d, this gives us a computational complexity of:

$$O(\log d \times (O(RECORRECT) + O(simulate))) \tag{6.14}$$

The $\log d$ factors correspond to binary searches for the right $EDist_{threshold}$.

6.3.5 Results for Random Networks

To test out the effectiveness of our optimization flow, we use both random circuits and some real quantum applications. We discuss the results of our tools on applications like adders and Shor's algorithm later in Chapter 7, but for now we look at performance on random circuits encoded with one level of Steane's [[7,1,3]] code. Our random circuit generation technique used is summarized in Section 2.2.3.

In Figure 6.11 and 6.12, we compare the number of physical gates and success probability in encoded random circuits of varying sizes and 2 circuit splitting fractions. The "unopt" lines are for circuits in which we correct after every logical gate, and for "opt" lines we perform recorrection with an $EDist_{threshold}$ of 3. We see that for larger circuits, the recorrection optimization gives us a factor of 3x in encoded operation reduction for circuits with both Rent's parameters. The probability of error degrades by 2% for the circuits with a Rent's parameter of 0.1 and 4% for the circuit with a Rent's parameter of 0.5.

The results above do not actually show the recorrection performed at the optimal $EDist$ parameter (which sets the distances between error corrections). We simply chose an $EDist_{threshold}$ of 3, which is pretty good for these types of random circuits. Figure 6.13 shows quality of the optimization as a function of EDist for an average of 5 different 1500 logical gate random circuits. In this figure, $EDist_{threshold} = 1$ corresponds to the unoptimized circuit in which we correct after every gate. The success probability degrades minimally at an $EDist_{threshold} = 3$, and then falls off after that. Even though this EDist is relatively small, we see that the number of operations falls off most dramatically going

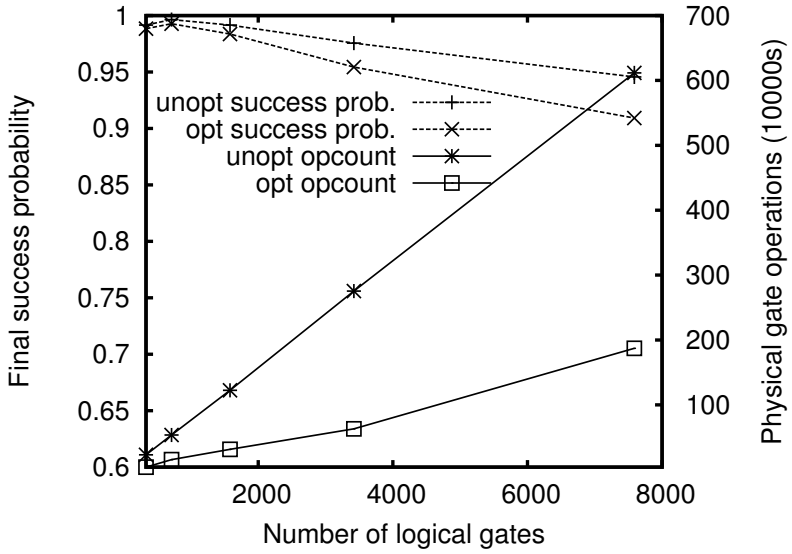


Figure 6.11: Circuit element count and success probability for random circuits with splitting fraction 0.5, with and without recorrection. Each point corresponds to the average number of physical operations in the circuit over 5 random circuits. We set $EDist_{threshold} = 3$ for this optimization.

from 1 to 3. This means that even if we are not willing to sacrifice success probability, we can still achieve a large amount of operation count savings.

6.3.6 Effect of Non-Gate Errors

The next question we ask is: what happens if we include errors from other sources that are not accounted for by our error counting model? It turns out that we can still do pretty well. In Figures 6.14 and 6.15, we plot the success probability of some ≈ 1500 logical gate circuits as a function of movement and idle error strength. In both figures, we see that the recorrection optimized version's success probability closely tracks the unoptimized version from the regions of high probability down to zero.

The recorrected success probability diverges by 10% at one point in the idle error curve in Figure 6.15, as both configurations roll off at 10^{-5}. Most of the rest of the points show less probability variation between the optimized and unoptimized versions of the circuit. This indicates that even though recorrection does not account for idle and movement error in its fault model, it still tracks the unoptimized version closely. One pos-

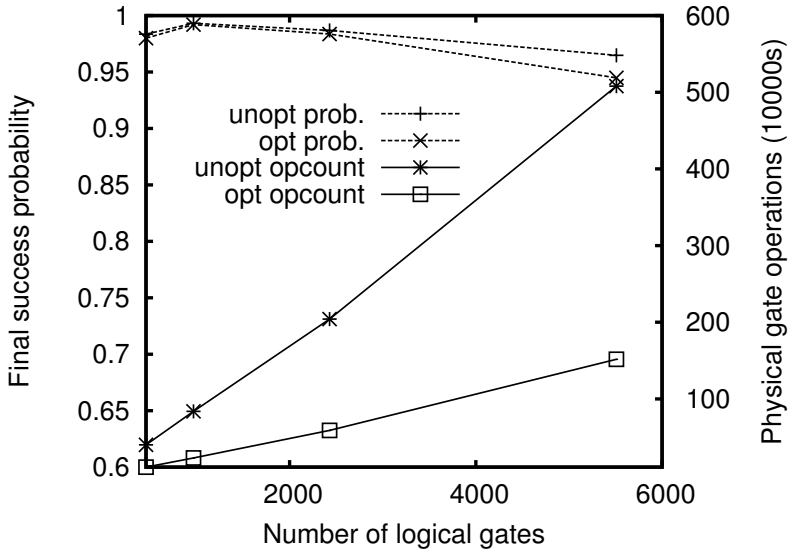


Figure 6.12: Circuit element count and success probability for random circuits with splitting fraction 0.9, with and without recorrection. Each point corresponds to the average number of physical operations in the circuit over 5 random circuits. We set $EDist_{threshold} = 3$ for this optimization.

sible explanation for this is that both correction placement techniques do not account for idle and movement error so there is no reason why one should be better than the other. We still might expect the unoptimized version to have a more noticeable improvement over the optimized one since having extra corrections might correct large movement and idle errors unintentionally. The flip side to this is that additional corrections could increase both area and latency of the design and introduce more movement and idle error sources.

6.3.7 Limitations

This optimization technique is rather simple. First of all, $EDist_{threshold}$ and n are simply determined by brute force search and trials of candidate values against a full simulation. Ideally, we would derive these values from circuit structure or at least get a good first estimate from the structure. Second, $EDist_{threshold}$ is set to be constant across the entire circuit so even if the structure varies radically in different circuit regions, it will be chosen so as to be the best overall fit that provides the minimum success probability.

We made the assumption a while back that the best thing to optimize in this model

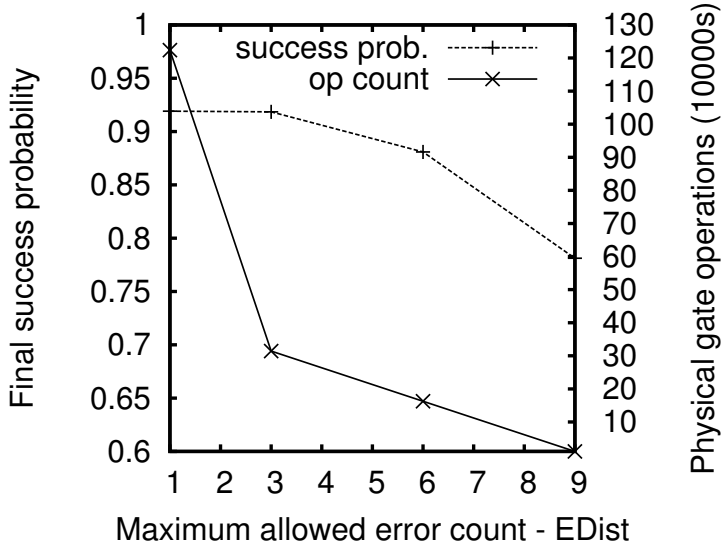


Figure 6.13: Circuit element count and success probability for random circuits with approximately 1500 logical gates, under recorrecting optimization. The maximum error count parameter $EDist$ is being varied and each point is an average over 10 random circuits. The correct-after-every-gate points for success probability and gate count correspond to the points on the left edge. The point for the success probability with no correction or encoding is also shown on the far right as $EDist_{threshold} = 1$.

was the maximum EDist in the circuit, or $EDist_{threshold}$. We tried to develop some intuition earlier in the last chapter as to why this is a good thing to optimize, but it is not clear that another metric might not do even better. There are other optimization fitness functions, such as the overall sum of all the error counts in the circuit of the sum of all error counts going into a correction stage. We will continue with our minimization of the $EDist_{threshold}$ since it matches our intuition and naturally lends itself to the retiming solution, but future work may be to try other fitness functions.

We mentioned in Section 6.3.4 that we were optimizing our circuit for a particular p_{final} target. The outer loop of our optimization is producing a layout, delay estimate in order to produce our final success probability, p_{final}. We could instead opt to use a different metric such as ADCR to drive our search for a good $EDist_{threshold}$. Since this chapter was mostly about establishing our optimization as a good way to reduce operation count without impacting success probability. In Chapter 7 we will discuss using ADCR as

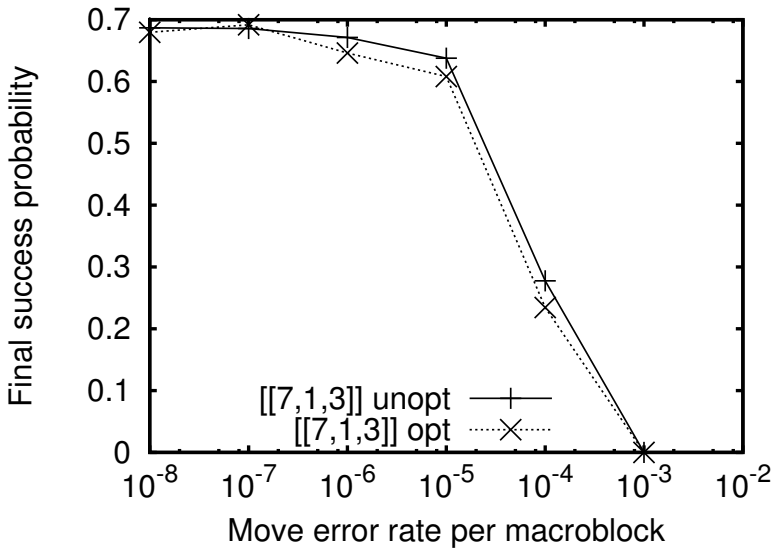


Figure 6.14: Probability of success as we vary the base movement error rate in a distance based movement error model. The gate and idle error rates are set to constant values of 10^{-3} and 10^{-7} respectively. $EDist_{threshold} = 3$

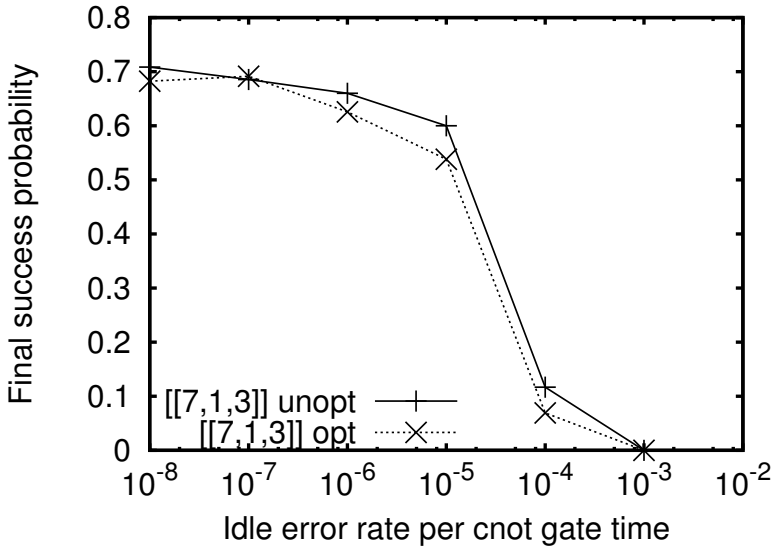


Figure 6.15: Probability of success as we vary the base idle error rate. The gate and movement error rates are set to constant values of 10^{-3} and 10^{-7} respectively. $EDist_{threshold} = 3$

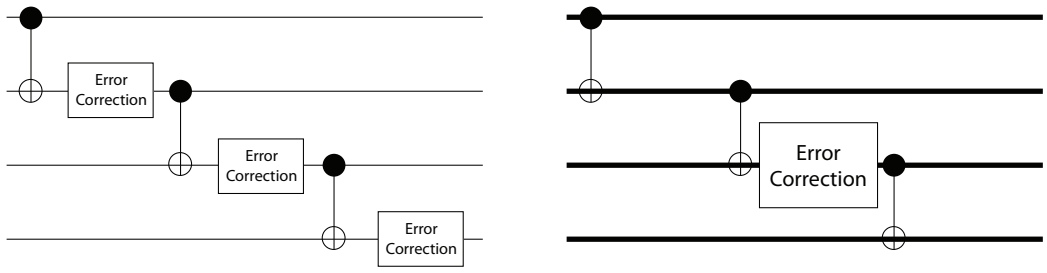


Figure 6.16: We can attain similar levels of fault tolerance by either correcting often with a weaker code, or correcting less often with a stronger code.

a driver for our $ADCR - optimal$ optimization levels.

Finally, as we mentioned above, none of the error correction placement techniques are sensitive to non-gate error sources in the design. We have shown that this is not a big problem for cases in which the gate error rate is dominant, but we might be able to get better success probability if we were to account for these added factors.

6.4 Code Selection and Optimization

Besides the variation of parameters in the optimization phase we have additional flexibility in trading off area/operations used for fault tolerance. The QEC synthesis phase allows us to select different codes to insert into a design as we showed in Section 5.2. Combining our optimization step with the code synthesis step gives us additional freedom to decide how many gates are used in the error correction subsystem.

Figure 6.16 gives an example of this trade-off. In general, we would expect data encoded in a stronger code to be able to go for longer and still be successfully corrected since more errors can be tolerated. Revisiting Figure 6.6, one would expect a $[[25, 1, 5]]$ code to tolerate twice the error count as a $[[7, 1, 3]]$ code, since it can correct 2 errors instead of one.

Revisiting the graphs in Figures 5.9, 6.15, and 6.14, we are now in a position to compare the performance of unoptimized and recorrected versions of the 3 codes we identified as most promising in Chapter 5.

Figures 6.17 and 6.18, show how our 3 top performing codes perform as a function of idle and movement error, with and without recorrection. In terms of movement impact,

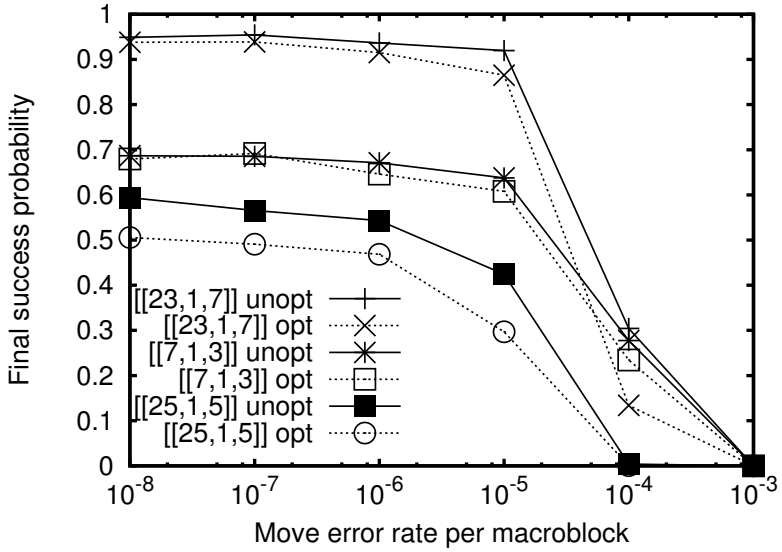


Figure 6.17: Probability of success as we vary the base movement error rate in a distance based movement error model. The gate and idle error rates are set to constant values of 10^{-3} and 10^{-7} respectively. $EDist_{threshold} = 3$

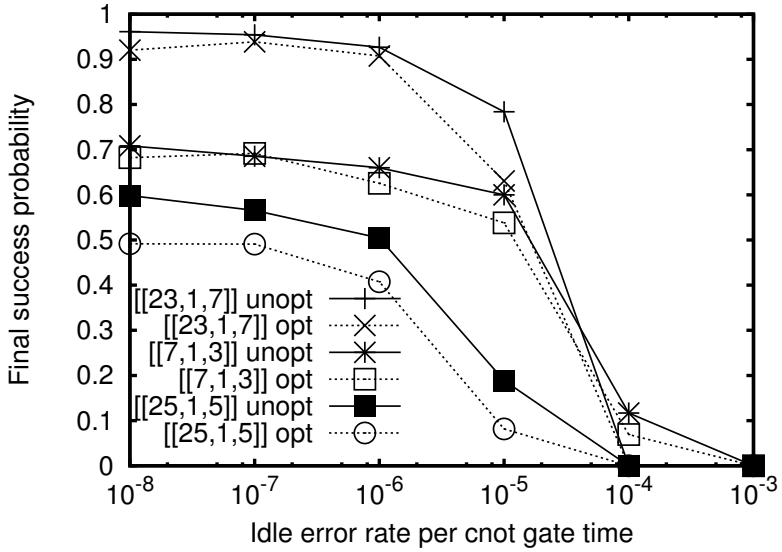


Figure 6.18: Probability of success as we vary the base idle error rate. The gate and movement error rates are set to constant values of 10^{-3} and 10^{-7} respectively. $EDist_{threshold} = 3$

Figure 6.17 shows us that the success probability for optimized and unoptimized versions of all the codes falls off at an error rate around 10^{-5}. The optimized version of the $[[25, 1, 5]]$ code is the worst performing fault tolerant design for all movement rates. Additionally, the optimized version of $[[25, 1, 5]]$ diverges the most from the unoptimized version. This indicates that the $[[25, 1, 5]]$ code is a poor candidate for recorrection. The $[[7, 1, 3]]$ code has the closest tracking optimized and unoptimized versions in general.

We have introduced our recorrection optimization, which is based on classical circuit retiming and have shown how it substantially reduces the number of resources necessary for a fault tolerant architecture with minimal sacrifice of error probability. We show that a $[[7, 1, 3]]$ encoded architecture can maintain essentially the same success probability on a set of random circuits while cutting the total number of operations by 4x. We also showed that none of the recorrected versions of circuit encoded in different codes are substantially more sensitive to different idle and movement error parameters, compared to unoptimized versions. The recorrected $[[25, 1, 5]]$ encoded circuits had the worst performance. From this analysis, we conclude that recorrecting $[[7, 1, 3]]$ and $[[23, 1, 7]]$ encoded circuits are a good way to substantially reduce error correction overhead while minimally affecting success probability. Additionally, we find that modest levels of recorrection are all that is necessary to substantially reduce the QEC resources.

Chapter 7

Fault Tolerant Optimization and Analysis for Large Circuits

In previous chapters, we developed tools to automatically synthesize QEC encoded circuits, optimize the inserted QEC modules, verify their fault tolerant properties, and place and route gates to generate layouts. We are now ready to put all this together to analyze the performance of large circuits. Figure 7.1 shows the our target application: Shor's factorization algorithm. The predominant component of this algorithm is modular exponentiation, and our implementation of modular exponentiation consists of repeated additions. Therefore, to start our study of large circuits we will focus on the main kernel used to perform Shor's algorithm: a quantum adder. Once we decide on a good adder design, we will use this to build the full Shor's factorization application. Therefore, the

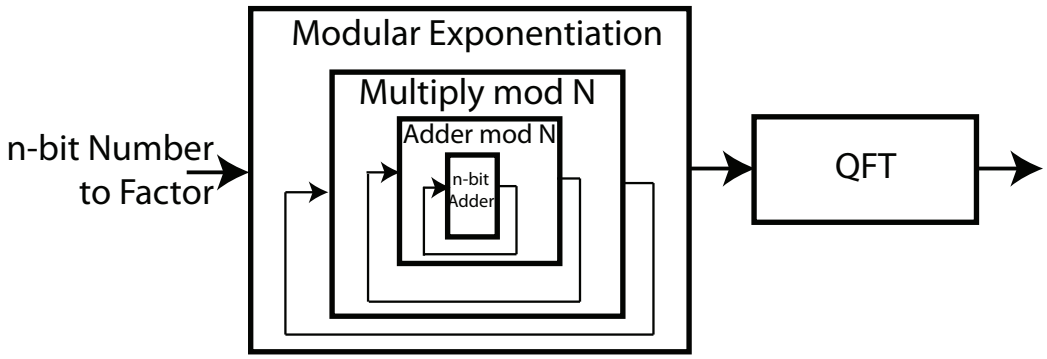


Figure 7.1: *Shor's factoring* architecture.

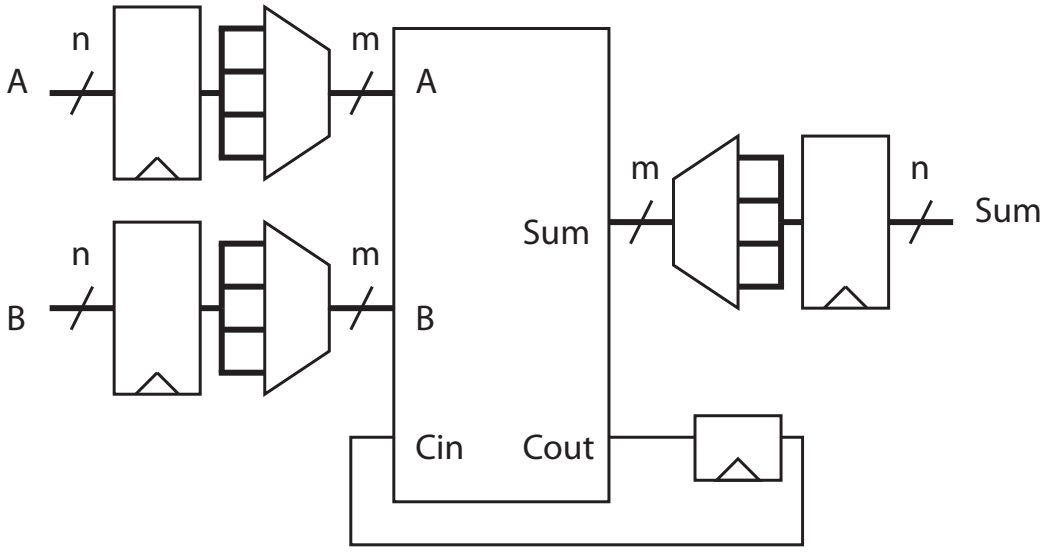


Figure 7.2: Quantum ripple carry adder with "subadder" serialization

goals of this chapter are to find the best adder circuit implementations, optimization levels, encodings, and datapaths for good ADCR-efficient designs.

7.1 Quantum Addition Circuits

The quantum adder is a fundamental component of Shor's factorization. Consequently, this section will apply the machinery that we developed in previous sections to produce optimized adder circuits. Since we are targeting 1024-bit factorization, we will examine 1024-bit adders. Further, for non-random circuits, we will switch to the more realistic Error Set 1 from Table 1.1.

7.1.1 Adder Implementation

We evaluate the quantum ripple-carry adder (QRCA) [29] and the quantum carry look-ahead adder (QCLA) [30], constructing larger adders from smaller adder modules, similar to what is done with classical bit-serial adders. Figure 7.2 shows how an n-bit QRCA is constructed with multiple passes through a single m-bit sub-adder. The registers can map to memory regions and the adder block to compute regions such that data is shuttled between memory and compute when it uses the sub-adder. Similarly, Figure 7.3 shows how an n-bit QCLA is constructed with smaller modules. The modular approach allows us to

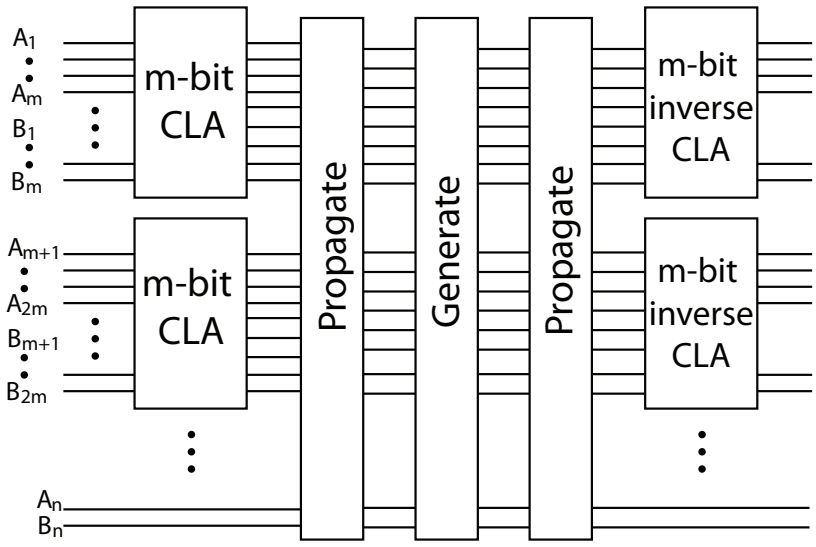


Figure 7.3: Quantum carry-lookahead adder

trade area for parallelism thus allowing us to construct optimal adder configurations.

All the following results on the QRCA and QCLA are for 1024-bit sized adders, unless otherwise indicated.

7.1.2 Adder Performance

We begin by observing how our recorrection optimization affects success probability and physical operation counts for the QCLA and QRCA adders. Figure 7.4 shows this relationship as $EDist_{threshold}$ is varied. We first note that the QRCA exhibits a uniformly higher success probability. This is due to the fact that there is really just a single main fault path through the circuit: the ripple carry chain. All the other qubits experience limited activity in terms of both gate and movement error. The number of fault paths are limited to mainly to those along the ripple carry chain. The QCLA, on the other hand, exhibits much more parallelism, performing gates and movement on a much wider variety of qubits. This leads to may more possible fault paths, thus decreasing the overall probability of success. In this particular error model, the extra idle time due to the higher latency of the QRCA does not counteract the success probability savings due to fewer fault paths.

If we look at the error events operating on the failing encoded qubit blocks, idle errors are 20000 times more likely than other types of errors! As in many of our examples,

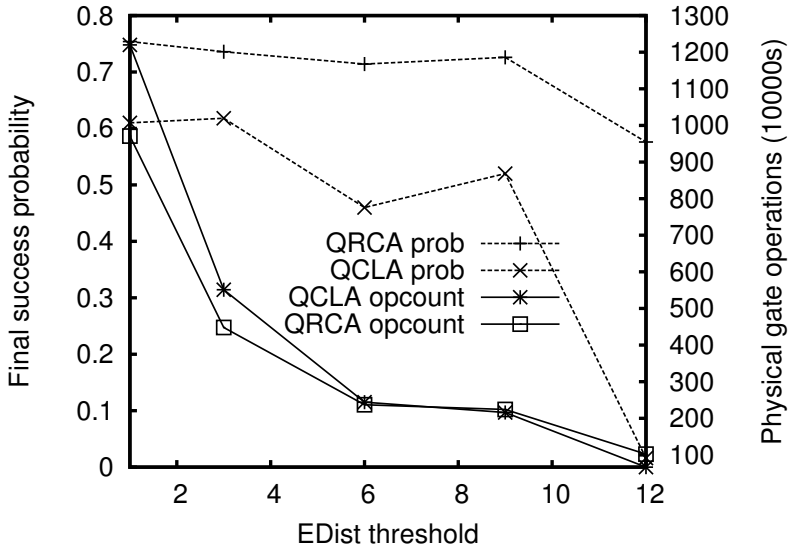


Figure 7.4: Success probability and operation count for QCLA and QRCA adders as a function of $EDist_{threshold}$. The underlying architecture is Qalypso and the $[[7, 1, 3]]$ code was used.

we assume that gate errors are 100 times more likely than idle errors, so therefore the overall idle error probability is 200x more than the gate error. This also highlights the importance of error correction insertion techniques that are sensitive to other error types.

Note also that the success probability for the QCLA with the highest level of optimization ($EDist_{threshold} = 12$) drops rapidly to near zero. This is because at this point, *all* correction steps have been removed from the entire circuit. Since the QRCA has a much longer critical path of gates, there are still error correction steps along the most error prone part of the circuit for this threshold. This is why the success probability for QRCA with $EDist_{threshold} = 12$ stays relatively high.

If we look at the reduction in operation counts between the two adder types, we see that the QCLA initially does more work and as more correction steps are removed, the two adders settle to an approximately equal number of operations. This is because the majority of the operations performed are correction steps, so when the majority of correction steps are removed, the two adders have approximately the same number of physical gates.

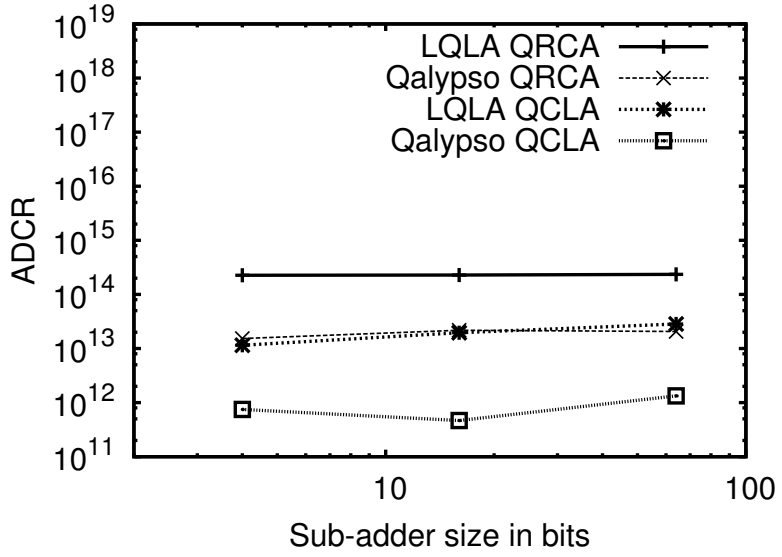


Figure 7.5: Ripple carry (QRCA) and carry-lookahead (QCLA) adders implemented on Qalypso and LQLA datapaths. The QEC used was the Steane $[[7, 1, 3]]$ code. The parameters searched to find the ADCR-optimal values were recorrection $EDist_{threshold}$ and the number of compute regions.

Impact of Datapaths on Adders

The above analysis shows that QRCA has a better success probability and an equal number of operations that needs to be performed. From these two metrics, we would conclude that it is the better adder to use for Shor's factorization. There is still the issue of latency, however, and we will see that operation count does not always directly translate to area.

Thus, we study the ADCR of different adder designs, QEC optimization levels, encodings, and microarchitectures. First, since there are a number of parameters that must be set in all these models, we introduce the notion of *ADCR-optimal*. This is simply the best ADCR we could find over all free parameter combinations. In the following graphs we indicate what free parameters we searched in order to find the ADCR-optimal values.

Figure 7.5 compares the QRCA and the QCLA from Section 7.1.1 on the two different quantum computer datapaths introduced in Section 3.4.1. We see that "Qalypso QCLA" is the best performing choice, in terms of ADCR, for all the sub-adder configurations considered. Furthermore, we see that the QCLA always outperforms the QRCA on the same

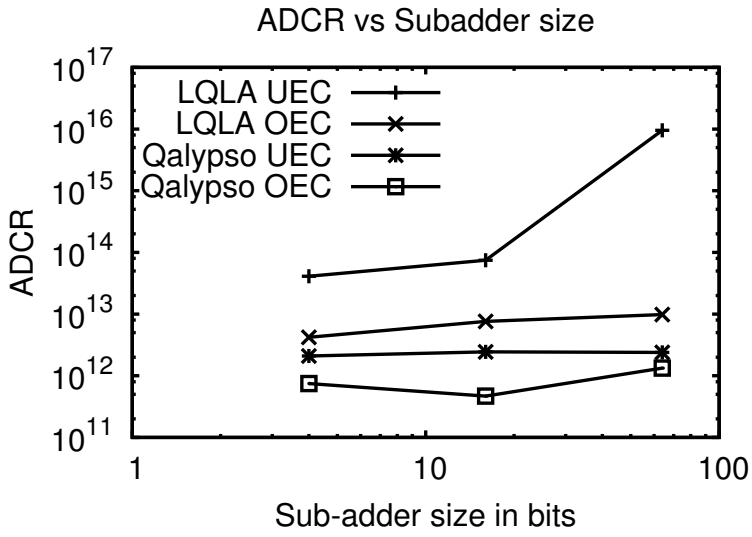


Figure 7.6: Carry-lookahead (QCLA) adders implemented on Qalypso and LQLA datapaths. The QEC used was the Steane $[[7, 1, 3]]$ code. We show the unoptimized and recorrection optimized to the ADCR-optimal $EDist_{threshold}$. In both cases we picked the ADCR-optimal compute region counts.

metric and Qalypso always outperforms LQLA. In all these cases, the success probability stays relatively high and therefore does not have that much impact on the different ADCRs, the main impact is from area and latency.

When compared to LQLA, Qalypso's ADCR for both adders is better since, as mentioned in Section 3.4.2, it has much more flexibility in distributing ancilla resources throughout compute regions. The gain in ADCR for Qalypso is predominantly due to a decrease in area, since Qalypso cuts out much of the unused ancilla generation area built into the more static LQLA design.

Finally, we notice the sub-adder structure of the our adder circuit has little effect in this case on the overall ADCR. We reason this is because the majority of the time difference between ballistic movement within a larger sub-adder and teleportation between smaller sub-adders is not a significant component of the overall latency. The 16-bit sub-adder version does marginally better for Qalypso QCLA, so we will use that structure moving forward.

The next question is: what is the impact of our recorrection optimization in re-

lation to these different microarchitectures? Figure 7.6 shows a 5x ADCR improvement in ADCR for Qalypso and a 10x improvement for LQLA in at the 16-bit sub-adder configuration. For LQLA, the gains are predominantly in latency. Since ancillae are generated on demand, the QEC ancilla generation is in the critical path, and recorrection removes much of this QEC ancilla need dramatically shortening the critical path. If we look at the increases in ADCR for the unoptimized LQLA design, this further confirms this reasoning. The latency increases as we go from 4 to 64-bit subadders. While the increase in latency is only a factor of 2, the success probability drops dramatically to near 0 for the 64-bit sub-adder size. This increased latency leads to a critical effective idle error point, as we saw in Figure 5.9b, where the success probability falls off dramatically with small additional increases in idle errors. From this, we see another advantage of recorrection, in cases where it can substantially reduce latency, it can substantially improve success probability, especially for long running circuits. In the cases of LQLA, recorrection yields a 2%, 5%, and $> 90\%$ improvement for the 4, 16, and 64-bit subadder structures. This makes sense, since doing unnecessary corrections can cause waiting qubits to lay idle, accumulating error. As mentioned before, an idle-sensitive placement of error correction stages would also work well to combat this problem.

The ADCR gains for Qalypso are more modest, since the baseline unoptimized version already does a better job of managing its ancilla resources. In this case, the gains are more evenly split between latency and area. Additionally, as in the case of LQLA, the success probabilities of the recorrected Qalypso runs are better than the unoptimized case, albeit by a small amount (2-4%).

Optimizing to Improve Reliability

We now turn our attention to interactions between recorrections and other facets of the datapath design. Figure 7.7 shows the overall success probabilities of Qalypso and LQLA as a function of compute regions. The idea is that as more compute regions are added, more parallelism is possible, leading to less qubit idling. In the 1-64 compute region range, Qalypso is idle error bound and therefore, adding more compute regions reduces qubit idle errors and improves the success probability. The 64-256 compute region range corresponds to saturation of the available parallelism in the QCLA and there are no further gains in success probability. When we have 1024 compute regions, the overall area of the

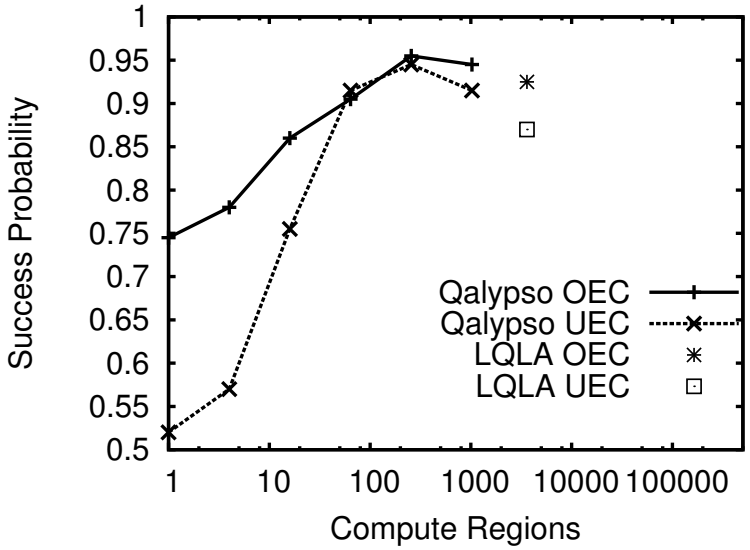


Figure 7.7: Overall success probability for the QCLA adder with and without recorrection, as a function of the number of datapath compute regions. Note that the number of compute regions for LQLA is statically set by the number qubits in the system, so we cannot vary the number of compute regions for that microarchitecture. The recorrected points correspond to the $EDist_{threshold}$s with the highest success probability.

design becomes significant and qubit movement error becomes a factor that degrades success probability. If we were to have as many compute regions as logical qubits, we would meet the LQLA points on the right.

In all these cases, the unoptimized versions of Qalypso and LQLA have lower success probabilities than the recorrected ones. In the idle bound range, this is due to the latency that recorrection eliminates. In the movement-bound range, recorrection reduces the size of the compute regions (by reducing necessary QEC ancilla) and therefore reduces movement error.

Code Selection for Optimized Adders

We can also analyze our adders with different error correcting codes. Figure 7.8 shows the ADCR for the QCLA with different encodings. Since LQLA was only specified for the Steane $[[7, 1, 3]]$ code in the original work and the ancilla generator is specially designed for that code, it only supports that single encoding. If we compare between Qalypso and

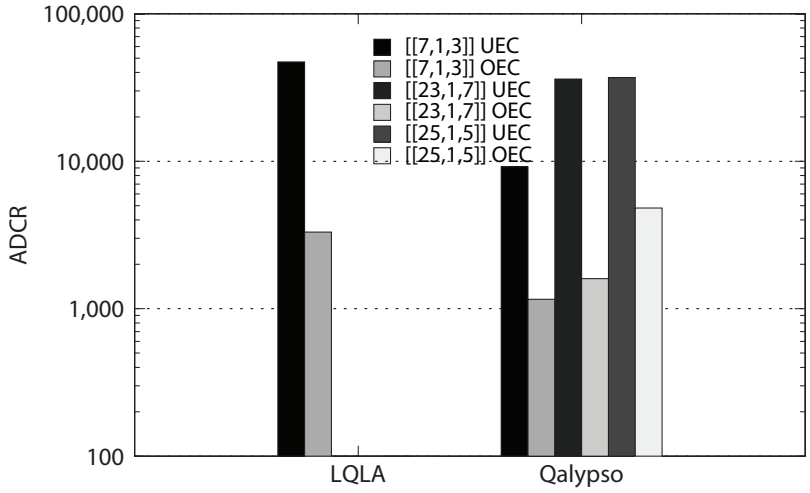


Figure 7.8: ADCR for the QCLA using 3 different codes, with and without recorrection optimization on a 128 bit QCLA adder. The ADCR-optimal values chosen over $EDist_{threshold}$ and the number of compute regions.

LQLA for this code, we see that Qalypso has a better ADCR for both the recorrected and unoptimized versions. This is again due to Qalypso's flexible ancilla provisioning. Within Qalypso, the $[[7, 1, 3]]$ code has the best ADCR. Since it has a more compact structure, it wins in terms of area (the unoptimized $[[7, 1, 3]]$ adder is about 3x smaller than $[[25, 1, 5]]$ and $[[23, 1, 7]]$ encoded adders).

Next we can compare the performance of recorrection for all these cases. The biggest ADCR improvement is for the $[[7, 1, 3]]$ encoded LQLA design. This is not surprising, since the unoptimized LQLA inefficiently uses QEC ancilla generators, so again, there is more room for improvement.

7.1.3 Which adder design to use?

Using our fault tolerance synthesis, optimization, layout and analysis tools, we have provided a survey of the different adder design parameters with respect to an interesting metric, ADCR. We conclude the best design is a $[[7, 1, 3]]$ encoded, recorrected QCLA using the Qalypso datapath. The result is interesting, because if we base our results on just which code provides the most nominal protection from errors, we would probably want to pick the $[[23, 1, 7]]$ encoded version.

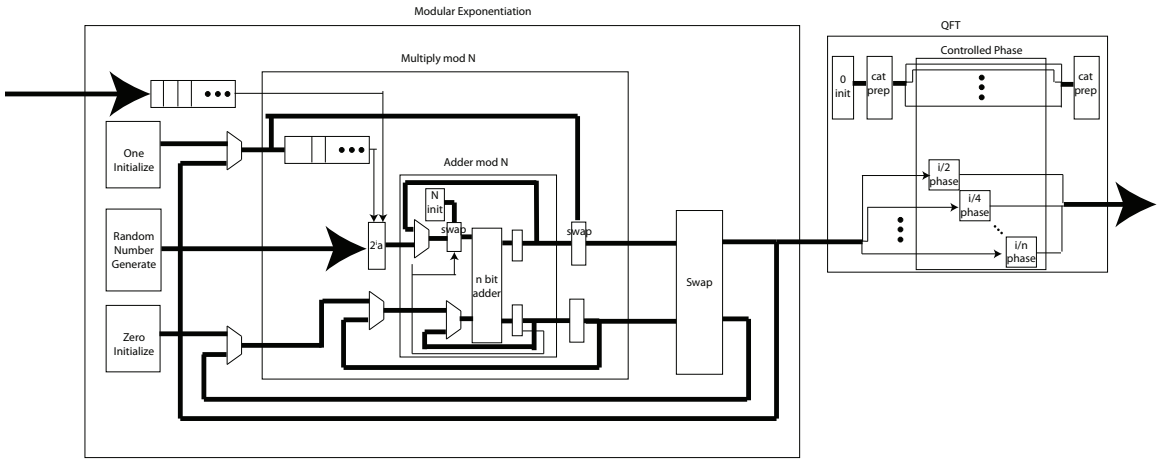


Figure 7.9: *Shor's factoring* architecture.

7.2 Shor's Factorization Algorithm

Using the insight we gained from analyzing the adder designs in the last section, we are now ready to look at potential designs for Shor's factorization algorithm.

7.2.1 Implementation of Shor's

Figure 7.9 shows a block-diagram of our target circuit. It consists of two main components: modular exponentiation and the quantum Fourier transform (QFT). For the modular exponentiation circuit, we rely on the work done in [100] and for the QFT, [33]. Since addition is a key component of the modular exponentiation circuit, we use our best adder designs from Section 7.1.1.

7.2.2 Performance of Shor's Factorization

A previous estimate of the area required to implement Shor's was approximately $1mm^2$ in area [64], in ion trap technology. One goal we have is to improve on this estimate, so that we have candidate implementations that would be more easily implementable.

We start by investigating the aggregate effect of recorrection on full Shor's factorization circuits. Figure 7.10 shows the number of physical operations needed for factoring various sized numbers. We see that recorrection has reduced the number of operations in the QCLA based Shor's by a factor of 2.5 and in the QRCA based Shor's by a factor of 4.

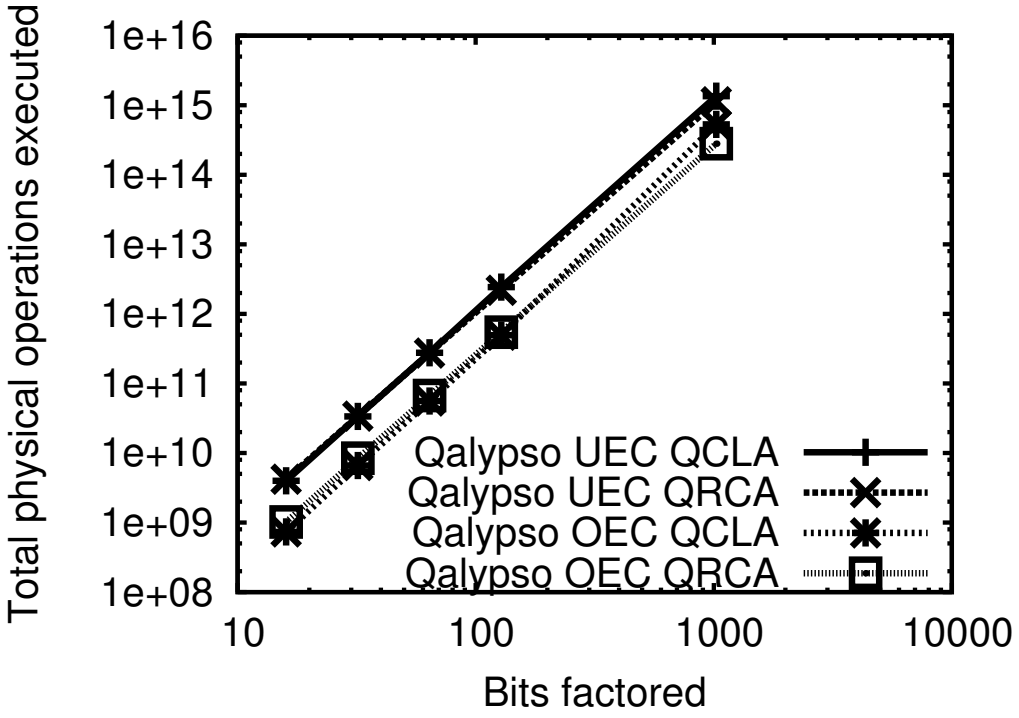


Figure 7.10: Physical operation count of Shor's factorization algorithm using the Steane $[[7, 1, 3]]$ code and either the QCLA or QRCA adders.

In Figure 7.11, we see that recorrection reduces the area of the QCLA based Shor's by a factor of 2.5 and for the QRCA by a factor of 4. We note that these area reductions are in line with the reduction in operation counts from Figure 7.10.

The improvements in latency that recorrection gives us is more modest compared to area. Recorrection reduces the latency of the QRCA based Shor's by a factor of 1.5 and the QCLA version by almost a factor of 2.

We also point out that the QCLA based factoring has consistently better latency over the QRCA version and catches up in area as well for the large circuits.

7.3 Future Work

The simulation, optimization and layout techniques developed in this work get us closer to building large scale, fault tolerant designs for applications like Shor's factorization algorithm. However, there are still many points that require more investigation and

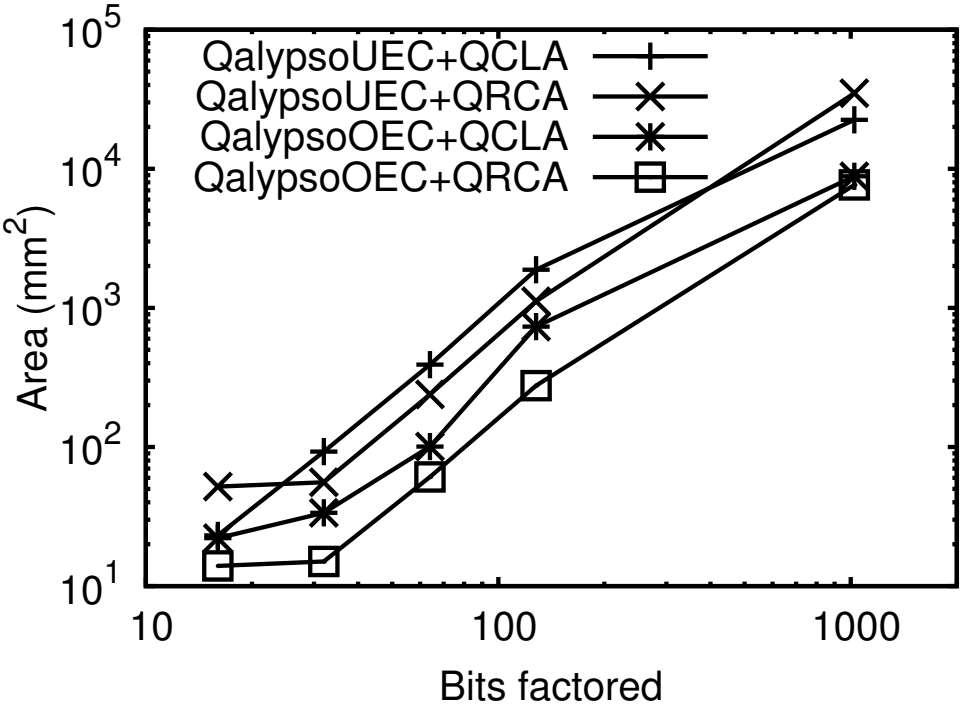


Figure 7.11: Area of Shor's factorization algorithm using the Steane $[[7, 1, 3]]$ code and either the QCLA or QRCA adders.

development.

7.3.1 Faster, Bigger Simulations

We found that our error simulation improvements can handle hundreds of millions of gates (as in the case of 1024 bit adders). However, it is still too slow to scale up to many billions of gates and get results in a reasonable amount of time. After running our vector Monte Carlo simulation on a 1024 bit Shor's factorization algorithm for a week, we opted to stop simulation since it still was not finished. We believe that there is not much room for improvement in the time per simulated instruction, therefore, additional improvements will have to come from simulating fewer overall instructions. The most logical choice is to compute error statistics for each module in our hierarchical specification once and then reuse these statistics for each instance of the module.

Section 4.5 on the joint probability calculation indicates that getting useful error statistics is challenging. If we compute marginal probabilities of error on each qubit

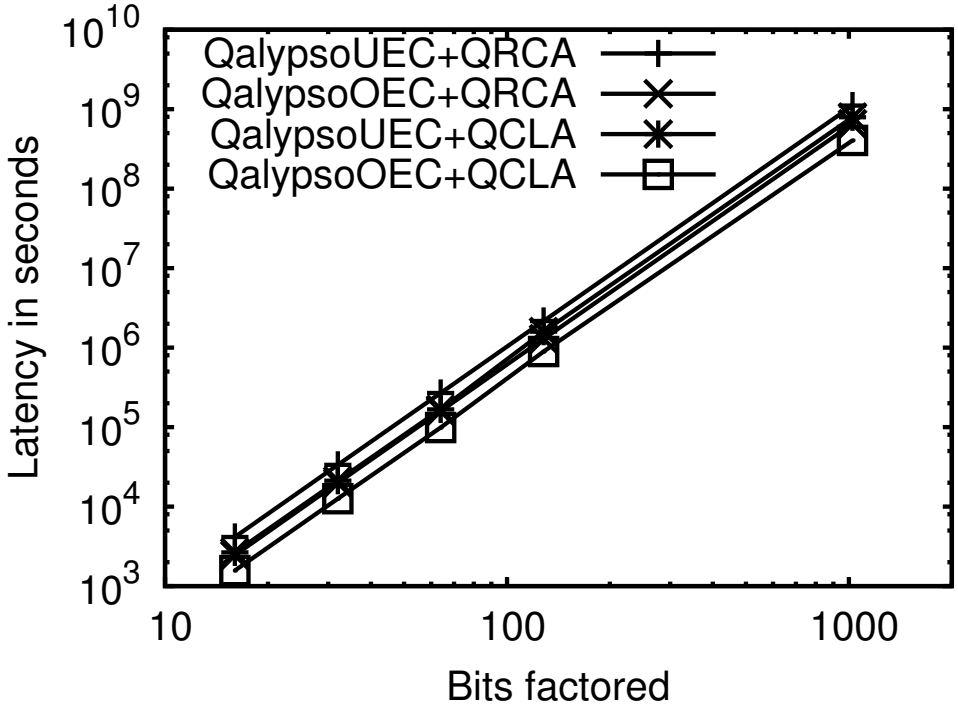


Figure 7.12: Single run latency of Shor's factorization algorithm using the Steane $[[7, 1, 3]]$ code and either the QCLA or QRCA adders.

independently in a module with error correction, we lose critical qubit error dependencies that enable proper error correction simulation. On the other hand, computing a full joint probability over all qubit errors, the problem rapidly becomes intractable for any module with more than a trivial number of qubits. One solution might be a mixed fidelity/Monte Carlo method, similar to what we do now with EPR fidelities for the teleportation network which then interacts with the data in a Monte Carlo simulation. We could push more of the circuit into the simple fidelity model and combine this with the more time consuming Monte Carlo method only at error dependency critical points like during error correction.

7.3.2 Error Model Refinements

Our physical error models could always be improved as well. As more information comes out experimental ion trap work, we can further refine our error parameters and the movement model. We have tried to make our ion trap movement error model as flexible as possible but there are additional improvements that may make sense. So far we have only

considered qubit/ion movement errors in isolation of one another, but if ions are in close proximity, ions could interact with one another in undesirable ways. Through tunneling, they could swap states, for example.

7.3.3 Layout and Mapping to Qalypso

Our circuit layout techniques currently rely on a particular set of heuristics for mapping computation to computation regions and scheduling the times at which everything happens in Qalypso. At this point, our heuristics are relatively simple and could always be improved. This could improve both our overall area and latency, as well as indirectly affecting reliability, by reducing movement and idle errors with a more efficient schedule of qubit movements.

Since our macroblock level layout techniques are already abstracted away somewhat from ion trap technology, it might make sense to look at other quantum computing technologies with all our CAD tools. Not only would this affect the layout heuristics but would also require new error models.

7.3.4 Expanding Code Comparisons

Our code comparison focused on the same CSS codes that were analyzed in previous studies by both Steane and Cross et al. In Section 1.2 we listed many other code categories, many of these codes have only been presented and analyzed in isolation.

One interesting comparison would be to compare CSS codes with Steane-style error correction with Knill's method of postselection with error detecting codes, to really determine accurate overheads of each in terms of area and latency. Post-selection has been shown to yield a higher threshold value than the standard correction method but it also requires a tremendous amount of overhead. It would be interesting to see if these reliability gains remain after all errors are accounted for.

Entanglement-assisted codes might be a natural match to our tiled microarchitecture with teleportation interconnect. The entanglement resources for error correction could be combined with the EPR entanglement resources for the network.

7.3.5 Recorrection Flexibility

Our initial attempt at optimizing error correction step placement in fault tolerant circuits leaves plenty of room for improvement. We currently optimize a hierarchical circuit design by taking a particular global $EDist_{threshold}$ and optimize each module with the same parameter. We would really like to let each module have separate $EDist_{threshold}$s and then try to pick the set of $EDist_{threshold}$s that give us the best ADCR or failure probability.

When we looked at adder circuits, we compared different codes with recorrection to see which optimized encoding has the best overall ADCR performance. The next step would be to allow multiple code domains within the same design. We could imagine that regions with a lot of qubits sitting around idling would be encoded in a code that is particularly good under high relative idle errors (like the $[[7, 1, 3]]$ code). Regions undergoing dense computation would be encoded using a code that works better under high gate error (such as the $[[23, 1, 7]]$ Golay code. Transcoding data within the circuit could be performed using the code teleportation technique used by Thaker et al [98].

When developing the recorrection model, we assumed a simplified error propagation model where gates propagate the maximum input $EDist$ to all outputs (Section 6.2). Additionally, we assumed that the goal of our optimization should be minimize $EDist_{max}$ over the entire circuit. While this strategy makes some intuitive sense, it is not the only solution to this problem. For example, we could imagine a goal of trying to minimize the sum of all $EDist$ values everywhere in the circuit. It would be interesting to see how different optimization strategies affect the overall failure probability.

The results on large adders in Section 7.1.2 are the result of a multi-parameter search over possible Qalypso and LQLA datapath configurations (# of compute regions, region sizes, # of ancilla factories, etc.), different $EDist_{threshold}$ values, and different codes. All these parameters have a non-linear relationship with respect to one another. In general, found good sets of parameter combinations through trial and error. A substantial improvement would be to have a more intelligent way to search this complex space to optimize the parameter of interest.

7.4 Conclusion

We have presented the first full computer aided design flow that is capable of synthesizing, optimizing, laying out, and verifying circuits at the 1024 logical bit scale. Using our design flow, we have designed the first 1024-bit Shor's factorization implementation that has an area substantially less than the previous state-of-the-art design of $0.9m^2$ [64]; our design is only $64cm^2$. This dramatic improvement was made possible by our comprehensive set of tools. Tools that account for all error sources, analyze their impact on overall application failure, and optimize for reliability and resource efficiency.

Our scalable, hierarchical layout techniques provide the first detailed layouts of large scale circuit designs. These layout techniques enable flexible, efficient extraction of error models over millions of qubits. We exploit multiple levels of granularity to simulate "commodity resources" like EPR qubits with aggregate error probabilities and only provide detailed qubit-level error simulations for more critical data qubits.

To use these error models of large circuits, we introduced a bit-parallel Monte Carlo technique that allows the detailed simulation of error in a design 10x faster than the standard Monte Carlo simulation method from other works.

Detailed error models and fast simulation have allowed us to perform the first comparison of error correcting codes using a comprehensive error model consisting of all movement, idle, and gate errors in the encoding and correction circuits. Through this detailed simulation, we have made a few interesting observations. First, in cases of higher movement error probability, concatenating a code does not always bring benefits, as is the case of the level 1 and level 2 $[[7, 1, 3]]$ codes. We showed that for more than minimal movement error the L2 encoded circuit appears to perform substantially worse. Second, while stronger codes like the $[[23, 1, 7]]$ Golay code appear to have clear advantages when considering gate dominant error models, idle intensive error models favor the L1 $[[7, 1, 3]]$ encoded circuits as having the fewest overall failures.

To better evaluate all the important factors that go into building a large quantum application, we introduce a new metric: area-delay product to correct result (ADCR) which gives us an idea of total space and time resources needed to build a reliable system.

To deal with the high overhead of quantum error correction in large circuits, we developed a new optimization technique that positions error correction steps in the optimal positions in a circuit to minimize overall failure probability. This optimization reduces cir-

cuit gate count by 4x while reducing success probability by only a few percent. Furthermore, ADCR is reduced by anywhere between 2 and 1000 times.

We also address QEC resource efficiency at the layout level by providing a new microarchitecture called Qalypso which effectively load-balances ancilla production from factories so that we can devote less layout area to fewer but more heavily utilized ancilla factories.

We have presented a comprehensive set of techniques and results for selecting the best fault tolerant architecture for a given application circuit. We can chose to focus on minimize overall failure probability or our holistic ADCR metric. Either way, our layout, optimization, and error analysis techniques produce smaller, faster designs that still guarantee comparable reliability to the slower, larger designs from the literature.

Bibliography

[1] S. Aaronson and D. Gottesman. Improved simulation of stabilizer circuits. *Physical Review A*, 70(5):52328, 2004.

[2] D. Aharonov and M. Ben-Or. Fault-tolerant quantum computation with constant error. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 176–188. ACM New York, NY, USA, 1997.

[3] D. Aharonov, V. Jones, and Z. Landau. A polynomial quantum algorithm for approximating the Jones polynomial. *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 427–436, 2006.

[4] P. Aliferis and A.W. Cross. Subsystem Fault Tolerance with the Bacon-Shor Code. *Physical Review Letters*, 98(22):220502, 2007.

[5] P. Aliferis, D. Gottesman, and J. Preskill. Quantum accuracy threshold for concatenated distance-3 codes. *Arxiv preprint quant-ph/0504218*, 2005.

[6] P. Aliferis and J. Preskill. Fault-tolerant quantum computation against biased noise. *Phys. Rev. A*, 78:052331, 2008.

[7] D. Bacon. Operator quantum error-correcting subsystems for self-correcting quantum memories. *Physical Review A*, 73(1):12340, 2006.

[8] S. Balensiefer, L. Kregor-Stickles, and M. Oskin. An evaluation framework and instruction set architecture for ion-trap based quantum micro-architectures. *Proc. 32nd Annual International Symposium on Computer Architecture*, 2005.

[9] A. Barenco, C.H. Bennett, R. Cleve, D.P. DiVincenzo, N. Margolus, P. Shor,

T. Sleator, J.A. Smolin, and H. Weinfurter. Elementary gates for quantum computation. *Physical Review A*, 52(5):3457–3467, 1995.

[10] C.H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W.K. Wootters. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Physical Review Letters*, 70(13):1895–1899, 1993.

[11] C.H. Bennett, G. Brassard, S. Popescu, B. Schumacher, J.A. Smolin, and W.K. Wootters. Purification of Noisy Entanglement and Faithful Teleportation via Noisy Channels. *Physical Review Letters*, 76(5):722–725, 1996.

[12] V. Betz. *VPR and T-VPack User's Manual*. http://www.eecg.toronto.edu/ vaughn/vpr/vpr.html, 2000.

[13] V. Betz and J. Rose. VPR: A New Packing, Placement and Routing Tool for FPGA Research. *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, pages 213–222, 1997.

[14] DJ Bishop, CR Giles, and GP Austin. The Lucent LambdaRouter: MEMS technology of the future here today. *Communications Magazine, IEEE*, 40(3):75–79, 2002.

[15] PO Boykin, T. Mor, M. Pulver, V. Roychowdhury, and F. Vatan. On universal and fault-tolerant quantum computing: a novel basisand a new constructive proof of universality for Shor's basis. *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 486–494, 1999.

[16] SB Bravyi and A.Y. Kitaev. Quantum codes on a lattice with boundary. *Arxiv preprint quant-ph/9811052*, 1998.

[17] T. Brun, I. Devetak, and M.H. Hsieh. Catalytic quantum error correction. *Arxiv preprint quant-ph/0608027*, 2006.

[18] AR Calderbank and P.W. Shor. Good quantum error-correcting codes exist. *Physical Review A*, 54(2):1098–1105, 1996.

[19] W.S. Carter, K. Duong, R.H. Freeman, H.C. Hsieh, J.Y. Ja, J.E. Mahoney, L.T. Ngo, and S.L. Sze. A User Programmable Reconfigurable Logic Array. In *Proceedings of the IEEE 1986 Custom Integrated Circuits Conference*, pages 233–235, 1986.

[20] C.M. Caves. Quantum Error Correction and Reversible Operations. *Journal of Superconductivity*, 12(6):707–718, 1999.

[21] E. Chi, S.A. Lyon, and M. Martonosi. Deterministic error model for quantum computer simulation. *Physical Review A*, 77(5):52315, 2008.

[22] L. Childress, JM Taylor, AS Sørensen, and MD Lukin. Fault-tolerant quantum repeaters with minimal physical resources and implementations based on single-photon emitters. *Physical Review A*, 72(5):52330, 2005.

[23] JI Cirac and P. Zoller. Quantum computing with cold trapped Ions. *Phys. Rev. Lett*, 74(20):4091–4094, 1995.

[24] A. W. Cross and K. M. Svore. A QASM Toolsuite. *http://web.mit.edu/awcross/www/qasm-tools/*, 2006.

[25] A.W. Cross, D.P. DiVincenzo, and B.M. Terhal. A Comparative Code Study for Quantum Fault Tolerance. *Arxiv preprint quant-ph/0711.1556*, 2007.

[26] D. Deutsch, A. Ekert, R. Jozsa, C. Macchiavello, S. Popescu, and A. Sanpera. Quantum Privacy Amplification and the Security of Quantum Cryptography over Noisy Channels. *Physical Review Letters*, 77(13):2818–2821, 1996.

[27] S. Devadas, A. Ghosh, and K. Keutzer. *Logic synthesis*. McGraw-Hill, Inc. New York, NY, USA, 1994.

[28] W. Donath. Placement and average interconnection lengths of computer logic. *Circuits and Systems, IEEE Transactions on*, 26(4):272–277, 1979.

[29] T.G. Draper. Addition on a Quantum Computer. *Arxiv preprint quant-ph/0008033*, 2000.

[30] T.G. Draper, S.A. Kutin, E.M. Rains, and K.M. Svore. A logarithmic-depth quantum carry-lookahead adder. *Arxiv preprint quant-ph/0406142*, 2004.

[31] W. Dür, H.J. Briegel, JI Cirac, and P. Zoller. Quantum repeaters based on entanglement purification. *Physical Review A*, 59(1):169–181, 1999.

[32] M.J. Madsen et al. Planar ion trap geometry for microfabrication. *Applied Phys. B: Lasers and Optics*, 78:639 – 651, 2004.

[33] A.G. Fowler and L.C.L. Hollenberg. Scalability of Shors algorithm with a limited set of rotation gates. *Physical Review A*, 70(3):32329, 2004.

[34] MR Garey and DS Johnson. Crossing Number is NP-Complete. *SIAM Journal on Algebraic and Discrete Methods*, 4:312, 1983.

[35] D. Gottesman. Class of quantum error-correcting codes saturating the quantum Hamming bound. *Physical Review A*, 54(3):1862–1868, 1996.

[36] D. Gottesman. Stabilizer Codes and Quantum Error Correction. *Arxiv preprint quant-ph/9705052*, 1997.

[37] D. Gottesman. Fault-tolerant quantum computation with local gates. *Journal of Modern Optics*, 47(2-3):333–345, 2000.

[38] M. Grassl and T. Beth. Quantum BCH Codes. *Arxiv preprint quant-ph/9910060*, 1999.

[39] M. Grassl, W. Geiselmann, T. Beth, and A.Q. Computing. Quantum ReedSolomon Codes. *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes: 13th International Symposium, Aaecc-13, Honolulu, Hawaii, Usa, November 15-19, 1999: Proceedings*, 1999.

[40] T. Grötker. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[41] L.K. Grover. A fast quantum mechanical algorithm for database search. *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.

[42] S. Guide, M. Riebe, G.P.T. Lancaster, C. Becher, J. Eschner, H. Haeffner, F. Schmidt-Kaler, I.L. Chuang, and R. Blatt. Implementation of the Deutsch-Jozsa algorithm on an ion-trap quantum computer. *Nature*, 421(6918):48–50, 2003.

[43] RN Hall, GE Fenner, JD Kingsley, TJ Soltys, and RO Carlson. Coherent Light Emission From GaAs Junctions. *Phys. Rev. Lett.*, 9(9):366–368, 1962.

[44] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill. MIPS: A microprocessor architecture. In *Proceedings of the 15th annual workshop on Microprogramming*, pages 17–22. IEEE Press Piscataway, NJ, USA, 1982.

[45] WK Hensinger, S. Olmschenk, D. Stick, D. Hucul, M. Yeo, M. Acton, L. Deslauriers, C. Monroe, and J. Rabchuk. T-junction ion trap array for two-dimensional ion shuttling, storage, and manipulation. *Applied Physics Letters*, 88:034101, 2006.

[46] P. Houle. RngPack: High quality random numbers for Java. *Computer software distributed through the CERN Colt library, W. Hoschek, op. cit*, 2002.

[47] N. Isailovic, Y. Patel, M. Whitney, and J. Kubiatowicz. Interconnection Networks for Scalable Quantum Computers. *Computer Architecture, 2006. ISCA'06. 33rd International Symposium on*, pages 366–377, 2006.

[48] N. Isailovic, M. Whitney, Y. Patel, and J. Kubiatowicz. Running a Quantum Circuit at the Speed of Data. *ISCA'08. 35rd International Symposium on*, 2008.

[49] N. Isailovic, M. Whitney, Y. Patel, J. Kubiatowicz, D. Copsey, F.T. Chong, I.L. Chuang, and M. Oskin. Datapath and control for quantum wires. *ACM Transactions on Architecture and Code Optimization (TACO)*, 1(1):34–61, 2004.

[50] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20:359–392, 1999.

[51] D. Kielpinski, C. Monroe, and D.J. Wineland. Architecture for a large-scale ion-trap quantum computer. *Nature*, 417(6890):709–711, 2002.

[52] J. Kim, S. Pau, Z. Ma, HR McLellan, JV Gates, A. Kornblit, RE Slusher, RM Jopson, I. Kang, and M. Dinu. System design for large-scale ion trap quantum information processor. *Quantum Information and Computation*, 5(7):515–537, 2005.

[53] S. Kirkpatrick, CD Gelatt Jr, and MP Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671, 1983.

[54] A.Y. Kitaev. Quantum computations: algorithms and error correction. *Russian Mathematical Surveys*, 52(6):1191–1249, 1997.

[55] E. Knill. Quantum Computing with Very Noisy Devices. *Arxiv preprint quant-ph/0410199*, 2004.

[56] L. Kreger-Stickles and M. Oskin. Microcoded Architectures for Ion-Tap Quantum Computers. In *Proceedings of the 2008 International Symposium on Computer*

Architecture-Volume 00, pages 165–176. IEEE Computer Society Washington, DC, USA, 2008.

[57] CM Kyung, JM Widder, and DA Mlynski. Adaptive Cluster Growth (ACG): a new algorithm for circuit packing in rectilinear region. In *Proceedings of the conference on European design automation*, pages 191–195. IEEE Computer Society Press Los Alamitos, CA, USA, 1990.

[58] BS Landman and RL Russo. On a Pin Versus Block Relationship For Partitions of Logic Graphs. *Transactions on Computers*, 100(20):1469–1479, 1971.

[59] C.E. Leiserson and J.B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[60] DA Lidar, D. Bacon, and KB Whaley. Concatenating Decoherence-Free Subspaces with Quantum Error Correcting Codes. *Physical Review Letters*, 82(22):4556–4559, 1999.

[61] DA Lidar, IL Chuang, and KB Whaley. Decoherence-Free Subspaces for Quantum Computation. *Physical Review Letters*, 81(12):2594–2597, 1998.

[62] R. Lipsett, C.F. Schaefer, and C. Ussery. *Vhdl: Hardware Description and Design*. Kluwer Academic Pub, 1989.

[63] Whitney M., N. Isailovic, Y. Patel, and J. Kubiatowicz. A Fault Tolerant, Area Efficient Architecture for Shor's Factoring Algorithm. *To appear in ISCA'09. 36rd International Symposium on*, 2009.

[64] T.S. Metodi, D.D. Thaker, A.W. Cross, F.T. Chong, and I.L. Chuang. A Quantum Logic Array Microarchitecture: Scalable Quantum Data Movement and Computation. *Proc. of Intl. Symp. on Microarchitecture (MICRO)*, 2005.

[65] T.S. Metodi, D.D. Thaker, A.W. Cross, F.T. Chong, and I.L. Chuang. Scheduling physical operations in a quantum information processor. In *Proceedings of SPIE*, volume 6244, pages 210–221, 2006.

[66] B. Mohar. A Linear Time Algorithm for Embedding Graphs in an Arbitrary Surface. *SIAM Journal on Discrete Mathematics*, 12:6–26, 1999.

[67] C. Monroe, DM Meekhof, BE King, WM Itano, and DJ Wineland. Demonstration of a universal quantum logic gate. *Phys. Rev. Lett*, 75:4714–4717, 1995.

[68] J.J. Mor. The Levenberg-Marquardt algorithm: implementation and theory. *Lecture Notes in Mathematics*, 630:105–116, 1977.

[69] HC Nägerl, D. Leibfried, H. Rohde, G. Thalhammer, J. Eschner, F. Schmidt-Kaler, and R. Blatt. Laser addressing of individual ions in a linear ion trap. *Physical Review A*, 60(1):145–148, 1999.

[70] M.A. Nielsen and I.L. Chuang. *Quantum computation and quantum information*. Cambridge University Press, 2000.

[71] H. Ozaktas. Paradigms of connectivity for computer circuits and networks. *Optical Engineering*, 31(7):1563–1567, 1992.

[72] R. Ozeri, WM Itano, RB Blakestad, J. Britton, J. Chiaverini, JD Jost, C. Langer, D. Leibfried, R. Reichle, S. Seidelin, et al. Errors in trapped-ion quantum gates due to spontaneous photon scattering. *Arxiv preprint quant-ph/0611048*, 2006.

[73] R. Ozeri, C. Langer, JD Jost, B. DeMarco, A. Ben-Kish, BR Blakestad, J. Britton, J. Chiaverini, WM Itano, DB Hume, et al. Hyperfine coherence in the presence of spontaneous photon scattering. *Physical review letters*, 95(3):30403, 2005.

[74] CE Pearson, DR Leibrandt, WS Bakr, WJ Mallard, KR Brown, and IL Chuang. Experimental investigation of planar ion traps. *Physical Review A*, 73(3):32307, 2006.

[75] J. Preskill. Fault-tolerant quantum computation. *Arxiv preprint quant-ph/9712048*, 1997.

[76] B.W. Reichardt. Fault-tolerance threshold for a distance-three quantum code. *Arxiv preprint quant-ph/0509203*, 2005.

[77] B.W. Reichardt. Error-detection-based quantum fault tolerance against discrete Pauli noise. *Arxiv preprint quant-ph/0612004*, 2006.

[78] B.W. Reichardt. Fault-Tolerance Threshold for a Distance-Three Quantum Code. *Lecture Notes in Computer Science*, 4051:50, 2006.

[79] R. Reichle, D. Leibfried, RB Blakestad, J. Britton, JD Jost, E. Knill, C. Langer, R. Ozeri, S. Seidelin, and DJ Wineland. Transport dynamics of single ions in segmented microstruc-tured Paul trap arrays. *Arxiv preprint quant-ph/0606237*, 2006.

[80] M. Riebe, H. Haeffner, CF Roos, W. Haensel, J. Benhelm, GPT Lancaster, TW Koerber, C. Becher, F. Schmidt-Kaler, and DFV James. Deterministic quantum teleportation with atoms. *Nature*, 429(6993):734–737, 2004.

[81] F. Schmidt-Kaler, H. Haeffner, M. Riebe, S. Gulde, G.P.T. Lancaster, T. Deutschle, C. Becher, C.F. Roos, J. Eschner, and R. Blatt. Realization of the Cirac–Zoller controlled-NOT quantum gate. *Nature*, 422(6930):408–411, 2003.

[82] N.A. Sherwani. *Algorithms for VLSI Physical Design Automation*. Kluwer Academic Publishers Norwell, MA, USA, 1995.

[83] P. Shivakumar, M. Kistler, SW Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. *Dependable Systems and Networks, 2002. Proceedings. International Conference on*, pages 389–398, 2002.

[84] PW Shor. Algorithms for quantum computation: discrete logarithms andfactoring. *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 124–134, 1994.

[85] A. Steane. Multiple-Particle Interference And Quantum Error Correction. *Proceedings- Royal Society. Mathematical and physical sciences*, 452(1954):2551–2577, 1996.

[86] AM Steane. Error Correcting Codes in Quantum Theory. *Physical Review Letters*, 77(5):793–797, 1996.

[87] AM Steane. Active Stabilization, Quantum Computation, and Quantum State Synthesis. *Physical Review Letters*, 78(11):2252–2255, 1997.

[88] AM Steane. Quantum Reed-Muller codes. *Information Theory, IEEE Transactions on*, 45(5):1701–1703, 1999.

[89] AM Steane. Space, Time, Parallelism and Noise Requirements for Reliable Quantum Computing. *Fortschritte der Physik*, 46(4-5):443–457, 1999.

[90] A.M. Steane. Overhead and noise threshold of fault-tolerant quantum error correction. *Physical Review A*, 68(4):42322, 2003.

[91] A.M. Steane. How to build a 300 bit, 1 Gop quantum computer. *Arxiv preprint quant-ph/0412165*, 2004.

[92] D. Strooband, H. Van Marck, and J. Van Campenhout. An accurate interconnection length estimation for computer logic. In *VLSI, 1996. Proceedings., Sixth Great Lakes Symposium on*, pages 50–55, 1996.

[93] D. Stroobandt. *A Priori Wire Length Estimates for Digital Design*. Kluwer Academic Publishers, 2001.

[94] K. Svore, A. Cross, A. Aho, I. Chuang, and I. Markov. Toward a Software Architecture for Quantum Computing Design Tools. *IEEE Computer, Los Alamitos*, 2006.

[95] K.M. Svore, D.P. DiVincenzo, and B.M. Terhal. Noise Threshold for a Fault-Tolerant Two-Dimensional Lattice Architecture. *Arxiv preprint quant-ph/0604090*, 2006.

[96] K.M. Svore, B.M. Terhal, and D.P. DiVincenzo. Local fault-tolerant quantum computation. *Physical Review A*, 72(2):22317, 2005.

[97] T. Szkopek, PO Boykin, H. Fan, V. Roychowdhury, E. Yablonovitch, G. Simms, M. Gyure, and B. Fong. Threshold Error Penalty for Fault Tolerant Computation with Nearest Neighbour Communication. *Arxiv preprint quant-ph/0411111*, 2004.

[98] DD Thaker, TS Metodi, AW Cross, IL Chuang, and FT Chong. Quantum Memory Hierarchies: Efficient Designs to Match Available Parallelism in Quantum Computing. *Computer Architecture, 2006. ISCA'06. 33rd International Symposium on*, pages 378–390, 2006.

[99] D.E. Thomas and P.R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 2002.

[100] V. Vedral, A. Barenco, and A. Ekert. Quantum networks for elementary arithmetic operations. *Physical Review A*, 54(1):147–153, 1996.

[101] M. Whitney, N. Isailovic, Y. Patel, and J. Kubiatowicz. Automated Generation of Layout and Control for Quantum Circuits. In *Proc. of ACM Intl. Conf. on Computing Frontiers*, 2007.

[102] T. Williams, C. Kelley, et al. GNUplot: an interactive plotting program. *Manual, version*, 3, 1998.

[103] DJ Wineland, C. Monroe, WM Itano, D. Leibfried, BE King, and DM Meekhof. Experimental issues in coherent quantum-state manipulation of trapped atomic ions. *Arxiv preprint quant-ph/9710025*, 1997.

[104] W.K. Wootters and W.H. Zurek. A single quantum cannot be cloned. *Nature*, 299(5886):802–803, 1982.

[105] C. Zalka. Simulating quantum systems on a quantum computer. *Proceedings: Mathematical, Physical and Engineering Sciences*, 454(1969):313–322, 1998.

[106] B. Zeng, A. Cross, and I.L. Chuang. Transversality versus Universality for Additive Quantum Codes. *eprint arXiv: 0706.1382*, 2007.