Quantum Rotations: A Case Study in Static and Dynamic Machine-Code Generation for Quantum Computers

Daniel Kudrow[†], Kenneth Bier[†], Zhaoxia Deng[†], Diana Franklin[†], Yu Tomita[*],
Kenneth R. Brown[*], and Frederic T. Chong[†]

[†]University of California at Santa Barbara, Santa Barbara, CA 93106, USA
[*]Georgia Institute of Technology, Atlanta, GA 30332, USA

dkudrow@cs.ucsb.edu, kenbier@gmail.com, zhaoxia@cs.ucsb.edu,
franklin@cs.ucsb.edu, yu.t@gatech.edu, ken.brown@chemistry.gatech.edu,
chong@cs.ucsb.edu

ABSTRACT

Work in quantum computer architecture has focused on communication, layout and fault tolerance, largely driven by Shor's factorization algorithm. For the first time, we study a larger range of benchmarks and find that another critical issue is the generation of code sequences for quantum rotation operations. Specifically, quantum algorithms require arbitrary rotation angles, while quantum technologies and error correction codes provide only for discrete angles and operators. A sequence of quantum machine instructions must be generated to approximate the arbitrary rotation to the required precision.

While previous work has focused exclusively on static compilation, we find that some applications require dynamic code generation and explore the advantages and disadvantages of static and dynamic approaches. We find that static code generation can, in some cases, lead to a terabyte of machine code to support required rotations. We also find that some rotation angles are unknown until run time, requiring dynamic code generation. Dynamic code generation, however, exhibits significant trade-offs in terms of time overhead versus code size. Furthermore, dynamic code generation will be performed on classical (non-quantum) computing resources, which may or may not have a clock speed advantage over the target quantum technology. For example, operations on trapped ions run at kilohertz speeds, but superconducting qubits run at gigahertz speeds.

We introduce a new method for compiling arbitrary rotations dynamically, designed to minimize compilation time. The new method reduces compilation time by up to five orders of magnitude while increasing code size by one order of magnitude.

We explore the design space formed by these trade-offs of dynamic versus static code generation, code quality, and quantum technology. We introduce several techniques to provide smoother trade-offs for dynamic code generation and evaluate the viability of options in the design space.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Compilers—*quantum computers*

Keywords

Compilers, Programming languages, Quantum computers

1. INTRODUCTION

Since the landmark *Nature* paper in 2000 [26] describing a technology for scalable quantum computation (part of the work for which Wineland received this year's Nobel prize in physics), interest in practical quantum computation has grown significantly. Recent announcements by commercial effort D-Wave of a 128-quantum-bit (*qubit*) adiabatic system [16], although controversial in terms of its quantum properties, illustrate the engineering progress that has been made.

By exploiting features of quantum mechanics to manipulate information, quantum computers offer the prospect of exponential speed up over classical computers on applications such as Shor's algorithm for the factorization of the product of two primes [32]. Previous work in quantum computer architecture has largely used Shor's algorithm as a driving application [34, 21, 7, 24, 25]. Due to the relative simplicity of Shor's algorithm, these studies have focused on error correction and communication in the context of quantum addition.

The work on Shor's algorithm was supported by NSF REU funds. This work was supported by Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center contract D11PC20167. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.

ISCA '13 Tel Avv, Israel
Copyright 2013 ACM 978-1-4503-2079-5/13/06 ...$15.00.

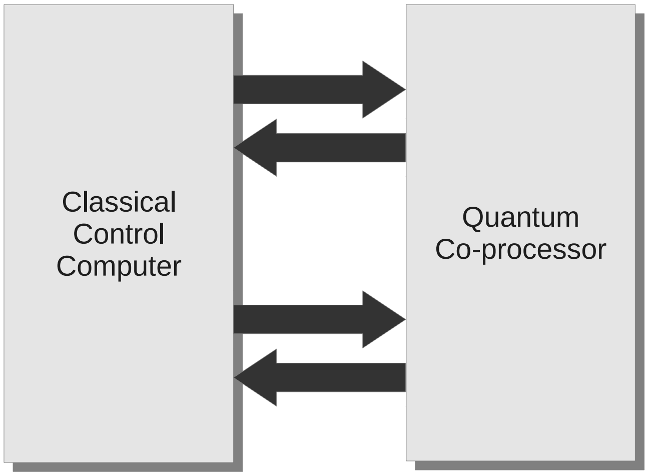

Figure 1: Applications are run on classical control hardware that sequences quantum computations on a quantum co-processor.

Recent efforts in the quantum computing community have sought to define a much broader set of quantum benchmark algorithms and to create a tool chain to compile quantum programs and synthesize quantum architectures. We examined these benchmarks and found that an important challenge lies in generating sequences of quantum operators (quantum assembly) to implement a "quantum rotation" (also known as a quantum phase shift operator) at the algorithm level.

1.1 Architecture and Execution Model

Although the fabrication and control of quantum devices varies with technology, we present a common framework with an abstract architecture, execution model, and associated software tool chain. We consider a quantum co-processor tethered to a classical control computer. Figure 1 illustrates this arrangement; applications are run on the classical control computer which executes co-processor instructions that control operations on the quantum co-processor and collect results through measurement. The quantum co-processor relies on the control computer for a number of functions including scheduling and error detection and correction.

Our applications are written in Scaffold, a high-level, C-like programming language with quantum extensions. The Scaffold compiler outputs an assembly program which is a mixture of x86 and quantum assembly instructions (QASM, a target-independent assembly code). We find that a difficult problem for the Scaffold compiler is code generation for arbitrary rotations. In the next section, we give some basic background in quantum computation to help motivate this difficulty.

1.2 Background

The fundamental unit of quantum computation is the quantum bit or *qubit.* Whereas a classical bit is confined to existing in either a high or a low state, a qubit can exist in both states simultaneously. This phenomenon is known as *superposition* and a qubit exists in superposition until it is measured. Upon measurement a qubit "collapses" into a classical bit, taking and keeping a value of either high or low. A qubit in superposition is described by the probability with which it will be found in each classical state once measured. A qubit's state, $|s\rangle$, is expressed by the equation,

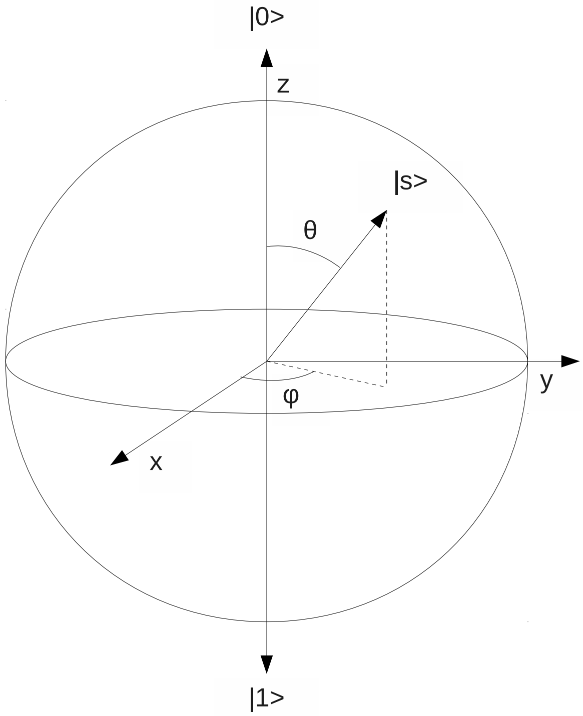

Figure 2: Bloch Sphere. The Bloch Sphere is a useful tool in visualizing the state space of a qubit. Every point on the surface of the sphere represents a unique qubit state with the poles (along the Z-axis) correspond to the classical high and low states. Points lying between the poles represent a superposition of the classical states. ϕ is a qubits phase angle which will be discussed in the next section.

$$|s\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \qquad (1)$$

where $|\alpha|^2$ is the probability of measuring the qubit in state $|0\rangle$, the classical low state, and $|\beta|^2$ is the probability of measuring the qubit in state $|1\rangle$.

The Bloch Sphere, shown in Figure 2, is a useful visualization of a qubit's state space[29]; the points on the surface of the Bloch Sphere represent the different states a qubit can occupy with the classical high and low states located at the poles. Proximity to the poles indicates the probability with which the qubit will collapse to that pole on measurement - a state lying on the equator is equally likely to be high or low on measurement. Longitude on the Bloch Sphere represents the phase between the eigenstates of a qubit. This will be discussed in more detail later on.

The Bloch Sphere is especially useful when considering transitions between qubit states. As different points on the Bloch Sphere correspond to different states, transitions can be visualized as rotations about the axes of the sphere. These rotations are abstracted as quantum logic gates that act on a single qubit. An accessible example is the quantum NOT gate which performs a 180 degree rotation about the

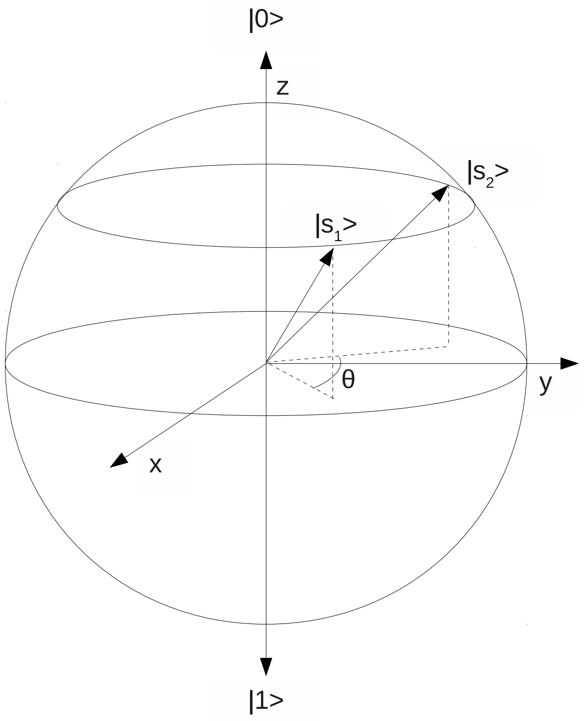

Figure 3: Bloch Sphere depicting a rotation. This figure shows the application of a phase-shift gate to a qubit. The initial state s_1 rotates θ radians about the Z-axis. The resultant state, s_2, maintains the same superposition but has a phase shifted by θ.

X- axis. When applied to a classical bit it simply transitions from one pole to another. When applied to a qubit, the NOT gate inverts the probabilities in the superposition.

The quantum gate (or, more precisely, set of gates) on which we focus our case study is the phase-shift gate. In the context of the Bloch Sphere model, this gate is represented by a rotation about the Z-axis as detailed by Figure 3. This gate changes the phase of a qubit by a specified angle θ. As evidenced by the Bloch Sphere, phase does not directly effect the outcome of a measurement however it is very important in determining how qubits interact with one another. This feature of qubits can be exploited in designing quantum algorithms and the methods of doing so will expounded upon in the next section. For the purposes of this paper, we refer to phase shift gates as rotations because, unlike other standard gates, they are parameterized by the angle of rotation (although technically, any single qubit operation can be represented as a Bloch Sphere rotation).

1.3 Contributions and Roadmap

The remainder of this paper provides a narrative that follows the contributions of our work:

- Section 2.1 examines, for the first time, a range of quantum benchmarks, motivating the problem of quantum rotations.
- Section 3 describes the design space formed by compilation methods, quantum technologies, and benchmark requirements.
- Section 4.2 describes our caching and parallelism optimizations to the baseline Solovay-Kitaev implementation, which result in nearly a 10X improvement.
- Section 4.4 introduces our dynamic compilation method for arbitrary quantum rotations.
- Section 5 describes our empirical and analytical evaluation of the design space.

2. MOTIVATION

Arbitrary rotations pose a challenge to quantum computer architects because they are parameterized by an angle in the range $[0, 2\pi]$ and thus form a continuous set. Because only discrete sets of gates can be implemented fault-tolerantly, arbitrary rotations must be approximated by sequences of discrete gates [27]. The Bloch Sphere helps to illustrate this idea; an arbitrary rotation can be thought of as a transition between two points separated by angle θ about the Z-axis. The set of fault-tolerant gates correspond to a series of 'allowed moves' on the Bloch Sphere. Approximating a rotation involves using a sequence of allowed moves to travel from one of these points to the other. The compilation of an arbitrary rotation is the process of generating a sequence of gates from a given set that provides a good approximation of the original rotation. We will define the metrics by which we evaluate approximations in the next section.

In Figure 4, we provide a short example illustrating the compilation of a rotation. The Scaffold code to perform a simple controlled rotation about an arbitrary angle *theta*. When this code is compiled into QASM, the R_z function (arbitrary quantum rotation) is replaced by a series of machine instructions from the set $\left\{H, T, T^{\dagger}\right\}$ (which are universal quantum operations, Hadamard and discrete rotations, assumed to be supported by our machine).

2.1 Benchmark Applications

Arbitrary rotations play a crucial role in a number of fundamental constructs that recur frequently in quantum algorithm design. Our suite of benchmark algorithms utilize arbitrary rotations in multiple contexts.

One common algorithm in our benchmarks is the quantum random walk, which is a graph traversal technique that is used for path-finding algorithms[17]. In a quantum random walk, the direction and length of each step in the walk are determined by the state of a qubit. When implemented without rotations, quantum walks are very similar to classical walks. At each step, all moves have equal probability and equal length. By applying a rotation, these probabilities can be skewed allowing for, at lower probabilities, longer steps than are possible in a classical walk.

Arbitrary rotations are also used in quantum phase estimation and the quantum Fourier Transform[27]. It was mentioned earlier that the phase between the eigenstates does not effect measurements but is nonetheless important to quantum computing. This is because it is often useful to encode information in the phase differences of qubits while they are in superposition. Quantum phase estimation is a method that is used to decode information encoded in qubit phases.

We examined eight quantum benchmarks covering just about all "practical" quantum algorithms. We describe each algorithm briefly below and summarize our analysis of these

Scaffold

```
module ControlledRz(
qreg target[1],
qreg control[1],
double theta)
{
        Rz(target, theta / 2.0);
        CNOT(target, control);
        Rz(target, −1.0 ∗ theta / 2.0);
        CNOT(target, control);
}
```

⇒

QASM

```
module ControlledRz
begin
qubit target
qubit control
double theta
H target
T target
T∗ target
T target
...
CNOT target control
end
```

Figure 4: Scaffold compilation example

benchmarks in Table 1. The number of Lines of Scaffold code is provided for each benchmark to give a sense of coding complexity. The resource requirements of each algorithm depend on the problem size. For each algorithm we selected a problem size large enough that the quantum algorithm may show speedup over classical algorithms. In the final column, we specify "Required Precision". This is a measure of how precisely rotations compiled for each algorithm must conform to those specified in the source code. This calculation is made by considering both circuit depth (the number of consecutive rotations that occur) and width (the number of qubits which must be measured following the rotations).

Binary Welded Tree A binary welded tree is the graph formed by joining two perfect binary trees at the leaves. Given an entry node and an exit node, The Binary Welded Tree Algorithm uses a quantum random walk to find a path between the two. The quantum random walk finds the exit node exponentially faster than a classical random walk[8].

Boolean Formula Two player games can be modeled as NAND trees or boolean formulas. The Boolean Formula Algorithm can determine a winner in a two player game by performing a quantum random walk on a NAND tree[1].

Ground State Estimation This algorithm determines the ground state energy of a molecule given a ground state wave function. This is accomplished using quantum phase estimation[33].

Linear Systems This algorithm makes use of the quantum Fourier Transform to solve systems of linear equations[15].

Shortest Vector The shortest vector problem is an NP-Hard problem that lies that the heart of some lattice-based cryptosystems. The Shortest Vector Algorithm makes use of the quantum Fourier Transform to solve this problem[30].

Class Number Computes the class number of a real quadratic number field in polynomial time [14]. This problem is related to elliptic-curve cryptography, which is an important alternative to the product-of-two-primes approach currently used in public-key cryptography.

Shor's Algorithm This algorithm is probably the most famous result of quantum computing and has important implications for public-key cryptography. This algorithm factors integers in polynomial time using a phase estimation algorithm[32].

Grover's Algorithm This algorithm is probably the second most famous result in quantum computing [12]. Often referred to as "quantum search," Grover's algorithm actually inverts an arbitrary function by searching n input combinations for an output value in $\sqrt{n}$ time. Grover's algorithm does not use rotation operations.

We find that seven of the eight benchmarks feature arbitrary rotations prominently. Two of the eight (Linear Systems and Ground State Estimation) feature a very large number of distinct rotations, posing potential code size challenges for static code generation. Two of eight (Linear Systems and Shortest Vector) feature rotation angles calculated at runtime, requiring dynamic code generation.

While we examine these benchmarks to motivate dynamic versus static compilation, none of these benchmarks can be simulated at scale. If quantum computations could be simulated in polynomial time, we wouldn't need quantum computers! In our subsequent analysis, we will assume random rotation angles. We will give error bars in our results, but we found that variations due to optimization parameters dominated any variations due to the value of rotation angles.

2.2 Existing Quantum Technologies

There are a number of candidate technologies for implementing a quantum computer, each with its own characteristics that influence our approach to arbitrary rotations. In order to evaluate the viability of dynamic code generation for a particular technology, we examine the "logical clock cycle" of each technology. This is the time it takes to perform a fault-tolerant discrete rotation. Quantum computers utilize extremely powerful and expensive error correction codes, which are applied recursively.

Technology	Experimental	Calculation
Ion Trap	10 μs [5]	1.0 μ s
Neutral Atom	31 μs [28]	0.915 μ s
Quantum Dot	2.6 ns [3]	1 ns
Superconductor	20 ns [23]	1 ns
Photons	15 ps [31]	1 ns

Table 2: Times for physical T gates in quantum information systems. Experimental refers to current examples of implemented high fidelity gates. Calculation indicates the predicted gate times. This test set underlies the error correction times in Table 3. The calculation times do not represent fundamental limits, e.g., ion trap single gates of 50ps duration have been demonstrated[6].

Table 2 provides the time for $\pi/4$ rotations on the physical layer (the equivalent of a T gate). Ref. [22] gives an overview

Benchmark	Lines of Code	Total Rotations	Distinct Rotations	Rot. Unknown at Compile Time	Required Precision
Binary Welded Tree	608	2080	5	0	6.9×10^{-4}
Boolean Formula	479	828	46	0	2.1×10^{-9}
Ground State Estimate	554	2.6×10^{14}	4095	0	5.3×10^{-9}
Linear Systems	1741	2.2×10^{6}	1794	66	4.1×10^{-8}
Shortest Vector	539	2450	50	<50	3.7×10^{-14}
Class Number	400	48	48	0	1.6×10^{-14}
n-bit Shor's Algorithm	140	n	n	0	2^{-n}
Grover's Algorithm	237	0	0	0	0

Table 1: A brief overview of benchmark quantum algorithms.

pointing to both successes and challenges. We examined the literature to find times for the fastest reported gates and the highest fidelity gates. These timings were used to determine the time overhead for error correction using the Bacon-Shor code[2] and Knill ancilla scheme[20] shown in Table 3. For simplicity, we use level 2 error correction in our logical cycle time for each technology.

2.3 Previous Work

The problem of arbitrary rotation compilation has been studied primarily in the context of Shor's Prime Factorization Algorithm. Shor's Algorithm reduces prime factorization to an order finding problem making use of a Quantum Fourier Transform[32]. The rotations performed by a Quantum Fourier Transform are very predictable, falling into the set

$$\left\{R_z(\frac{\pi}{2^1}), R_z(\frac{\pi}{2^2}), R_z(\frac{\pi}{2^3}), ..., R_z(\frac{\pi}{2^k})\right\}$$

where k is the size of the input register. Because Shor's Algorithm presents such a manageable set of rotations, it has been tacitly assumed that these rotations would be precalculated and used when needed. In the context of this narrow application scope, there was little motivation for rapid decomposition of rotations, particularly at runtime. While there are a number of techniques in use for compiling arbitrary rotations[10][4], their suitability for runtime code generation has not been examined.

3. DESIGN SPACE

In this section we introduce a design space that is bounded by the algorithm requirements as well as computational resource considerations. The purpose of this design space is to define metrics that can be used to determine an appropriate compilation method for a given application and technology.

	Level 1	Level 2
Neutral Atom (T)	4.46e-04	6.68e-03
Neutral Atom (EC)	9.45e-05	5.57e-03
Ion Trap (T)	2.08e-02	9.00e-01
Ion Trap (EC)	6.47e-03	7.95e-01
Photons (T)	6.12e-07	1.66e-05
Photons (EC)	2.25e-07	1.44e-05
Quantum Dot (T)	3.08e-05	5.60e-04
Quantum Dot (EC)	7.66e-06	4.75e-04
Superconductor (T)	2.80e-06	1.33e-04
Superconductor (EC)	1.06e-06	1.21e-04

Table 3: Times for implementing a fault-tolerant T gate using Bacon-Shor code for one and two levels of error correction. Time is in seconds.

3.1 Executable Size

The amount of assembly code generated is another important consideration in evaluating compilation. Generally the number of gates needed for an approximation is proportional to the required accuracy - higher precision sequences tend to be longer. Some algorithms require tens of thousands of high precision rotations which can translate into terabytes of generated code. This compilation is only the first step in a long tool chain required in preparing a quantum application for execution. This large amount of code would need to pass through the remaining steps in the tool chain which may not be able to manage such input of such a size. The generated sequence length also has a number of consequences for the execution of the algorithm on a quantum processor. Each gate carries an inherent error frequency. Having to execute more gates per rotation increase the likelihood that the rotation will fail. Furthermore, in an algorithm that bottlenecks at rotation execution, changes in sequence length can dramatically influence performance by reducing the number of gates necessary to perform each rotation.

3.2 Compilation Time

Finally, we consider the time needed to compile arbitrary rotations. Similar to sequence length, there is generally a trade-off between accuracy and compilation time with higher precision sequences taking longer to generate. Furthermore, optimizations for shorter sequences also tend to lead to longer compilation time. While compilation time is not a major concern in the static approach, it becomes a limiting factor when we consider dynamic compilation schemes and on-the-fly code generation. In the event that we need to generate code between quantum operations, compilation time determines the magnitude of the interruption to quantum code execution.

4. STATIC VS. DYNAMIC COMPILATION

We divide general approaches to the compilation of arbitrary rotations into static schemes and dynamic schemes. During static compilation, approximating sequences are generated for each rotation along with the rest of the quantum assembly code. The major advantage in this scheme is that the time overhead of sequence generation is paid for once, at compile time, and then does not factor into the algorithm's performance. Because the time cost of static compilation can be largely ignored, there is no need to compromise accuracy or gate count optimization.

Note that static compilation in quantum computation generally assumes that you have the input parameters for a program, but that you can not do any quantum computation in the compilation process. In other words, the situation is essentially trace-based compilation with constant propagation using classical computing resources.

There are, however, circumstances in which static compilation may not be an option. One such case is when rotation angles are not known at compile time as with the Linear Systems and Shortest Vector Algorithms. If the angle of a rotation is dependent on the measurement of a quantum register, its value cannot be known until the algorithm executes and compilation of the rotation would have to occur at run time. In this case, execution must halt pending the compilation of the rotation. Unlike the static scheme, the time overhead of rotation compilation would be incurred at every execution. Fast compilation becomes very important here because holding up execution of the quantum algorithm will affect algorithm performance. Another instance in which dynamic compilation may be preferable is an algorithm with a large number of distinct rotations. Statically compiling quantum code that contains a large number of distinct rotations can lead to vast quantities of assembly code. Although we have chosen moderate problem sizes, the majority of the benchmarks are easily scalable. Consider the Ground State Estimation Algorithm for Fe_2S_2. This algorithm requires on the order of 10^{14} rotations. If each of these is decomposed into a sequence of 10^5 gates, the generated assembly code would contain 10^{19} logical gates. This code would then have to be handled by the rest of the tool chain for circuit mapping and error correction and it is very possible that this code would be simply too large for the classical control computer. If an algorithm promises to generate an unmanageable amount of quantum assembly code due to the presence of arbitrary rotations, it may be necessary to compile the rotations dynamically.

4.1 The Solovay-Kitaev Algorithm

The Solovay-Kitaev Theorem[18] is a result in quantum computing which states that any single qubit gate can be approximated to a precision ϵ by a sequence of $\Theta(log^c(\frac{1}{\epsilon}))$ gates from a universal set. The Solovay-Kitaev Algorithm, as implemented by Chris Dawson and Michael Nielsen[9], is widely utilized by the quantum computing community and illustrates how the different dimensions of our design space interact. We consider Solovay-Kitaev in our study primarily as a method of static compilation but also explore the extent to which it can be used to generate code dynamically.

The Solovay-Kitaev algorithm requires two inputs. The first input is a finite set of quantum gates (expressed as two-by-two unitary matrices) that form a basis. In our case, the basis will be the set of fault-tolerant gates that comprise the instruction set of the target quantum processor. The second input is a single-qubit gate that will be decomposed into the gates supported by the target architecture, for our purposes an arbitrary rotation.

Solovay-Kitaev decomposes the input gate by recursively factoring it to a predetermined depth. In the leaves of the recursion tree, each factor is approximated by a short sub-sequence (generally about 20 gates long) selected from a database. These subsequences are then concatenated to form the final approximation of the rotation.

The Solovay-Kitaev algorithm is parameterized by the maximum length of the sub-sequences in the database as well as the recursion depth. Increasing the target sequence length increases the accuracy of the approximation, but also increases compilation time. Increasing recursion depth increases accuracy as well, but also increase both compilation time and resulting sequence length.

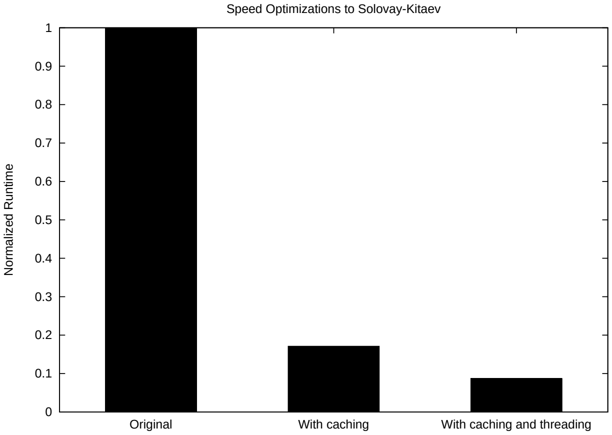

Figure 5: Optimizations to Solovay-Kitaev. The leftmost bar depicts a normalized average runtime of Solovay-Kitaev. The center bar shows the effects of capitalizing on processor caching. The rightmost bar shows the effects of parallelization.

4.2 Optimizations to Solovay-Kitaev

While the Dawson and Nielsen implementation of Solovay-Kitaev is commonly used to statically compile arbitrary rotations, it runs too slowly to be considered practicable as a runtime code generator. To use it in our study, we had to make a number of modifications to the original code to optimize runtime. We first employed serial code optimizations, and then we parallelized the code.

The majority of execution time is spent in the leaves of the recursion tree where Solovay-Kitaev performs an iterative search through a list of sub-sequences used to approximate the rotation's factors. We determined that the best approaches to reducing run time were to attempt to parallelize the recursion tree and to exploit processor caching to expedite the iterative search. We modified the implementation to store base-case sequences in contiguous memory which allowed us to maximize the advantage of processor caching. While making this change we also discovered that the limited precision of the database made a large number of sub-sequences redundant. In many cases, the difference between consecutive subsequences was less than the floating point precision in the database. By checking for and eliminating these sub-sequences, we decreased the number of sub-sequences that needed to be searched for in the database substantially. Together these modifications brought nearly a 5X decrease in compilation time.

By executing independent branches of the recursion tree in separate threads, we were able to further improve runtime by a factor of two. All together, we were able to decrease Solovay-Kitaev's compilation time by nearly an order of magnitude, illustrated by Figure 5.

4.3 Single Qubit Circuit Toolkit

We also include in our study the Single Qubit Circuit Toolkit (SQCT), recently released by the University of Waterloo as another static compilation method[19]. SQCT is a novel technique that is able to minimize the number of T gates in its approximations. This is important because in many technologies, T gates dominate the time cost of execution.

SQCT is built around a key result of [19] that describes a method of exactly synthesizing a particular class of single qubit gates out of H and T gates. For this class of gates, Maslov et al. have found a way to generate sequences that exactly implement the input gate and that have optimal T gate counts. The results of Solovay-Kitaev compilations fall into the class of gates on which this method can operate and so this method can be used to re-synthesize Solovay-Kitaev results to produce length-optimal sequences. As such, SQCT can be considered a drop-in replacement for Solovay-Kitaev.

SQCT decomposes a single qubit gate in two steps. It first utilizes its own (faster) implementation of Solovay-Kitaev to generate an approximation for the input gate. This approximation is not length-optimal but, being a Solovay-Kitaev result, can be exactly synthesized using the T gate optimal technique. To generate the T gate optimal approximation, the Solovay-Kitaev result must be resynthesized using the technique described above. The extra step adds an element of linear linear complexity to the compilation but leads to machine code that executes more quickly due to the minimization of T gates. As with the Solovay-Kitaev Algorithm, we consider SQCT to be primarily a static compilation technique but will evaluate its fitness to generate code dynamically.

4.4 Library Construction

As an alternative to the aforementioned techniques we propose a new method for approximating rotations that focuses on minimizing compilation time. In this technique, a set of rotations is pre-compiled (using a static compilation technique such as Solovay-Kitaev) and stored as a library. Arbitrary rotations are then rapidly assembled by concatenating rotations from this library. We are essentially trading storage (and a less optimized result) for execution time. The library rotations follow the form,

$$\left\{\frac{(n-1)\pi}{n^1}, \frac{(n-2)\pi}{n^1}, ..., \frac{\pi}{n^1}, ..., \frac{\pi}{n^k}\right\}. \tag{2}$$

An arbitrary angle can be built by concatenating the elements in the set using the construction,

$$\theta = \sum_{k=1}^{k=k_{max}} C_k \frac{\pi}{n^k}. \tag{3}$$

where C is the k^{th} digit in the base n representation of θ.

As an example consider a binary construction where $n = 2$. The library would consist of the gates

$$\left\{\frac{\pi}{2^1}, \frac{\pi}{2^2}, \frac{\pi}{2^3}, ..., \frac{\pi}{2^k}\right\}. \tag{4}$$

Now constructing an arbitrary rotation is as simple as rewriting it as a binary fraction. For instance,

$$\theta = \frac{7\pi}{9} = 0.11\overline{000111}. \tag{5}$$

By Adding the corresponding rotations in our library set,

$$\theta \approx \frac{\pi}{2^1} + \frac{\pi}{2^2} + \frac{\pi}{2^6} + \frac{\pi}{2^7} + \frac{\pi}{2^8} + ... \tag{6}$$

we are guaranteed an approximation to within $\frac{\pi}{2^k}$ of the original angle.

This technique allows for faster compilation because there is no need to generate new sequences. Determining which library sequences to concatenate takes $|\log_k(\epsilon)|(n-1)$ iterations to achieve an accuracy of ϵ or better. However, the cost for this speed comes in the form of increased sequence length for the approximation. Because each library rotation has the same sequence length as a statically compiled rotation, a rotation created in this manner has a worst case sequence length that is $|\log_k(\alpha)|$ times greater than its statically compiled counterpart. This drawback can be mitigated by increasing the size of the library. By using a large value for n we achieve higher precision at each iteration of the construction. A library built around a base of $n = 100$ can achieve an accuracy of $\frac{\pi}{100}$ in the first term of the sum whereas a library built around $n = 10$ will achieve this after only after the second term resulting in a sequence that is twice as long in the worst case. However the first library will be $\frac{(100-1)k}{(10-1)k} = 11$ times larger then the second.

This method is also in a position to take advantage of optimized static compilation techniques that are too slow to be used dynamically. Consider SQCT which sacrifices compilation time to generate length-optimal sequences. Although it may not be fast enough to generate code dynamically, we can still use it to create a library. A library composed of compilations that surpass the precision requirement of the target can compensate for the loss of precision incurred by dynamic code generation. In this way, optimized techniques can still be useful for dynamic code generation.

5. RESULTS AND ANALYSIS

This section presents the results of our analysis of the compilation techniques that we introduced in the previous section. We use the candidate technologies and our suite of benchmark algorithms to define realistic bounds within our design space for dynamic rotation compilation. We then evaluate our techniques within the design space and determine their fitness for use in a quantum computing tool chain.

5.1 Methodology

As discussed in Section 2.1, due to fundamental limitations in simulating quantum computers at scale, we do not use the rotation angles of our actual benchmarks in our study. Instead, we use a random distribution of rotation angles and provide error bars to illustrate the dependence of our results on specific angles. We also interpret our results in the context of the precision required for each benchmark, as well as the number of rotation angles not known statically (where applicable).

All of the tests were conducted on a 3.3GHz Intel i7 processor with 32Gb of memory, running a Linux 3.2 kernel. We ran each method over a wide range of parameters to find the best results for each one. For each parameterization, the compilation was performed on the same set of evenly distributed input angles. In the following figures, each point represents one parameterization, averaged across the set of input angles. Error bars are used to display the standard deviation of a value over the set of input angles. We omit error bars in cases where they would be drawn too finely in figure to be of practical use.

For all methods we used the universal set of quantum gates consisting of $\{H, T, T^{\dagger}, X, Y, Z\}$ as our basis because it is implemented fault tolerantly by all of the considered technologies. In Solovay-Kitaev we varied the recursion depth of the algorithm as well as the base-case sub-sequence length to find optimal parameters. In implementing the library construction method, we used SQCT to generate the library due to the length-optimal sequences it produces. We used multiple rotation libraries varying in size from a few tens to a few thousands of sequences by varying the base n of the construction (recall the use of base n in Section 4.4).

Because Solovay-Kitaev and SQCT decompose all rotations in the same way their is very little correlation between input angle and performance; sequence length and compilation time do not vary much for different rotations. In the library construction, because the number of sequences to be concatenated is determined by input angle, there is more fluctuation across inputs. However, due to the unpredictability of the input, these fluctuations cannot be capitalized upon to improve performance.

The accuracies of the Solovay-Kitaev algorithm and the library construction were limited by the built-in floating-point data type. Evaluating sequences in both of these methods involves many matrix operations and the loss of precision propagates through the results. We were able to achieve an accuracy of approximately 10^{-11} before the accuracy plateaued. Based on our empirical data we extrapolate the performance of both methods for benchmarks with higher accuracy requirements. An attempt was made to extend these methods using the CLN arbitrary precision software library[13], however the time overhead incurred by CLN was unmanageable. Slowdowns on the order of 1,000X prevented the collection of meaningful data. SQCT achieves high levels of precision using the GNU Multiple Precision Arithmetic Library[11] and we intend to use this in future development of this work.

5.2 Relating Compilation Time and Accuracy

The scale of the time axis of our design space is defined by the speeds of the candidate quantum technologies. Our goal is to be able to compile an arbitrary rotation on the classical control computer with minimal effect on the execution of the algorithm on the quantum processor. We can think of quantum stalls in terms of the logical cycle time of the quantum processor, given by the error-corrected T gate speed of the target quantum technology (which can be found in Table 3). We define logical cycles thusly because the T gates in all of our technology descriptions, when implemented fault tolerantly, incur the highest runtime cost and dominate algorithm execution time.

In Figure 6 we compare the accuracy of a generated sequence and the time taken to generate it. The horizontal lines in the plot provide error-corrected gate execution times for a few technologies as a reference. We see that that the library construction offers up to five orders of magnitude speedup over the static methods for most of our benchmarks.

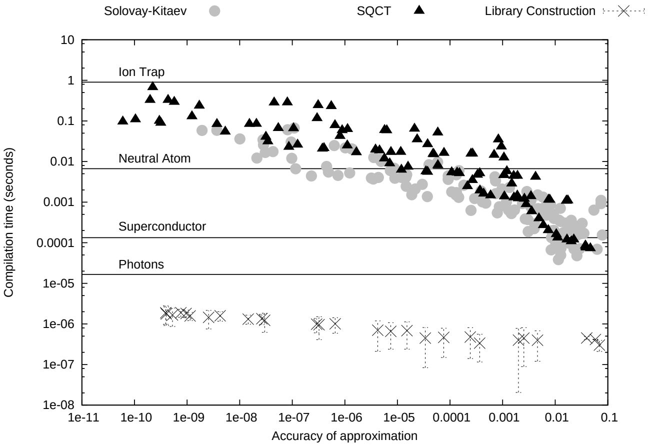

Figure 6: Accuracy vs. Compilation Time. This plot demonstrates the time required to perform rotation compilations of varying precision. The accuracies of the resultant sequences were measured by calculating the trace distance between the compiled rotation and the original. The T gate execution times of different quantum technologies are included for reference.

Our results indicate that both Solovay-Kitaev and SQCT are fast enough to perform dynamic compilation for all of the benchmarks in the test range, but only for the ion trap technology. Shortest Vector and Class Number lie outside of this range, but it is easy to extrapolate from the data that dynamic compilation of these algorithms lies outside the capabilities of Solovay-Kitaev and SQCT. We arrive at this interpretation by referring to the required precision for each benchmark in Table 1, finding this precision on the X axis of Figure 6, finding the corresponding Y-axis time on each curve, and then referring to the horizontal reference lines for each technology's logical cycle time.

Using data gathered with the library construction technique, we see that it provides sufficient speed to perform dynamic compilation for all of the algorithms in the test range and that it does so well within the gate speeds for all technologies too. Extrapolating these results to the accuracies required by Shortest Vector and Class Number, we find that the library construction method can be used dynamically with our full suite of benchmarks.

5.3 Sequence Length

While the library construction of arbitrary rotations yields much faster compilations than Solovay-Kitaev, it does produce longer sequences of gates. Figure 7 shows the relationship between accuracy and sequence length for the data sets used in Figure 6. The decrease in runtime is accompanied by, in some cases, a one order of magnitude increase in sequence length. This is significant especially when considering the error-correction overhead accompanying each gate.

In our plot, We measure sequence length in terms of T-gate counts. The striations in the Solovay-Kitaev and SQCT results are manifestations of the parameterization of these techniques. The strata occur at each level of recursion for these compilation methods.

All of the compilation methods function by concatenating shorter, pre-generated sequences of gates to form their approximations. The static methods represent a very large database, containing on the order of 10^8 very short sequences (20 gates). The library construction method uses longer se-

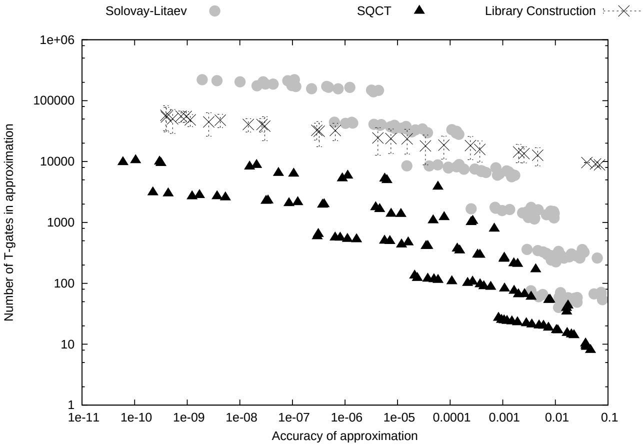

Figure 7: Accuracy vs. Sequence Length. In this figure we compare the lengths of sequences generated using different methods. We use the number of T-gates in each approximation as a metric as they incur the highest runtime cost in most technologies.

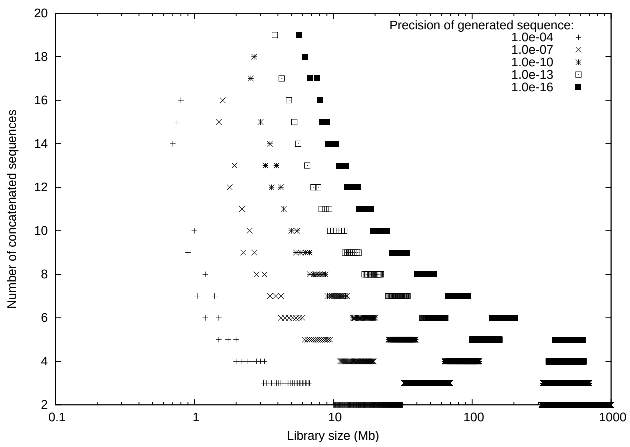

Figure 8: Library Size vs. Sequence Length. We can smooth the trade-off between sequence length and compilation time by increasing database size in the library construction method. Each curve demonstrates the relationship between database size and sequence length for a given accuracy. We used an analytical model to generate this data.

quences but fewer of them, in the above case less than 30.

As mentioned in the previous section, we can decrease the sequence length required by the library constructions by increasing the database size. Figure 8 shows the effects of increasing the size of the database by increasing the repertoire of subsequences to build from. This is done by increasing the base n of the construction. Recall that this effectively gives us more fractional rotations so that we can achieve an arbitrary rotation by concatenating fewer subsequences (see Section 4.4. The overall size of a database built around base n will be a factor of $(n-1)$ larger than the binary construction. We use analytical data to predict the results of different configurations of the library construction method. The plot assumes an ideal library of perfect sequences, however it accurately displays the inverse relationship between sequence length and database size. By using large databases ($>$ 100Mb) we can achieve high accuracies in less than one order of magnitude more gates than a static compilation. Since we can keep even the larger databases in memory, compilation time across database sizes should not vary significantly.

5.4 Dynamic Compilation

For an application in which static compilation is not an option such as Linear Systems or Shortest Vector, we must consider the feasibility of using our compilation techniques dynamically. Our criteria is that dynamic compilation should only cause a small number of stall cycles in the quantum processor, once again measured as the time for the error-corrected T operator.

To determine the fitness of a method of dynamic compilation we must take into account compilation time as well as the length of the sequence generated. The total time incurred by a rotation can be expressed as,

$$T_{rotation} = t_{compile} + l \times t_{gate}, \tag{7}$$

where $t_{compile}$ is the compilation time of the gate, l is the length of the generated sequence and t_{gate} is the technology-specific gate time. When evaluating compilation strategies for a particular technology, we can use this equation determine which of several methods has the least impact on the running of the application.

From this simple equation, we see that the choice of technology and the precision to which we are compiling have a sizeable impact on our choice of compilation technique. The library construction is most useful for technologies with fast gate speeds such as photonics which mitigate the increased sequence length. Similarly at higher precisions, where the sequence length overhead is less pronounced, the library construction's faster compilation time make it the better choice.

As an example consider Linear Systems, which has a low minimum precision requirement, and Shortest Vector, which has a middling precision requirement. When compiling Linear systems for the faster technologies, there is no significant difference in performance between the library construction and SQCT methods. Meanwhile slower technologies cause the sequence length term to dominate giving SQCT an order of magnitude advantage over the library construction. However when compiling Shortest Vector, the library construction has a T count only one order of magnitude higher than SQCT. In this case, the increased sequence length is compensated for by the fast compilation time in all technologies, making library construction the preferable method.

5.5 Error Correction Level

In our study we consider technologies that make use of concatenated error correction codes. In this scheme qubits are encoded recursively to ensure fault tolerance. Circuits can be made more robust by increasing recursion depth, however each level of recursion is accompanied by a large increase in time and resources necessary for execution. As circuits grow, the accumulated probability of failure in their constituent components increases, necessitating adding levels of error correction. One concern is that the extra code generated by the library construction would require such an increase the recursion depth.

We performed a resource estimation for each algorithm using the Bacon-Shor error correction code[2] for all of the compilation methods described in section 4. We determined that for all of the algorithms, the increased circuit depth resulting from use of the library construction did not result in the need for another level of error correction. However, in situations where the benchmark is on the verge of requiring another level of error correction, larger sequence lengths could present a substantial liability and the library construction may not be the right choice.

5.6 Discussion

Our results show that dynamic compilation for quantum rotations can be extremely challenging. The widely-varying logical cycle times of quantum technologies, coupled with widely-varying precision requirements from different quantum benchmarks, result in a large design space.

For slower technologies, such as ion traps, the problem is considerably easier. It should be noted, however, that faster clock speeds are desirable even in quantum computation. Although quantum computations may have an exponential advantage over classical computations, the high error-correction overhead of quantum computers can make the crossover point between polynomial and exponential performance occur at 100 years of computation! Gigahertz technologies such as superconducting qubits could make quantum computation much more competitive, but we must be careful not to lose that advantage to time spent dynamically generating code.

Our library construction is a first step towards approaching this dynamic-code-generation problem. There is much work to be done in refining this and related approaches.

6. FUTURE WORK

Referring back to our results, we see that there does not yet exist a method of providing dynamic compilation in very high precision applications. Most of the benchmarks are scalable and, for more challenging problem sizes, can require precisions in excess of 10^{-50}. Implementing either of our static methods for such high accuracy would result in compilation times many orders of magnitude greater than the technology speed. Extrapolating the results of our dynamic method, it appears we method may be able to make such compilations on the fly, but only for the trapped ion technology. An important next step in this work is to extend compilation strategies to higher precisions while still being able to perform quick compilations. The results outlined above for the library construction method indicate that we can optimize the time and precision dimensions of our design space by expanding into the storage dimension. By utilizing large databases of pre-compiled high precision rotations, we will be able to perform on-the-fly compilation for more demanding applications.

Furthermore, we expect that any practical implementation of classical computing resources associated with high-precision quantum operations will require hardware implementation of precision greater than double precision floating point. This could be in the form of special-purpose accelerators or FPGAs. Higher precision targets will probably require a combination of hardware and software support. Any software extension of precision will most likely preclude dynamic compilation due to the associated time cost. Many of the algorithms in our benchmark suite such as Boolean Formula and Ground-State Estimation can have much higher precision targets (up to 10^{-54}). Fortunately, the rotation angles for these benchmarks are known at compile time. Under such targets, static compilation may be the only viable option and the caching and management of extremely large executables will be necessary.

7. CONCLUSION

For the first time, our work examines a broad set of quantum computing benchmarks and motivates static and dynamic compilation problems. In particular, we find that generating code for arbitrary rotations presents significant challenges. We optimized the established Solovay-Kitaev method (mostly appropriate for static compilation) and present our own dynamic method. We evaluate these methods alongside the field's current state-of-the-art in the context of benchmark and technology constraints. We find that the dynamic method offers up to five orders of magnitude speedup over existing compilation methods at a cost of one order of magnitude increase in code size. We observe that significant future work is needed in the both static and quantum compilation in order to make many quantum applications feasible.

8. ACKNOWLEDGEMENTS

The quantum benchmarks in this paper were the work of many researchers on our larger team, including John Black (Colorado); Lukas Svec, Aram Harrow (Washington); Chen-Fu Chiang (Sherbrooke); Amlan Chakrabarti (Princeton); Oana Catu (UCSB); and Mohammad Javad Dousti (USC). Kelly Stevens and Greg Mohler (GTRI) contributed to the technology and error correction estimates.

9. REFERENCES

[1] A. M. C. Andris Ambainis, B. W. Reichardt, R. Spalek, and S. Zhang. Every nand formula of size n can be evaluated in time $n^{1/2+o(1)}$ on a quantum computer, 2007.

[2] D. Bacon. Operator quantum error-correcting subsystems for self-correcting quantum memories. *Phys. Rev. A*, 73:012340, Jan 2006.

[3] C. Barthel, J. Medford, C. M. Marcus, M. P. Hanson, and A. C. Gossard. Interlaced dynamical decoupling and coherent operation of a singlet-triplet qubit. *Phys. Rev. Lett.*, 105:266808, Dec 2010.

[4] A. Bocharov and K. M. Svore. A depth-optimal canonical form for single-qubit quantum circuits, 2012.

[5] K. R. Brown, A. C. Wilson, Y. Colombe, C. Ospelkaus, A. M. Meier, E. Knill, D. Leibfried, and D. J. Wineland. Single-qubit-gate error below 10(-4) in a trapped ion. *Phys. Rev. A*, 84(3):030303, SEP 14 2011.

[6] W. C. Campbell, J. Mizrahi, Q. Quraishi, C. Senko, D. Hayes, D. Hucul, D. N. Matsukevich, P. Maunz, and C. Monroe. Ultrafast Gates for Single Atomic Qubits. *Phys. Rev. Lett.*, 105(9):090502, AUG 26 2010.

[7] E. Chi, S. A. Lyon, and M. Martonosi. Tailoring quantum architectures to implementation style: a quantum computer for mobile and persistent qubits. In D. M. Tullsen and B. Calder, editors, *ISCA*, pages 198–209. ACM, 2007.

[8] A. M. Childs, R. Cleve, E. Deotto, E. Farhi, S. Gutmann, and D. A. Spielman. Exponential algorithmic speedup by quantum walk. Proc. 35th ACM Symposium on Theory of Computing (STOC 2003), pp. 59-68, 2002.

[9] C. M. Dawson and M. A. Nielsen. The solovay-kitaev algorithm, 2005.

[10] A. Folwer. Constructing arbitrary Steane code single logical qubit fault-tolerant gates. *Quantum Information and Computation*, 11:867–873, 2011.

[11] T. Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.5 edition, 2012.

[12] L. Grover. A fast quantum mechanical algorithm for database search. *Symposium on Theory of Computing (STOC 1996)*, pages 212–219.

[13] B. Haible and R. B. Kreckel. Class library for numbers (cln).

[14] S. Hallgren. Polynomial-time quantum algorithms for pell's equation and the principal ideal problem. *J. ACM*, 54(1), 2007.

[15] A. W. Harrow, A. Hassidim, and S. Lloyd. Quantum algorithm for solving linear systems of equations. Phys. Rev. Lett. vol. 15, no. 103, pp. 150502 (2009), 2008.

[16] M. W. Johnson et al. Quantum annealing with manufactured spins. *Nature*, 473:194–198, 2011.

[17] J. Kempe. Quantum random walks - an introductory overview. Contemporary Physics, Vol. 44 (4), p.307-327, 2003, 2003.

[18] A. Y. Kitaev. Quantum computations: algorithms and error correction. *Russian Mathematical Surverys*, 52:1191–1249, 1997.

[19] V. Kliuchnikov, D. Maslov, and M. Mosca. Fast and efficient exact synthesis of single qubit unitaries generated by Clifford and T gates, 2013.

[20] E. Knill. Quantum computing with realistically noisy devices. *Nature*, 434:39–44, Nov 2005.

[21] L. Kreger-Stickles and M. Oskin. Microcoded architectures for ion-tap quantum computers. In *ISCA*, pages 165–176. IEEE, 2008.

[22] T. D. Ladd, F. Jelezko, R. Laflamme, Y. Nakamura, C. Monroe, and J. L. OBrien. Quantum computers. *Nature*, 464(7285):45–53, 2010.

[23] E. Magesan, J. M. Gambetta, B. R. Johnson, C. A. Ryan, J. M. Chow, S. T. Merkel, M. P. da Silva, G. A. Keefe, M. B. Rothwell, T. A. Ohki, M. B. Ketchen, and M. Steffen. Efficient measurement of quantum gate error by interleaved randomized benchmarking. *Phys. Rev. Lett.*, 109:080505, Aug 2012.

[24] R. V. Meter, K. Nemoto, W. J. Munro, and K. M. Itoh. Distributed arithmetic on a quantum multicomputer. In *ISCA*, pages 354–365. IEEE Computer Society, 2006.

[25] T. S. Metodi, D. D. Thaker, and A. W. Cross. A quantum logic array microarchitecture: Scalable quantum data movement and computation. In *MICRO*, pages 305–318. IEEE Computer Society, 2005.

[26] C. Monroe, C. Sackett, D. Kielpinski, B. King, C. Langer, V. Meyer, C. Myatt, M. Rowe, Q. Turchette, W. Itano, and D. Wineland. Scalable entanglement of trapped ions. *Workshop on Trapped Ion Quantum Computing (NIST, Boulder, Colorado)*, 2000.

[27] I. L. C. . M. A. Nielsen. *Quantum Computing and Quantum Information*. Cambridge University Press, Cambridge, 2000.

[28] S. Olmschenk, R. Chicireanu, K. D. Nelson, and J. V. Porto. Randomized benchmarking of atomic qubits in an optical lattice. *New Journal of Physics*, 12(11):113007, 2010.

[29] R. L. Phillip Kaye and M. Mosca. *An Introduction to Computing Computing*. Oxford University Press, Oxford, 2007.

[30] O. Regev. Quantum computation and lattice problems, 2003.

[31] P. J. Shadbolt, M. R. Verde, A. Peruzzo, A. Politi, A. Laing, M. Lobino, J. C. F. Matthews, M. G. Thompson, and J. L. O'Brien. Generating, manipulating and measuring entanglement and mixture with a reconfigurable photonic circuit. *Nature Photonics*, 6(1):45–49, JAN 2012.

[32] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[33] J. D. Whitfield, J. Biamonte, and A. Aspuru-Guzik. Simulation of electronic structure hamiltonians using quantum computers. Molecular Physics, Volume 109, Issue 5 March 2011 , pages 735 - 750, 2010.

[34] M. Whitney, N. Isailovic, Y. Patel, and J. Kubiatowicz. A fault tolerant, area efficient architecture for Shor's factoring algorithm. In S. W. Keckler and L. A. Barroso, editors, *ISCA*, pages 383–394. ACM, 2009.