Tools for Quantum and Reversible Circuit Compilation

Martin Roetteler(✉)

Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
martinro@microsoft.com

Abstract. We present tools for resource-aware compilation of higher-level, irreversible programs into lower-level, reversible circuits. Our main focus is on optimizing the memory footprint of the resulting reversible networks. We discuss a number of examples to illustrate our compilation strategy for problems at scale, including a reversible implementation of hash functions such as SHA-256, automatic generation of reversible integer arithmetic from irreversible descriptions, as well as a test-bench of Boolean circuits that is used by the classical Circuits and Systems community. Our main findings are that, when compared with Bennett's original "compute-copy-uncompute", it is possible to reduce the space complexity by 75% or more, at the price of having an only moderate increase in circuit size as well as in compilation time. Finally, we discuss some emerging new paradigms in quantum circuit synthesis, namely the use of dirty ancillas to save overall memory footprint, probabilistic protocols such as the RUS framework which can help to reduce the gate complexity of rotations, and synthesis methods for higher-dimensional quantum systems.

Keywords: Quantum circuits · Reversible circuits · Quantum programming languages · Pebble games · Dirty ancillas · Repeat-Until-Success protocols · Ternary systems

1 Introduction

The compilation of quantum algorithms into sequences of instructions that a quantum computer can execute requires a multi-stage framework. This framework needs to be capable of taking higher level descriptions of quantum programs and successively breaking them down into lower level net-lists of circuits until ultimately pulse sequences are obtained that a physical machine can apply. Independent of the concrete realization of the compilation method, one of the key steps is to implement subroutines[1] over the given target instruction set. As often the underlying problem is a classical problem in that the problem specification involves classical data (such as finding the period of a function or searching an assignment that satisfies a given Boolean predicate), the question arises how

[1] In quantum computing literature, such subroutines are often implementing "oracles".

I. Phillips and H. Rahaman (Eds.): RC 2017, LNCS 10301, pp. 3–16, 2017.
DOI: 10.1007/978-3-319-59936-6_1

such functions can best be implemented on a quantum computer. Examples are Shor's algorithm for factoring and the computation of discrete logarithms over a finite field [38], Grover's quantum search algorithm [23], quantum walk algorithms [27], the HHL algorithm for solving linear equations [16,25], and quantum simulation methods [8,9]. The field of reversible computing deals with the latter problem and investigates such issues as how to minimize the gate count over a given universal gate set and how to minimize various other resources, such as the circuit depth, the total number of qubits required, and other metrics.

There are many ramifications to this compilation problem. Some stem from the choice of programming language to express the tools that perform the translation. Choices that have been reported in the literature range from C-like languages such as QCL [32] and Scaffold [26] to functional languages such as Quipper [21,22] and LIQ$Ui|\rangle$ [40]. Further choices involve the methods to compile classical, irreversible programs into quantum circuits and several approaches have been taken in the literature. One approach is to hide all classical subroutines in libraries and to provide an optimized collection of functions to implement these. This is the approach taken in several languages and as long as quantum programming remains a very much circuit-centric endeavor, this approach might well be appropriate. On the other hand, tools that allow the translation of classical, irreversible code into, say, networks of Toffoli gates have been developed: in the Haskell-based Quipper language, there is a monadic bind to lift classical computation to reversible circuits. In the LIQ$Ui|\rangle$ there is Revs [34], a tool to perform the task of obtaining reversible networks automatically from a little language that can be used to express classical programs.

The main idea behind Revs is to improve on Bennett's [6] method to make computations reversible: arbitrary computations can be carried out by a computational device in such a way that in principle each time-step can be reversed by first performing a forward computation, using only step-wise reversible processes, then copying out the result, and finally undoing all steps in the forward computation in reverse order. This solves the reversible embedding problem, albeit at the cost of very large memory-requirements as the result from each intermediate process have to be stored. Bennett already pointed out a solution [7] that is applicable in principle to reduce the memory-overhead by studying time-space trade-offs for reversible computation. He introduced the notion of reversible pebble games which allow to systematically study ways to save on scratch space at the expense of recomputing intermediate results. To determine the best pebbling strategy for the dependency graph imposed by actual real-world programs and to automate the process of pebbling in general, however, are non-trivial matters. In the Revs framework, we follow a pragmatic approach: (i) Boolean functions are synthesized directly using various heuristics and optimizations, such as exclusive-sum-of-products (ESOP) based optimization [15,30], (ii) the compiler provides different strategies for making irreversible computations reversible: one is Bennett's method, another is heuristic that computes data dependencies in the source program and tries to uncompute data that is no longer needed as soon as possible.

As a real-world example we consider cryptographic hash-functions such as SHA-256, which is part of the SHA-2 family [1]. This cipher can be thought of as a Boolean function $f : \{0,1\}^N \to \{0,1\}^n$, where $n \ll N$. It has a simple and straightforward classical program for its evaluation that has no branchings and only uses simple Boolean functions such as XOR, AND, and bit rotations. However, it has internal *state* between rounds. The fact that there is state prevents the Boolean function from being decomposed, thereby making purely truth-table or BDD-based synthesis methods useless for this problem.

The basic underlying fault-tolerant architecture and coding scheme determines the universal gate set, and hence by extension also the synthesis problems that have to be solved in order to compile high-level, large-scale algorithms into a sequence of operations that an actual physical quantum computer can then execute. A gate set that arises frequently and that has been oft studied in the literature, but by no means the only conceivable gate set, is the so-called Clifford+T gate set [31]. This gate set consists of the Hadamard gate $H = \frac{1}{\sqrt{2}} \left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right]$, the phase gate $P = \mathrm{diag}(1, i)$, and the CNOT gate which maps $(x, y) \mapsto (x, x \oplus y)$ as generators of the Clifford group, along with the T gate given by $T = \mathrm{diag}(1, \exp(\pi i/4))$. The Clifford+$T$ gate set is known to be universal [31], i.e., any given target unitary single qubit operation can be approximated to within ε using sequences of length $4 \log_2(1/\varepsilon)$ [28,37] and using an entangling gate such as the controlled NOT gate. Often, only T-gates are counted as many fault-tolerant implementation of the Clifford+T gate set at the logical gate level require much more resources [19] for T-gates than for Clifford gates. We based reversible computations entirely on the Toffoli gate $|x, y, z\rangle \mapsto |x, y, z \oplus xy\rangle$ which is known to be universal for reversible computing [31] and which can be implemented exactly over the Clifford+T gate set, see [36] for T-depth 1 implementation using a total of 7 qubits and [3] for a T-depth 3 realization using a total of 3 qubits.

2 Data Dependency Analysis in Revs

Data dependencies that might be present in a given F# program are modeled in Revs using a data structure called a mutable data dependency graph (MDD). This data structure tracks the data flow during a classical, irreversible computation. MDDs allow to identify parts of the data flow where information can be overwritten as well as other parts where information can be uncomputed early as it is no longer needed. These two techniques of *overwrite*, which are implemented using so-called in-place operations, and *early cleanup*, for which we use a strategy that can be interpreted as a particular pebble game played on the nodes of the data flow graph, constitute the main innovation of the present work. The cleanup methods described here can be thought of as an analog to garbage collection for quantum architectures. Revs outputs a Toffoli network which then can directly imported as an internal representation into LIQ$Ui|\rangle$ and be used as part of another quantum computation.

```
let rippleAdd (a:bool[])(b:bool[])=
   let n = Array.length a
   let res =  Array.zeroCreate (n)
   res.[0] <- a.[0] <> b.[0]
   let mutable carry = a.[0] && b.[0]
   res.[1] <- a.[1] <> b.[1] <> carry
   for i in 2 .. n - 1 do
      // compute outgoing carry
      carry <- (a.[i-1]
               && (carry <> b.[i-1]))
               <> (carry && b.[i-1])
      res.[i]  <-  a.[i] <> b.[i]
                   <> carry
   res
```

(a) REVS source program

```
CNOT [qs.[0]; qs.[10]]
CNOT [qs.[5]; qs.[10]]
CCNOT [qs.[0]; qs.[5]; qs.[15]]
CNOT [qs.[15]; qs.[11]]
CNOT [qs.[1]; qs.[11]]
CNOT [qs.[6]; qs.[11]]
CNOT [qs.[15]; qs.[17]]
CNOT [qs.[6]; qs.[17]]
CCNOT [qs.[1]; qs.[17]; qs.[16]]
CCNOT [qs.[15]; qs.[6]; qs.[16]]
CNOT [qs[6]; qs.[17]]
CNOT [qs.[15]; qs.[17]]
CNOT [qs.[16]; qs.[12]]
...
```

(b) Segment of the compiled LIQ$Ui|\rangle$ program

Fig. 1. F# program that implements a carry ripple adder using a for-loop and maintaining a running carry.

REVS is an embedded language into the .NET language F# and as such inherits some functions and libraries from its host language. Also, the look-and-feel of a typical REVS program is very similar to that of F# programs. In fact, it is one of our design goals to provide a language that provides different *interpretations* of the same source program, i.e., the same source code can be compiled into (a) an executable for a given classical architecture such as the .NET CLR, (b) a Toffoli network, (c) rendered form of output, e.g., pdf or svg, or (d) an internal representation which can then be simulated efficiently on a classical computer.

The current implementation of the REVS compiler supports Booleans as basic types only. The core of the language is a simple imperative language over Boolean and array (register) types. The language is further extended with ML-style functional features, namely first-class functions and *let* definitions, and a reversible domain-specific construct *clean*. It should be noted also that REVS was designed to facilitate interoperability with the quantum programming language LIQ$Ui|\rangle$ which is also F# based and which provides rich support for expressing and simulating quantum circuits on classical machines, but which also provides support for compiling quantum algorithms for target hardware architectures and abstract quantum computer machine models.

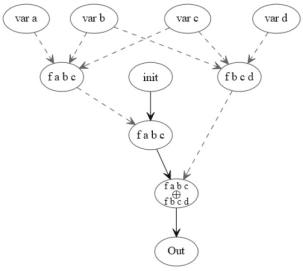
An example REVS program is shown in Fig. 1(a). This example implements a simple carry ripple adder of two n-bit integers. Shown in (b) is one of the possible target intermediate representations, namely LIQ$Ui|\rangle$ code.

At a high level, all compilation strategies that are implemented in REVS proceed start from a classical description of the given function which is then turned into an abstract syntax tree (AST) by a parser. This level might use libraries and further optimizations by the F# compiler. The subsequent levels are domain-specific to the reversible synthesis domain and use the MDD data-structure presented in [34]. See Fig. 2 for an example. The overall compilation can use pre-computed libraries, e.g., for reversible arithmetic and other optimized functions.

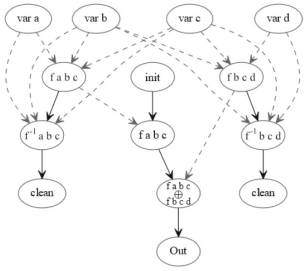
It should be noted that possibly the overall compilation can fail, namely in case the given target strategy cannot be implemented using the given upper bound on the number of available qubits.

3 An Example at Scale: SHA-256

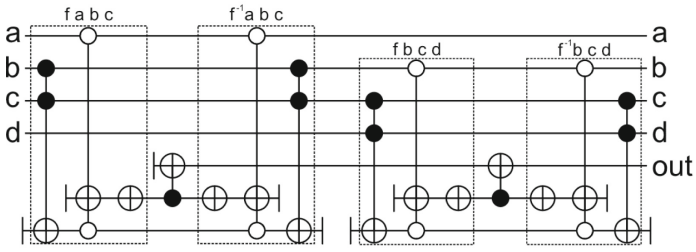
We implemented the round function of SHA-256 which is a hash function as specified in the FIPS 180-2 publication [1]. Like many other hash functions, SHA proceeds in a round-like fashion and uses the current state of a finite state machine, the next incoming data block, and various constants in order to define the next state of the finite state machine. In the round function of the cipher 32 bit registers A, B, ..., E are needed. The following Boolean functions are introduced to describe the round functions:



(a) MDD for h before cleanup



(b) MDD for h after eager cleanup



(c) Final resulting Toffoli network implementing the function h.

Fig. 2. Shown in (a) is the mutable data dependency graph (MDD) for the function $h(a,b,c,d) = f(a,b,c) \oplus f(b,c,d)$ where $f(a,b,c) = a \| (b \& c)$. Shown in (b) is the MDD that results in applying Eager cleanup (as described in [34]) to the MDD in (a). Shown in (c) is the final circuit that REVS emits based on the MDD in (b). Qubits that are initially clean are shown as $\vdash$, qubits that terminate in a clean state are shown as $\dashv$. Overall, the circuit uses a total of 7 qubits to compute the function h. This should be compared with applying the Bennett cleanup which would result in a much larger number of qubits, namely 11.

$$
\begin{aligned}
Ch(E,F,G) &:= (E \wedge F) \oplus (\neg E \wedge G) \\
Ma(A,B,C) &:= (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C) \\
\Sigma_0(A) &:= (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22) \\
\Sigma_1(E) &:= (E \ggg 6) \oplus (E \ggg 11) \oplus (E \ggg 25).
\end{aligned}
$$

For a given round, the values of all these functions is computed and considered to be 32 bit integers. Further, a constant 32 integer value K_i is obtained from a table lookup which depends on the number i of the given round, where $i \in \{0, \ldots, 63\}$ and finally the next chunk of the message W_i is obtained from the message after performing a suitable message expansion is performed as specified in the standard. Finally, H is replaced according to $H \leftarrow H + Ch(E,F,G) + Ma(A,B,C) + \Sigma_0(A) + \Sigma_1(E) + K_i + W_i$ and then the cyclic permutation $A \leftarrow H, B \leftarrow A, \ldots, H \leftarrow G$ is performed. The implementation of the entire round function for a given number of rounds n was presented in [34] using the Revs high-level language.

To test the performance of the Revs compiler, in [34] we hand-optimized an implementation of SHA-256. This circuit contains 7 adders (mod 2^{32}). Using the adder from [17] with a Toffoli cost of $2n - 3$ this corresponds to 61 Toffoli gates per adder or 427 per round.

Next, we used Revs to produce Toffoli networks for this cipher, depending on various increments of the number n of rounds. The circuits typically are too large to be visualized in printed form, however, an automatically generated .svg file that the LIQ$Ui|\rangle$ compiler can be navigated by zooming in down to the level of Toffoli, CNOT, and NOT gates. The resource estimates are summarized in Table 1. Shown are the resulting circuit sizes, measured by the total number of Toffoli gates, the resulting total number of qubits, and the time it took to compile the circuit for various numbers of rounds. All timing data in the table

Table 1. Comparison of different compilation strategies for the cryptographic hash function SHA-256.

Rnd	Bennett			Eager			Reference	
	Bits	Gates	Time	Bits	Gates	Time	Bits	Gates
1	704	1124	0.254	353	690	0.329	353	683
2	832	2248	0.263	353	1380	0.336	353	1366
3	960	3372	0.282	353	2070	0.342	353	2049
4	1088	4496	0.282	353	2760	0.354	353	2732
5	1216	5620	0.290	353	3450	0.366	353	3415
6	1344	6744	0.304	353	4140	0.378	353	4098
7	1472	7868	0.312	353	4830	0.391	353	4781
8	1600	8992	0.328	353	5520	0.402	353	5464
9	1728	10116	0.334	353	6210	0.413	353	6147
10	1856	11240	0.344	353	6900	0.430	353	6830

are measured in seconds and resulted from running the F# compiler in Visual Studio 2013 on an Intel i7-3667 @ 2GHz 8 GB RAM under Windows 8.1. The table shows savings of almost $4X$ in terms of the total numbers of qubits required to synthesize the cipher when comparing the simple Bennett cleanup strategy versus the Eager cleanup strategy. The reason for this is that the Bennett cleanup methods allocates new space essentially for each gate whereas the Eager cleanup strategy tries to clean up and reallocate space as soon as possible which for the round-based nature of the function can be done as soon as the round is completed.

Besides SHA-256, and other hash functions such as MD5, this technique has also been applied to SHA-3 [4]. Our findings supports the thesis that it is possible to trade circuit size (time) for total memory (space) in reversible circuit synthesis. To the best of our knowledge, Revs is the first compiler that allows to navigate this trade space and that offers strategies for garbage collection for quantum architectures that go beyond the simple Bennett strategy which generally leads to very poor memory utilization as most of the qubits are idle most of the time.

4 Quantum Computing Software Architecture

Revs is part of a larger framework provided by the LIQ$Ui|\rangle$ software architecture. LIQ$Ui|\rangle$ is a quantum programming language and a high-performance simulator for quantum circuits. LIQ$Ui|\rangle$ is an embedded language into F# which itself is a full .NET language, i.e., F# supports object-oriented, imperative and functional programming, as well as ease of using reflection and pattern matching which helps with walking complex datastructures. LIQ$Ui|\rangle$ can be obtained from https://github.com/StationQ/Liquid. Runtimes supported in LIQ$Ui|\rangle$ are client/server versions, as well as an Azure based cloud service. There are several ways in which LIQ$Ui|\rangle$ code can be executed, e.g., from the command line running the .NET Common Language Runtime, or directly in a Visual Studio interactive session (particularly useful for script files), or in a normal Visual Studio development mode.

The Revs compiler can compile classical, irreversible code into functions that can then be further processed, e.g., by using simulators in LIQ$Ui|\rangle$. An example are Toffoli networks for specific functions such as the SHA-256 example from the previous section. These circuits can then be executed by various simulation backends that are available in LIQ$Ui|\rangle$, e.g., a full functional simulator which can simulate arbitrary circuits on up to 32 qubits using about 32 GB of memory, or a special purpose Toffoli simulator which can be used, e.g., to simulate large Toffoli networks to implement controlled modular multiplication. For the latter see e.g. [24] where simulation of modular multiplication networks have been reported for bit sizes up to $8,192$.

5 Other Paradigms for Quantum and Reversible Synthesis

5.1 Using Dirty Ancillas

By *dirty* ancillas we mean qubits which can be in an unknown state, possibly entangled with other qubits in an unknown way, but which are available as scratch space for other computations. There are not many use cases of this situation and a priori it seems even difficult to imagine any situation where such a "full quantum memory" could be of use at all as any manipulation that uses dirty ancillas without restoring them to their state before they were used, will destroy interferences between computational paths.

So far, we are aware of two specific situations where dirty ancillas help: (i) the implementation of a multiply controlled NOT operation, see [5] and recent improvements [2,29]. The second use case is an implementation of a constant incrementer $|x\rangle \mapsto |x+c\rangle$, where c is an integer that is known at compile time and x an input that can be in superposition. In [24] it was shown that dirty ancillas help to realize this operation using $O(n \log n)$ Toffoli gates and a total of n qubits which are needed to represent x, along with $O(n)$ dirty ancillas. This in turn can be used to implement the entire Shor algorithm using almost entirely Toffoli gates.[2]

Table 2. Costs associated with various implementations of addition $|a\rangle \mapsto |a+c\rangle$ of a value a by a classical constant c.

	Cuccaro et al. [17]	Takahashi et al. [39]	Draper [18]	Häner et al. [24]
Size	$\Theta(n)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n \log n)$
Depth	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Ancillas	n+1 (clean)	n (clean)	0	$\frac{n}{2}$ (dirty)

Mathematically, the underlying idea how to make use of dirty ancillas can be illustrated in case of an addition "+1" which is an observation due to Gidney [20]: Using the ancilla-free adder by Takahashi [39], which requires no incoming carry, and its reverse to perform subtraction, one can perform the following sequence of operations to achieve an incrementer using n borrowed ancilla qubits in an unknown initial state $|g\rangle$:

$$\begin{aligned} |x\rangle|g\rangle &\mapsto |x-g\rangle|g\rangle \mapsto |x-g\rangle|g'-1\rangle \\ &\mapsto |x-g-g'+1\rangle|g'-1\rangle \mapsto |x+1\rangle|g\rangle, \end{aligned}$$

[2] Indeed, the only non-Toffoli gates in the quantum circuit presented in [24] are single qubit Hadmard gates, single qubit phase rotations, and single qubit measurements. The vast majority of other gates in the circuit form one big circuit component which can be classically simulated and tested.

where g' denotes the two's-complement of g and $g' - 1 = \overline{g}$, the bit-wise complement of g. Notice that $g + g' = 0$ holds for all g and that the register holding the dirty qubits $|g\rangle$ is returned to its initial state.

Table 2 provides a comparison between different ways to implement addition on a quantum computer with the last column being the implementation based on dirty ancillas.

In total, using the standard phase estimation approach to factoring this leads to an $\mathcal{O}(n^3 \log n)$-sized implementation of Shor's algorithm from a Toffoli based in-place constant-adder, which adds a classically known n-bit constant c to the n-qubit quantum register $|a\rangle$, i.e., which implements $|a\rangle|0\rangle \mapsto |a + c\rangle$ where a is an arbitrary n-bit input and $a + c$ is an n-bit output (the final carry is ignored).

5.2 Repeat-Until-Success Circuits

Recently, Paetznick and Svore [33] showed that by using non-deterministic circuits for decomposition, called Repeat-Until-Success (RUS) circuits, the number of T gates can be further reduced by a factor of 2.5 on average for axial rotations, and by a larger factor for non-axial rotations. They emphasized that synthesis into RUS circuits can lead to a shorter *expected* circuit length that surpasses the theoretical lower bound for the length of a purely unitary circuit design. Leveraging the RUS framework, in [12,13] efficient algorithms were presented to synthesize a non-deterministic Repeat-Until-Success (RUS) circuits for approximating any given single-qubit unitary. Our algorithm runs in probabilistically polynomial classical runtime for any desired precision ε. Our methods demonstrate the power of using ancilla qubits and measurement for quantum circuit compilation.

The general layout of a RUS protocol is shown in Fig. 3. Consider a unitary operation U acting on $n + m$ qubits, of which n are target qubits and m are ancillary qubits. Consider a measurement of the ancilla qubits, such that one measurement outcome is labeled "success" and all other measurement outcomes are labeled "failure". Let the unitary applied to the target qubits upon measurement be V. In the RUS protocol, the circuit in the dashed box is repeated on the $(n + m)$-qubit state until the "success" measurement is observed. Each time a "failure" measurement is observed, an appropriate Clifford operator $W_i^\dagger$ is applied in order to revert the state of the target qubits to their original input state $|\psi\rangle$. The number of repetitions of the circuit is finite with probability 1.

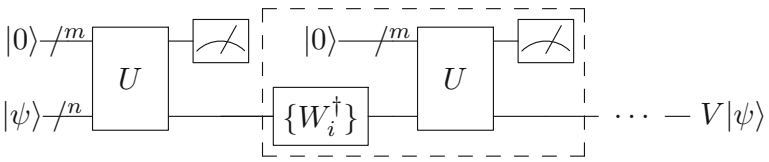
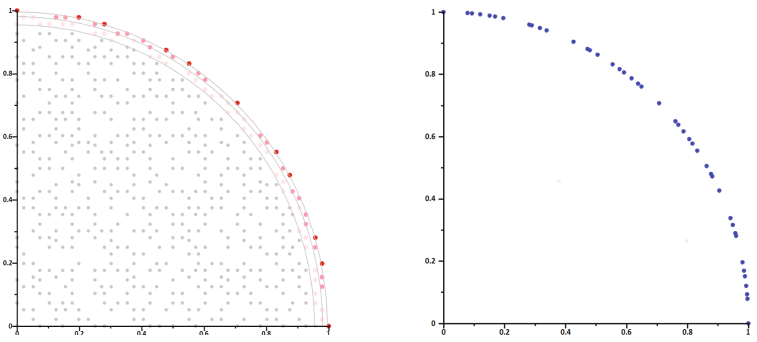


Fig. 3. Repeat-Until-Success (RUS) protocol to implement a unitary V.

In [12,13] efficient algorithms were given to synthesize RUS protocols and so-called fallback protocols which also allow to implement unitary gates using probabilistic circuits. The inputs to the synthesis algorithms are the given unitary U, typically assumed to be a Z-rotation, and a target accuracy ε. Under mild number-theoretic conjectures, the complexity of the compilation method is in $\widetilde{O}(\log(1/\varepsilon))$ and the length of the output, i.e., a sequence of H and T gates that ε-approximates U, scales as $(1+\delta)\log_2(1/\varepsilon)$, where δ can be made arbitrary close to 0. These results demonstrate the power of using ancilla qubits and measurement for quantum circuit compilation as the currently best known deterministic schemes lead to lengths of the resulting circuits that scale as $c\log_2(1/\varepsilon)$, where $3 \leq c \leq 4$, with the actual choice of c depending on various computational and number-theoretic assumptions. See [12,13,28,35,37] for further reading about single qubit unitary decomposition methods. Figure 4 conveys the basic intuition behind RUS based methods: by allowing measurement and, if needed, repetition, it is possible to achieve a much higher density of rotations that can effectively be addressed.



(a) (b)

Fig. 4. Comparing approximations of z-rotations by (a) unitary $\langle H, T\rangle$ circuits of T-depth at most 8 and (b) RUS protocols with a comparable expected T-depth of at most 7.5. In (a) only 40 circuits are close to any z-rotation (dark red points separated by the outer arc), illustrating a higher density of RUS protocols. In the asymptotic limit for T-depth this ratio tends to 3. (Color figure online)

5.3 Higher-Dimensional Alphabets

It turns out that some fault-tolerant scalable quantum computing schemes underline the importance to work with higher-dimensional alphabets to encode quantum information. In particular, a ternary quantum framework recently emerged from proposals for a metaplectic topological quantum computer (MTQC) which offers native topological protection of quantum information. MTQC creates an inherently ternary quantum computing environment; for example the common binary CNOT gate is no longer a Clifford gate in that environment.

In [11], compilation and synthesis methods for ternary circuits were developed for 2 different elementary gate sets: the so-called Clifford+$R_{|2\rangle}$ basis [10] and the Clifford+P_9 basis [11], where $R_{|2\rangle}$ and P_9 are both non-Clifford single qutrit gate defined as $R_{|2\rangle} = \mathrm{diag}(1, 1, -1)$ and $P_9 = \mathrm{diag}(e^{-2\pi\, i/9}, 1, e^{2\pi\, i/9})$.

The Clifford+$R_{|2\rangle}$ basis, also called metaplectic basis, was obtained from a MTQC by braiding of certain metaplectic non-abelian anyons and projective measurement. The gate $R_{|2\rangle}$ is produced by injection of the magic state $|\psi\rangle = |0\rangle - |1\rangle + |2\rangle$. The injection circuit is coherent probabilistic, succeeds in three iterations on average and consumes three copies of the magic state $|\psi\rangle$ on average. The $|\psi\rangle$ state is produced by a relatively inexpensive protocol that uses topological measurement and consequent intra-qutrit projection. This protocol requires only three qutrits and produces an exact copy of $|\psi\rangle$ in 9/4 trials on average. This is much better than any state distillation method, especially because it produces $|\psi\rangle$ with fidelity 1. In [10] effective compilation methods for Clifford+$R_{|2\rangle}$ were developed to compile efficient circuits in the metaplectic basis. In particular, given an arbitrary two-level Householder reflection r and a precision ε, then r is effectively approximated by a metaplectic circuit of $R_{|2\rangle}$-count at most $C\, \log_3(1/\varepsilon) + O(\log(\log(1/\varepsilon)))$, $C \leq 8$.

The Clifford+P_9 basis is a natural generalization of the binary $\pi/8$ gate. The P_9 gate can be realized by a certain deterministic measurement-assisted circuit given a copy of the *magic* state $\mu = e^{-2\pi\, i/9}|0\rangle + |1\rangle + e^{2\pi\, i/9}|2\rangle$, which further can be obtained from the usual magic state distillation protocol. Specifically, it requires $O(\log^3(1/\delta))$ raw magic states of low fixed fidelity in order to distill a copy of the magic state μ at fidelity $1 - \delta$. The paper [11] developed a novel approach to synthesis of reversible ternary classical circuits over the Clifford+P_9 basis. We have synthesized explicit circuits to express classical reflections and other important classical non-Clifford gates in this basis, which we subsequently used to build efficient ternary implementations of integer adders and their extensions.

In [14] further optimizations were given under the assumption of binary-encoded data and applied the resulting solutions to emulating of the modular exponentiation period finding (which is the quantum part of the Shor's integer factorization algorithm). We have performed the comparative cost analysis of optimized solutions between the "generic" Clifford+P_9 architecture and the MTQC architecture (the Clifford+$R_{|2\rangle}$) using magic state counts as the cost measure. We have shown that the cost of emulating the entire binary circuit for the period finding is almost directly proportional to the cost of emulating the three-qubit Toffoli gate and the latter is proportional to the cost of the P_9 gate.

6 Conclusions

We presented Revs, a compiler and programming language that allows to automate the translation of classical, irreversible programs into reversible programs. This language does not constrain the programmer to think in a circuit-centric way. In some cases (e.g., hash functions such as SHA-256) the savings of our

method over Bennett-style approaches can even be unbounded. We navigate the PSPACE completeness of finding the optimal pebble game by invoking heuristic strategies that identify parts of the program that are mutable which then can be implemented via in-place operations. In order to manage the arising data dependencies, we introduced MDD graphs which capture data dependencies as well as data mutation. Using an example benchmark suite compiled from classical circuits and systems community, we show that the method can be applied for medium to large scale problems. We also showed that hash functions such as SHA-256 can be compiled into space-optimized reversible circuits.

Also, we highlighted that there are paradigms that break out of the usual framework considered in quantum circuit synthesis: we highlighted that there are concrete case in which the presence of qubits helps, even if they have already been used and are entangled with the rest of the quantum computer's memory. Using such dirty ancillas, it is possible to reduce circuit sizes e.g. for constant incrementers. Next, discussed the power of using probabilistic protocols to implement unitaries, which helps to bring down circuit sizes by a constant factor. In a concrete case, probabilistic protocols such as RUS or fallback schemes help to reduce the cost for single qubit axial rotations from $4\log_2(1/\varepsilon)$ to $\log_2(1/\varepsilon)$. Finally, we mentioned that in some physical systems, ternary alphabets arise very naturally from the way universal operations are performed. We mentioned two such gate sets and gave pointers to method for synthesizing into these gate sets.

References

1. Federal information processing standards publication 180–2, 2002. See also the Wikipedia entry. http://en.wikipedia.org/wiki/SHA-2
2. Abdessaied, N., Amy, M., Drechsler, R., Soeken, M.: Complexity of reversible circuits and their quantum implementations. Theor. Comput. Sci. **618**, 85–106 (2016)
3. Amy, M., Maslov, D., Mosca, M., Roetteler, M.: A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. **32**(6), 818–830 (2013)
4. Amy, M., Di Matteo, O., Gheorghiu, V., Mosca, M., Parent, A., Schanck, J.M.: Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3. IACR Cryptol. ePrint Arch. **2016**, 992 (2016)
5. Barenco, A., Bennett, C.H., Cleve, R., DiVincenzo, D.P., Margolus, N., Shor, P., Sleator, T., Smolin, J.A., Weinfurter, H.: Elementary gates for quantum computation. Phys. Rev. A **52**(5), 3457 (1995)
6. Bennett, C.H.: Logical reversibility of computation. IBM J. Res. Dev. **17**, 525–532 (1973)
7. Bennett, C.H.: Time/space trade-offs for reversible computation. SIAM J. Comput. **18**, 766–776 (1989)
8. Berry, D.W., Childs, A.M., Cleve, R., Kothari, R., Somma, R.D.: Exponential improvement in precision for simulating sparse hamiltonians. In: Symposium on Theory of Computing (STOC 2014), pp. 283–292 (2014)
9. Berry, D.W., Childs, A.M., Kothari, R.: Hamiltonian simulation with nearly optimal dependence on all parameters. In: IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS), pp. 792–809 (2015)

10. Bocharov, A., Cui, S.X., Kliuchnikov, V., Wang, Z.: Efficient topological compilation for weakly-integral anyon model. Phys. Rev. A **93**, 012313 (2016)
11. Bocharov, A., Cui, S.X., Roetteler, M., Svore, K.M.: Improved quantum ternary arithmetics. Quantum Inf. Comput. **16**(9&10), 862–884. arXiv preprint (2016). arXiv:1512.03824
12. Bocharov, A., Roetteler, M., Svore, K.M.: Efficient synthesis of probabilistic quantum circuits with fallback. Phys. Rev. A **91**, 052317 (2015)
13. Bocharov, A., Roetteler, M., Svore, K.M.: Efficient synthesis of universal repeat-until-success circuits. Phys. Rev. Lett. **114**, 080502. arXiv preprint (2015). arXiv:1404.5320
14. Bocharov, A., Roetteler, M., Svore, K.M.: Factoring with qutrits: Shor's algorithm on ternary and metaplectic quantum architectures. arXiv preprint (2016). arXiv:1605.02756
15. Chrzanowska-Jeske, M., Mishchenko, A., Perkowski, M.A.: Generalized inclusive forms - new canonical reed-muller forms including minimum esops. VLSI Des. **2002**(1), 13–21 (2002)
16. Clader, B.D., Jacobs, B.C., Sprouse, C.R.: Preconditioned quantum linear system algorithm. Phys. Rev. Lett. **110**, 250504 (2013)
17. Cuccaro, S.A., Draper, T.G., Kutin, S.A., Moulton, D.P.: A new quantum ripple-carry addition circuit. arXiv preprint (2004). arXiv:quant-ph/0410184
18. Draper, T.G.: Addition on a quantum computer. arXiv preprint (2000). arXiv:quant-ph/0008033
19. Fowler, A.G., Mariantoni, M., Martinis, J.M., Cleland, A.N.: Surface codes: towards practical large-scale quantum computation. Phys. Rev. A **86**, 032324 (2012). arXiv:1208.0928
20. Gidney, C.: StackExchange: creating bigger controlled nots from single qubit, toffoli, and CNOT gates, without workspace (2015)
21. Green, A.S., Lumsdaine, P.L.F., Ross, N.J., Selinger, P., Valiron, B.: An introduction to quantum programming in quipper. In: Dueck, G.W., Miller, D.M. (eds.) RC 2013. LNCS, vol. 7948, pp. 110–124. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38986-3_10
22. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: Quipper: a scalable quantum programming language. In: Proceedings of Conference on Programming Language Design and Implementation (PLDI 2013). ACM (2013)
23. Grover, L.: A fast quantum mechanical algorithm for database search. In: Proceedings of the Symposium on Theory of Computing (STOC 1996), pp. 212–219. ACM Press (1996)
24. Häner, T., Roetteler, M., Svore, K.M. Factoring using $2n+2$ qubits with Toffoli based modular multiplication. arXiv preprint (2016). arXiv:1611.07995
25. Aram, W., Harrow, A.H., Lloyd, S.: Quantum algorithm for linear systems of equations. Phys. Rev. Lett. **103**(15), 150502 (2009)
26. Heckey, J., Patil, S., JavadiAbhari, A., Holmes, A., Kudrow, D., Brown, K.R., Franklin, D., Chong, F.T., Martonosi, M.: Compiler management of communication and parallelism for quantum computation. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015), pp. 445–456. ACM (2015)
27. Kempe, J.: Quantum random walks - an introductory overview. Contemporary Phys. **44**(4), 307–327 (2003)
28. Kliuchnikov, V., Maslov, D., Mosca, M.: Practical approximation of single-qubit unitaries by single-qubit quantum Clifford and T circuits. IEEE Trans. Comput. **65**(1), 161–172 (2016)

29. Maslov, D.: On the advantages of using relative phase Toffolis with an application to multiple control Toffoli optimization. Phys. Rev. A **93**, 022311 (2016)
30. Mishchenko, A., Brayton, R.K., Chatterjee, S.: Boolean factoring and decomposition of logic networks. In: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, pp. 38–44. IEEE Press (2008)
31. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press, Cambridge (2000)
32. Oemer, B.: Classical concepts in quantum programming. Int. J. Theor. Phys. **44**(7), 943–955 (2005)
33. Paetznick, A., Svore, K.M.: Repeat-until-success: non-deterministic decomposition of single-qubit unitaries. Quantum Inf. Comput. **4**(15&16), 1277–1301 (2014)
34. Parent, A., Roetteler, M., Svore, K.M.: Reversible circuit compilation with space constraints. arXiv preprint (2015). arXiv:1510.00377
35. Ross, N.J., Selinger, P.: Optimal ancilla-free Clifford+T approximation of z-rotations. arXiv preprint (2014). arXiv:403.2975
36. Selinger, P.: Quantum circuits of T-depth one. Phys. Rev. A **87**, 042302 (2013)
37. Selinger, P.: Efficient Clifford+T approximation of single-qubit operators. Quantum Inf. Comput. **15**(1–2), 159–180 (2015)
38. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. **26**(5), 1484–1509 (1997)
39. Takahashi, Y., Tani, S., Kunihiro, N.: Quantum addition circuits, unbounded fan-out. arXiv preprint (2009). arXiv:0910.2530
40. Wecker, D., Svore, K.M.: LIQ Ui|⟩: a software design architecture and domain-specific language for quantum computing. arXiv preprint arXiv:1402.4467