REVIEW

doi:10.1038/nature23459

Programming languages and compiler design for realistic quantum hardware

Frederic T. Chong[1], Diana Franklin[1] & Margaret Martonosi[2]

Quantum computing sits at an important inflection point. For years, high-level algorithms for quantum computers have shown considerable promise, and recent advances in quantum device fabrication offer hope of utility. A gap still exists, however, between the hardware size and reliability requirements of quantum computing algorithms and the physical machines foreseen within the next ten years. To bridge this gap, quantum computers require appropriate software to translate and optimize applications (toolflows) and abstraction layers. Given the stringent resource constraints in quantum computing, information passed between layers of software and implementations will differ markedly from in classical computing. Quantum toolflows must expose more physical details between layers, so the challenge is to find abstractions that expose key details while hiding enough complexity.

In the fifty years since Gordon Moore first predicted the exponential technology scaling now known as Moore's law[1], the hardware resources in classical computers have scaled dramatically with improvements in fabrication technology, allowing software to prioritize increased abstraction and modularity over stringent efficiency in hardware usage. Thus, Moore's law scaling has indirectly allowed classical computer systems to experience faster runtimes and improved portability and programmability with each generation of technology.

Figure 1 depicts how the design and programming of systems has evolved over the years. In the 1940s and 1950s, computers were programmed using very low-level assembly language, and few design tools or compilers existed. Today's classical (non-quantum) computers manage massive hardware and software complexity (billions of transistors and lines of code, respectively) by layering many abstractions and tools. These layers that connect algorithms to devices are often termed a 'software toolchain', in which one layer feeds into another to form a chain of input–output transformations that can take a program written in a high-level language as the first input and produce, in the final output, the low-level machine instructions to execute on specific hardware. In classical computing, a good example of such a toolchain is one for synthesizing hardware from a high-level language[2,3]. Such a toolchain offers opportunities for portability and for optimization, but may also introduce inefficiency at every level, both in computation time and storage use. In classical computers, hardware resources are often ample, so the benefits in reducing programmer time and the possibilities for automated optimization far outweigh potential inefficiencies in the resulting product.

In contrast to current classical computers, despite progress on quantum algorithms[4–18], quantum computers are in their infancy. In some ways, their state of the art is analogous to 1950s era classical computers. Current quantum computers can support very few bits, making every quantum bit count. In addition, quantum states are fragile, and longer execution times result in more errors, making every instruction count. Because quantum computing (QC) is so sensitive to storage and execution efficiency, the temptation is to revert to explicit, low-level programming approaches, akin to the assembly code used in early classical computers.

Although a software toolchain will be imperative once quantum computers are larger, even before such hardware becomes available, QC software and hardware development will depend on several of the features toolchains provide. First, QC algorithms are complex to develop and difficult to debug. Toolchains can provide abstractions to support algorithm development as well as program correctness through static compiler analysis and dynamic assertion checks. Second, the requirements of even the smallest interesting instance of any QC algorithm far outstrip current QC hardware resources. Toolflows are necessary to estimate and optimize resource needs to guide system design. To be useful, however, toolchains need aggressive space and time optimizations (circuit width and depth) to minimize qubit usage and to lessen the load on quantum error correction.

As shown in Fig. 1, QC tools do have some similarities to classical computation tools. The challenges arise in how to implement them and in what details should be made available to the next higher level. To make quantum programming manageable and quantum systems practical, we must expose the appropriate set of low-level details to upper software layers. The trick is to pick the best ones, those that will allow programmers to express programs while producing software that gets the most computation out of physical machines. Here we will consider the state of the art and the key future challenges for the quantum programming languages that form the first input to the toolchain, the compiler that makes the first transformations, and the runtime system software that performs transformations that can only be performed during execution of the program.

Quantum toolflow challenges

Before introducing quantum computation and the ways in which its characteristics affect its toolchain, it is useful to understand the stack for classical computing and hardware synthesis. In classical computers, software developers programming in high-level languages need to know little about the semiconductor technologies used to fabricate the microprocessor that executes their programs. In fact, they rarely need to know anything below the architecture level (see Fig. 1). The architecture level includes high-level specifications of the operations a processor is capable of and methods for interacting with memory or input/output devices. For example, programmers need to know whether they are programming for a cell phone, graphics processor, or general-purpose processor, and each processor's differing attributes result in different ways of managing memory and computation. A compiler transforms high-level language to assembly language for a particular architecture.

Hardware synthesis is an alternative design methodology. If a system or a portion of a system design has area, power or cost constraints that cannot tolerate the inefficiencies introduced by execution solely on general-purpose processor hardware, a developer can express the desired

[1]Department of Computer Science, University of Chicago, Chicago, USA. [2]Department of Computer Science, Princeton University, Princeton, USA.

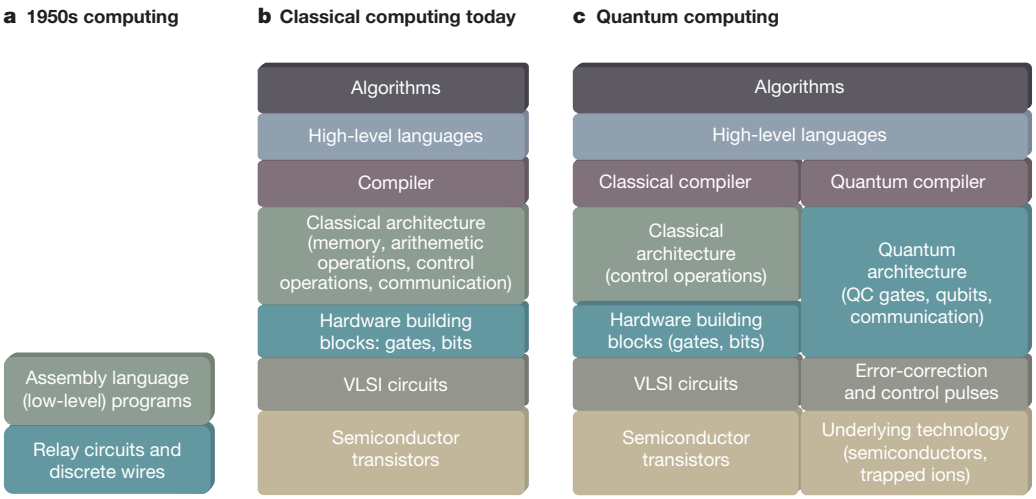


Figure 1 | Design tool flows and abstraction stacks. a, Toolchain for computing in the 1950s, when programmers used assembly directly, which controlled relay circuits and wires. **b**, Over time, improved hardware and software allowed algorithms to be expressed in high-level languages with the compiler translating and optimizing to machine instructions. VLSI, very large-scale integration. **c**, QC is split into classical control structures and quantum instructions. Whereas the classical side can take advantage of the classical tool chain, quantum operations are treated differently owing to the physical constraints imposed by quantum computers.

computation, such as a simple classical function (for example, a fast Fourier transform), so that it can be automatically synthesized into more customized hardware. In these cases, the developer uses a programming language called a hardware description language. The hardware description language program is then analysed and heavily optimized to produce a low-level hardware design that can be fabricated as an application-specific integrated chip or downloaded onto a field-programmable gate array. However, the design languages used for hardware synthesis are much more detailed than high-level languages.

QC shares characteristics with both design models. As in classical computing, QC algorithm designers work at a very high level. As described below, quantum computers will depend on classical computation, so high-level programming languages are an attractive target. At the same time, however, quantum computers have strict resource constraints that, for the foreseeable future, require the compilation of quantum code to a level of specification similar to the circuit level. As we navigate this unique computation model, we reflect on what characteristics are shared with existing computation models so that we can draw upon previous techniques whenever possible.

Before exploring the layers of the software toolchain in more detail (Fig. 1), it is useful to have an execution model in mind. Almost all models of quantum computation require classical control. This is because it would be difficult to achieve reliable measurement and fault-tolerant computation with error-prone quantum devices if there were no way to sequence operations reliably and to make error-correction decisions. Consequently, all known software toolchains assume a 'quantum co-processor' model. That is, a classical central processing unit controls a quantum processing unit and orchestrates its execution, much as today a classical microprocessor orchestrates and interacts with the graphics processing unit residing on the graphics co-processor card. Although actual implementations may differ, the software writer can think of a single microprocessor sending instructions to the quantum co-processor every cycle. Unlike a graphics processing unit, which can perform sophisticated operations with no microprocessor intervention, the quantum co-process does not fetch its own instructions. At all times, the quantum unit is controlled by the microprocessor. A good recent discussion of this model can be found in ref. 19 and in a series of quantum architecture studies since 2005[20–29]. Although several variations exist, most available tools support the QASM quantum assembly language[20], a common language among quantum software toolchains, using it to specify instructions for the quantum co-processor (the Scaffold QC tools are available at https://github.com/epiqc/ScaffCC). Unlike classical assembly languages, the QASM language is specified at the gate level rather than at the arithmetic operation level.

Table 1 shows some differences between QC and classical computing that affect the toolchain. There are three main differences that are relevant to toolchain design and implementation. First, although algorithms may be written for varying problem sizes, in order to minimize space and time requirements, the final QC compilation step targets a specific problem and hardware configuration. All inputs related to the size of the problem must be provided. In addition, to allow for further optimization, the code may be recompiled for each data value. This allows the compiler maximal knowledge of the complete execution of the program. Second, quantum computers are very unreliable, so error-correction techniques are a major algorithm design constraint. Third, there is a chasm between the resources available today and the programs being designed for future quantum computers. Commercially available hardware has so few qubits that the problem sizes supported do not expose bugs in the code. In addition, useful quantum computation is only useful insofar as it is computationally unfeasible on classical computers, which makes useful quantum programs impossible to simulate. That is to say, if quantum programs could be simulated in reasonable time on a classical computer, quantum computation would not be necessary. Finally, we note that, unlike most classical computers, most QC machines support substantial simultaneous parallel operations on potentially all quantum bits.

Having established the execution model, we explore the causes of the differences between classical computing and QC. Some of them are fundamental to QC itself, and others are traditions of the field that could be adjusted over time. We divide these issues into those that 'flow upwards' from the physical device layer and those that 'flow downwards' from software and algorithms.

Exposing physical properties

As specific examples of the many broader ways in which underlying technology attributes affect higher-level software abstractions and toolchains, there are three physical properties that affect design decisions made in higher levels of the toolchain: two that are common to all technologies (no cloning and fragility of state) and one that is dependent on particular technology (parallelism).

All QC systems must adhere to the 'no cloning' theorem, which says that qubit state cannot be copied. Therefore, any QC software with qubit copies is invalid. Language abstractions can encourage or enforce this, compilers can check that it is achieved, and any techniques to implement module hardware or architecture primitives must follow this.

Table 1 | Differences between quantum and classical software tools

Quantum software	Classical software
Compiles from algorithms to gate-level instructions	Compiles from algorithms to machine instructions
Some inputs of problems known at compilation time	Inputs of problems unknown at compilation time
Very limited reliability through error correction	Reliability generally assumed
Can run only small programs for debugging	Runs all programs for debugging
Vast parallelism	Limited parallelism

Although software in both domains shares many similarities, some key differences drive many of the challenges in developing quantum software tools.

All proposed QC systems are suffer from fragility of state. QC hardware performs computations based on manipulating 'state' associated with quantum bits or qubits. Although technologies vary, common operations include applying an operation to a single qubit where the qubit resides (usually in the form of electromagnetic pulses) or moving qubits (or their state) around the system because multi-qubit gates require the qubits to be physically adjacent. Qubit states are fragile and susceptible to decohering (that is, collapsing their complex state into a simple 1 or 0) caused either by the passage of (very little) time or logical operations. Particular technologies have different characteristics in terms of holding state and performing operations, ultimately influencing the execution of a quantum algorithm.

The combination of state fragility and the no cloning theorem makes error correction much more important in QC than in classical computing. A copy of the state cannot be kept, to restore a qubit's value easily. Thus, to design QC systems properly, characteristics such as the timing of different operations, likely error patterns, and quantum error correction code coverage are all hardware-level attributes that must be shared upwards through all the layers of software.

Although errors make communication and holding state costly, the vast parallelism in computing resources help to mitigate this problem. In classical computing, computing resources are dominated by the storage available. Data are communicated from various storage areas to the microprocessor to perform calculations. In many proposed QC hardware implementations[30–36], on the other hand, communication is necessary only to place interacting qubits next to each other; they are not moved to computing resources. This means that the amount of parallelism available is potentially the same as the number of qubits. In addition, owing to the low-level nature of quantum instructions, this parallelism is available at each cycle without extra resources such as classical computation constructs like threads and nodes.

Finally, just as in classical computation, the low-level physical design will have an impact on characteristics that are relevant to the architecture level (such as the relative speed of computing versus memory versus qubit motion, or such as the size limits on different hardware structures) that are important for compiler optimizations. In today's classical computers, we often have methods for 'virtualizing' hardware resources—such as abstracting how much physical memory storage is available in a particular hardware implementation. Given the tight constraints on QC implementations, such virtualization cannot yet be accommodated. Below, we discuss how low-level physical information can best inform higher-level compilation while maintaining abstraction.

Supporting algorithm analysis and debugging

As QC hardware prototypes become more useful, QC algorithms that could previously only be analysed theoretically should now be more pragmatically mappable onto real hardware. QC algorithms are fundamentally different from classical algorithms because a single qubit stores not just a 1 or 0, but the superposition of both states. Furthermore, n qubits can form a superposition of 2^n states (through entanglement). As a result of this exponentially complex state, only the smallest of quantum programs can be executed on a physical machine or even simulated on a classical machine. QC software toolchains do not perform full system simulation. Instead, they perform tasks such as estimating the time and number of quantum bits larger programs will require. With these resource estimates as a metric, the quantum toolchain enables the development and evaluation of algorithms, compiler optimizations, error correction codes, layout schemes for quantum data, communication methods and architectural designs.

In addition to the nuts and bolts requirements of translating an algorithm to an orchestrated execution (for example, a QC assembly code program) the software toolflow can also carry additional information downwards to assist with analysis or debugging. For example, by annotating programs with information about which qubits are believed to be entangled (operations on one can affect state in the other) with each other, lower levels of the toolchain can check that manipulations applied to entangled qubits are done so legally and correctly. Likewise, one can use algorithm-level information to guide qubit 'garbage collection'. That is, when a qubit is no longer being used for one purpose, it can be reclaimed and reused. Languages can provide ways for the programmer to indicate that he/she has finished using a qubit, and the compiler can determine when qubits may be reused without perturbing any entangled bits.

Software toolchains can also assist with hardware design decisions. For example, by analysing different algorithms one can estimate the operation-level parallelism that might be useful to them, and use this to guide how the hardware or quantum error correction code is implemented. Having established the core differences between classical and quantum software, as well as the reasons for them, we examine how these issues affect QC programming languages, compilers and debugging techniques.

Programming languages

Many people hope that quantum programming languages will aid in the discovery of new quantum algorithms. At this point, however, most QC algorithm designers work at a more mathematical level and do not consider the programming language to be the limiting factor in designing new algorithms.

Our experience is that implementing a quantum algorithm—that is, transforming them from a mathematical description in a scientific paper into an executable implementation—can be complex and error-prone. It is in this endeavour that quantum programming languages, compilers and debuggers can make a big difference. In fact, this is analogous to classical computing, where software tools allow programmers to create, debug, and verify complex implementations that are based upon combining and adapting a relatively small set of known classical algorithms. As with classical computing, better abstractions and software infrastructure can accelerate the development of practical quantum programs. For example, quantum algorithms for ground-state estimation and variational solvers have been developed, but substantial work is involved in developing detailed software implementations that solve real problems. The Quarc group at Microsoft Research used their language (Liquid) and software to develop an algorithm to optimize nitrogen fixation using these kernels, improving the efficiency of the implementation by five orders of magnitude through compiler optimization and error tolerance[37].

As with classical programming languages, quantum programming languages fall into two categories: functional and imperative. Purely functional languages do not allow variables to be directly modified—rather, a new value for a variable requires a new name for that variable. Most functional languages encourage a more abstract or mathematical implementation of algorithms which tends to be more compact and some argue is less error-prone. Imperative languages, on the other hand, allow direct modification of variables and tend to specify each step of a computation sequentially and in great detail. Imperative languages tend to produce more efficient programs, but are generally more complex for programmers to think in.

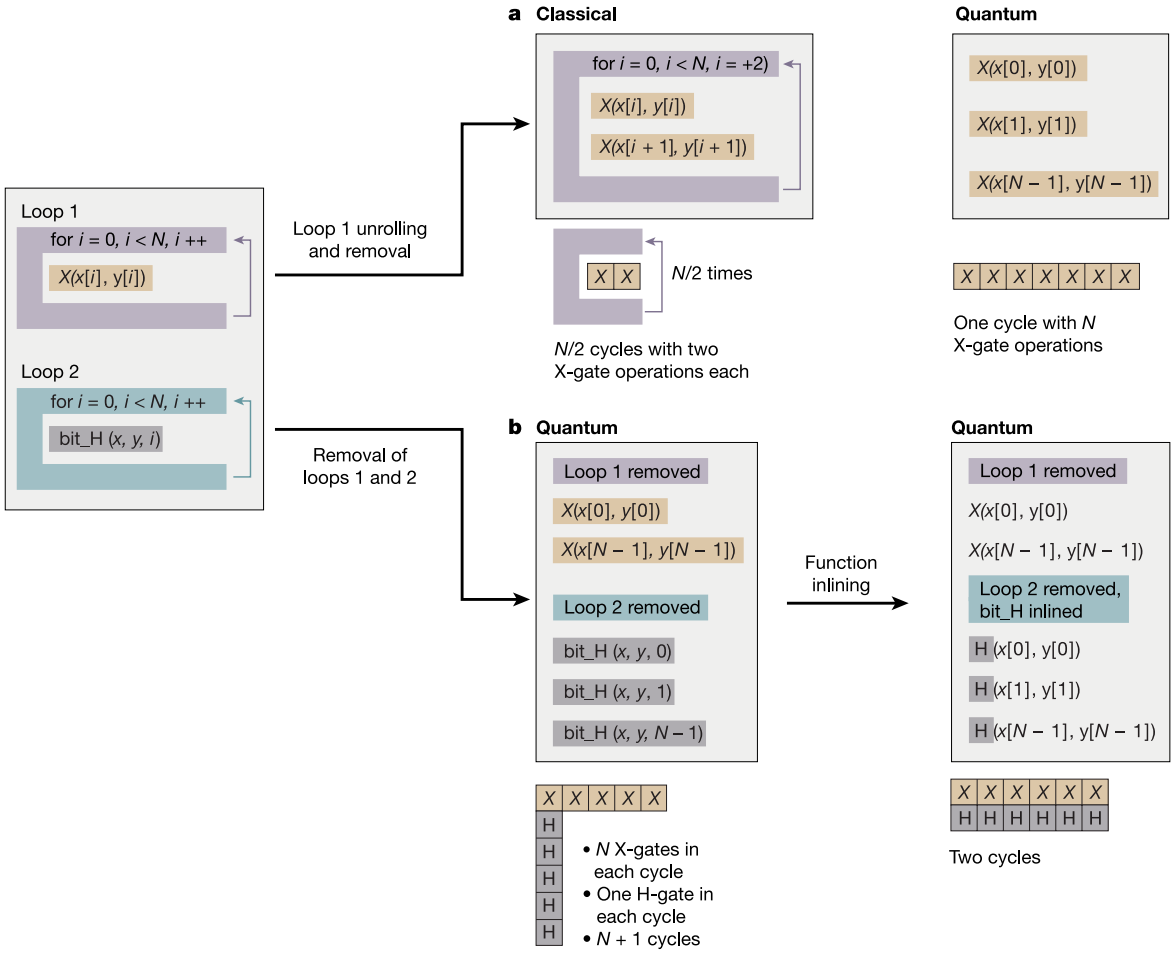
One of the earliest quantum programming languages was qcl[19,38], an imperative language based on C that expressed quantum bits as vectors of data and quantum operations as matrix operations on those vectors. More recently, the IARPA Quantum Computer Science research program resulted in three languages and software toolchains: two functional

(Quipper[39] and Quafl[40]) and one imperative (Scaffold[41]). Quipper and Quafl are embedded languages, which means that they piggyback on top of existing functional languages, greatly simplifying their implementation. We found that the development of the compilers for Quipper and Quafl was initially much faster and agile than our compiler for Scaffold, but that our toolchain is better equipped to handle large quantum programs or a greater amount of automated program analysis and optimization. Specifically, Scaffold's LLVM infrastructure[42] and scalable optimization heuristics allow compilation of programs involving a larger number of qubits and gate operations. Microsoft Research's Liquid[43,44] is another good functional quantum programming language. There are also some lower-level languages that focus on generating control pulses for physical machines[45,46] and efficient representation of both quantum operations and classical control[47].

We observe several aspects that need further development in quantum programming languages. First, there is a need for languages that allow programmers to specify easily what a program ought to do at a high level as opposed to existing operation-oriented languages that specify only

BOX 1
Loop unrolling and inlining

X is a single X-gate operation; H is a single H-gate operation. In the classical case of Box 1 figure a, the compiler does not know the amount of parallel hardware. The ability of the hardware to detect and schedule the parallel operations is crucial. In the QC case of Box 1 figure a, parallelism exists up to the number of qubits in the machine. The number of iterations is a constant, unrolling loop to the point of removal. All tasks are scheduled in a single parallel step. In the QC case of Box 1 figure b, loop 1 is removed as described in Box 1 figure a. Loop 2 is also removed as described in Box 1 figure a, but a function is called that prevents the detection of parallel operations. Once bit_H gets inlined (that is, replaced by the contents of the function), the compiler can recognize possible parallelizations, allowing for full parallelism.



Box 1 Figure | Loop unrolling and function inlining. a, Loop unrolling and removal of loop 1. Loop 1 performs the X operation. In a classical microprocessor: for generality, classical computation supports different values of *N*, so the number of iterations is unknown. The loop is unrolled (the loop body is copied multiple times, reducing the number of iterations accordingly) to provide more instructions to reorder. Hardware detects parallelism at runtime, possibly resulting in two-way parallelism. Therefore, *N* operations require *N*/2 cycles. In a quantum computer: parallel hardware supports abundant hardware, up to the number of qubits in the machine. Both the number of cycles is known and ample parallel hardware is available. Therefore, *N* operations require 1 cycle. **b**, Loop removal of loops 1 and 2, and function inlining of loop 2. Loop 2 performs the H operation. In a quantum computer: a function call prevents detection of parallel operations. If the procedure call is inlined (the call is replaced by function arguments propagated through the call), then loop unrolling and removal becomes useful, allowing full parallelism.

the step-by-step mechanics of what to do. We need such specification languages so that we can specify what it takes for a program to be correct and check the implementation of the program against that specification. Such verification support is crucial because QC developers will always implement quantum programs for the next-generation quantum machine that cannot be tested on current machines. Although recent experimental progress has been impressive[48,49], fully capable quantum computers large enough to test most programs do not yet exist, so all testing must be done through classical simulation. Once we have small quantum computers, we will still have difficulty because each new qubit in future machine sizes will increase computational power exponentially.

Second, we need to specify the desired precision and error tolerance in quantum programming languages. This need is similar to that in approximate computing paradigms on classical machines. For two reasons, quantum algorithms are typically designed to be inherently error tolerant. First, quantum machines have high rates of error. Second, some quantum operations (such as an arbitrary rotation) can only be approximated on real machines. Consequently, many quantum algorithms have easy classical tests for correct output (for example, multiplying the results in Shor's algorithm to check the proper factorization of the product of two primes) and may require multiple executions before producing correct output. A problem that is easy to check is often referred to as a 'promise problem'. For a quantum algorithm to find a solution to a promise problem in a small, constant number of executions, the probability of correct output must be reasonably high (for example, 0.5). Additionally, some algorithms (for example, ground-state estimation in quantum chemistry) require a numerical precision that scales with the complexity of the physical system modelled.

The current state of the art in quantum languages and software tools makes conservative assumptions about the error tolerance and precision required for a quantum program. These assumptions are often made by the programmer and hard-coded into compiler command-line options or code declarations. Error correction codes and approximation algorithms are then employed with a wide margin of safety, often with an overestimate of the number of qubits and the number of quantum operations a program employs.

To target near-term machines, we expect quantum software to adopt a much more optimized approach, evolving as have classical approximate computing tools. We expect that error probabilities and precision requirements will be specified for each quantum program output, perhaps in a manner similar to classical languages for approximate computation[50]. Compilers and runtime systems will need to track the data flows into these outputs and the computation will need to be adjusted to meet the output requirements.

Compilers

Compilers are software programs that translate an algorithm written in a high-level language into machine language instructions that can be executed step by step on particular hardware. Along the way, compilers perform code analysis and optimizations to identify more efficient mappings onto the targeted hardware. Giving the compiler more information—either 'from above' about the algorithm and its likely data inputs, or 'from below' about the hardware details of the target platform—allows the compiler to tailor its chosen translation to the expected program input and target. Hardware synthesis uses similar information to tailor hardware designs to expected inputs. We now discuss static and dynamic compilation (static compilation occurs before execution, whereas dynamic compilation generates code while the program is running) and their relationship to the unique properties of quantum computation.

Static compilation

Compared to compilation for classical computer systems, compilation for a quantum system has both advantages and disadvantages. The primary advantage of compilation for QC systems, compared to the classical analogue, is that QC programming models typically offer much more detailed information about the program and the data; this extra information allows QC compilers to perform optimizations not available to classical compilers. In addition, QC hardware provides much more parallelism than classical hardware—some designs propose that the same operation can be applied to every qubit at the same time. This information and physical capability is critical because the primary disadvantage QC systems face is such severe resource constraints (for example, low gate and storage counts and high quantum error correction code requirements) that QC compilers must perform much more aggressive optimizations to 'make things fit'. These aggressive optimizations themselves require substantial computation resources, hindering the compiler's ability to compile effectively and scalably for large applications or large data input sizes.

In classical machines, programs favour generality; they are usually written for a range of possible data inputs or underlying hardware. As a result, compilers have a general sense of what operations will occur, but do not know what inputs they will see, and conditional branches (if-then-else) mean that compilers will not fully know in what order operations may occur. Although modern classical compilers offer impressive program analysis capabilities, these unknowns do place limits on the degree of useful optimization that can occur. For example, compiling matrix multiplication will typically provide for general iterative constructs, but would not typically tailor the compiled program to a particular matrix size, and certainly not to the particular data values stored in the matrix.

In contrast, although high-level QC code may be designed for a wide range of inputs, a particular compilation of a quantum program requires knowledge of problem sizes. Furthermore, because even long compilation times seem inexpensive compared to the potential savings compiler optimizations can offer in quantum resources, it may be beneficial to recompile and tailor the quantum code before execution for every instance of a program. This customization allows the compiled program to make the best use of every qubit and other constrained QC resources. Interestingly, this expectation of customization means that QC compilation can be much more difficult than its classical counterpart. More precisely, out of a desire for efficient hardware tailoring, QC compilers end up considering a very large (exponential) number of possible ways to organize instructions for a quantum machine.

The classical computational requirements of QC compiler optimizations are challenging for current machines in terms of computing capability and memory capacity. The desire for specificity in compilation leads to substantial scalability challenges. For example, many aspects of classical compilation require computation or memory that increases linearly with the number of (static) lines of code, or scale modestly for practical scenarios. On the other hand, if QC compilation is tailored to a particular data input, then much of its scaling is relative to dynamic instruction counts rather than static (owing to loop unrolling and function inlining, as discussed below). As a result, analyses such as identifying parallelism or optimally scheduling operations can quickly outgrow the capabilities of even large-scale classical machines.

Box 1 illustrates a concrete example of two common compiler optimizations, loop unrolling and function inlining[51], and how the QC setting allows greater optimizations than occur in classical computing.

In loop unrolling, the compiler converts a loop that performed one task in n iterations to a loop that performs x tasks in n/x iterations. This allows the compiler to consider two iterations of the loop for optimizations rather than only one, providing more opportunities for parallelism and instruction reordering. Loop unrolling to the point of removal is dependent on knowing the exact number of iterations, something unique to quantum computation owing to its perfect knowledge of the number of iterations and the vast parallelism available.

In function inlining[51], a function call is replaced by the function call's body (its contents), with function arguments propagated through the body. Function inlining increases the number of static instructions in the program because it replicates the body at the location of each call. However, it allows the compiler to consider the body of the function in its optimization decisions[52,53]. Both loop unrolling to the point of removal and function inlining are necessary to fully optimize this code.

BOX 2

The state of the art in quantum toolchains

The layers of QC toolchains are being gradually filled in, but we also note that the right abstractions for passing information between the layers are still evolving. There are also gaps and weaknesses in the layers; this is especially true in the area of verification, where much more than basic simulators is needed.

Languages and compilers

qcl[38]—One of the earliest languages, good for teaching and toy problems. Open source.

Quipper[39]—Functional language implemented in another functional language; one of the easiest to modify. Open source.

Quafl[40]—Another functional language. Not currently available.

Scaffold[29]—C-like language with an LLVM-based[42] industrial-strength compiler. Open source.

Liquid[43]—Functional language with a highly optimized compiler written in F#. Closed source. Binaries available.

Project Q[45]—Language and Python-based system targeted at compiling smaller programs for prototype machines. Open source.

ARTIQ[46]—Another language for prototype (trapped-ion) machines based on a long line of experimental work at NIST[67,68]. Open source.

QUIL[47]—A new language with an emphasis on the classical-quantum interface. Open source.

Dynamic execution

Quantum Rotations[28]—Dynamic compiler specialized for generating rotation operations across arbitrary angles. Open source.

ARTIQ[46]—Real-time control code which can be loaded onto field-programmable gate arrays. Open source.

Verification (simulators)

qHipster[61]—The most scalable simulator for noise-free computations; runs on supercomputer resources. Future open source release as the Intel Quantum Simulator.

Project Q[45]—Highly-optimized simulator for noise-free simulation. Open source.

Liquid[43]—Efficiently supports both noise-free and error simulation; leverages compiler optimizations to speed up simulation; also has verified reversible circuit synthesis[69]. Closed source. Binary available.

QX[62]—Rapidly improving simulator with an easy-to-use development environment and machine simulation features. Open source.

On the other hand, loop unrolling and function inlining are not always beneficial, and therefore the QC compiler must use them judiciously. Consider several nested loops (loops within loops) or function calls (function calls within functions). If we unroll and remove a loop within a loop within a loop (or inline a function within an inlined function), the number of instructions quickly becomes large: exponential in the number of nested loops or inlined functions! Therefore, loop unrolling is typically performed for a small number of iterations. In addition, the vast parallelism available may not be infinite; technological limitations on control may not allow arbitrary parallelism. For example, it might restrict certain regions to perform the same gate on any number of bits.

The compiler's decisions regarding when to unroll and inline, as well as optimizing the resulting code, are very time- and memory-intensive at compilation time. We, and other groups, are still working on this. Our work on managing unrolling and inlining offer initial examples of this[29]. More sophisticated approaches will require a model of target machine behaviour and a means of developing a statistical profile of what portions of a program are most important.

Dynamic compilation

Assuming that error correction and control can be largely automated and abstracted into a lower layer of software, the control flow and computational requirements of most quantum computation is then known statically and can be handled by the compiler (subject to scaling limits). Yet there are a few operations that depend on information measured at runtime. We now discuss both generating instructions at runtime (that is, dynamic compilation) and performing operations for which code is available but dynamic information is necessary to know whether to run those instructions (that is, dynamic execution).

Two QC scenarios can benefit from dynamic compilation. First, for an operation called a quantum rotation, the angle of rotation sometimes depends on intermediate measurements of quantum data[28]. The precise rotation instructions must be generated at runtime. Second, rotations are implemented by performing a series of discrete (physically reliable) rotations. For some very precise angles, this can result in very long sequences. In proposed QC systems running at 30 millikelvin, classical computing has been proposed to reside with the quantum qubits, but with low bandwidth connection to classical storage. Therefore, the system would benefit from transmitting a single instruction from storage to the classical processor to generate the rotation sequence at runtime, and the quantum processor would then be sequenced through each step.

Finally, there is the potential of software to control physical machine conditions at runtime. For example, all superconducting qubits studied thus far can be described by an LCJ circuit, consisting of inductors (L), capacitors (C), and Josephson junctions (J). These circuits can be controlled to reduce one of two kinds of qubit errors, amplitude and phase, but not both at the same time[54]. This error tradeoff may be useful for some error correction algorithms. More speculatively, there is some potential that machine learning techniques can be used to adapt qubits to environmental noise[55], if the cost of measuring the noise in order to train the machine to perform the techniques can be made low enough.

Classical co-processing

With the growing importance of hybrid algorithms that use both quantum and classical processing, such as the variational eigensolver[18,56], representing and coordinating these computations is an important task. Some new software tools from Rigetti focus on this task[47]. The challenge is to represent the coordination between quantum and classical processing efficiently across multiple iterations, each of which alternate between the two domains. The classical computation may need to communicate precision requirements to the quantum computation, and the quantum computation may need to communicate noise and error information back to the classical computation.

Furthermore, the interface between classical and quantum computation may involve a slow and low-capacity communication channel. Such is the case in quantum machines built from devices that require cryogenic temperatures[30,33,57]. In such situations, the quantum computation is at the bottom of a dilution refrigerator stack and the classical computation is outside at room temperature. Signals between classical and quantum machines must travel on a limited number of cables through several temperature-controlled layers in the refrigeration stack.

A complicating factor is that some classical computation could be moved inside the cryogenic stack, requiring hardware implemented with cryo-CMOS[58] (conventional silicon at cryogenic temperatures) or superconducting logic[59]. These options would greatly improve communication between portions of the classical computation and quantum computations, but the cryogenic hardware can have severe limitations in terms of power (amount of classical computing allowed) or storage (amount of memory for classical program code and data). The result is that the quantum software stack must navigate several tradeoffs by breaking up

classical computation into room-temperature and cryogenic domains, keeping track of communication between room-temperature classical, cryogenic classical, and cryogenic quantum hardware.

Debugging and assertions

A fundamental challenge for QC is that when a new computer is built and tested, it will have errors. How will we distinguish between hardware and software bugs? Even with a fully functional toolchain, QC's superposition properties mean that any classical simulation of a quantum algorithm will experience exponential scaling in state space or runtime, making simulation intractable. However, progress can still be made.

Programs can be run on problem sizes small enough for classical simulation. These algorithms are useful tests for automated gate compilation. For example, automated compilation of small quantum chemistry problems at Microsoft revealed errors in the circuits reported in the literature[60]. Microsoft and others are developing simulators of up to 40–50 qubits of noise-free computation (simulating noise requires more resources)[43,45,61,62].

At large problem sizes, some QC algorithms have output that can be efficiently simulated owing to the simplified nature of the circuits. For example, classical arithmetic on qubits can be simulated efficiently. More interestingly, simulations of all Clifford circuits scale only quadratically, not exponentially, in the size of the quantum circuit. These models are ideal for testing the performance of quantum circuits in the presence of noise and provide an idea of what noise levels still allow for a quantum advantage. Recent work[63] has shown that a few non-Clifford gates can be added without losing the ability to simulate the system tractably. While such approaches cannot simulate full programs, they do allow toolchain writers to compare simulated and ideal outputs to test whether automated tools are correctly synthesizing circuits.

An alternative approach is to focus on specific properties of the algorithm. For example, one can use mathematical constructs and formalisms in the programming language, and then allow the compiler to infer and annotate subsequent levels of the toolchain with attributes of the state that are known at different points of execution[64]. Such programmer annotations can be checked by a classical satisfiability modulo theory solver or theorem prover[65]. In addition, type analysis can be used to support QC correctness checks, such as the no-cloning rule[66], entanglement, qubit uncomputation or re-use, and probability of error.

Furthermore, there will be challenges for building the classical control resources for a QC. For technologies that operate at high clock rates (for example, superconductors at gigahertz speeds), it will be challenging to provide parallel control of qubits at speed. These challenges could be exacerbated by the cryogenic temperatures for some technologies (for example, superconductors again), where speed is high but energy and storage constraints are tight.

Outlook

With 50–100 qubit prototypes anticipated in the near future, QC is on the cusp of a revolution. This revolution, however, will depend heavily on the software toolchains (see Box 2) that will bridge the gap between algorithms and physical machines. Efficient compilers and resource estimation (runtime, qubit counts) will allow different QC hardware design tradeoffs to be explored thoroughly. These software toolchains will also accelerate the success of QC, by allowing QC algorithms to be compiled into shorter-running programs that require fewer qubits and operations. Given the intrinsic unreliability of quantum operations, such optimizations will make an important difference when QC is ready to be deployed commercially.

In many ways, the current constraints of QC toolchains resemble the constraints of computing in the 1950s, but now we can benefit from the knowledge gained from decades of classical toolchain development. As such, although QC compilation remains quite similar to traditional compilation at present, we expect an increasing divergence as more QC and compilation experts begin to focus on this fascinating and challenging problem.

Received 1 March; accepted 4 June 2017.

1. Moore, G. E. Progress in digital integrated electronics. *IEEE Solid-State Circ. Soc. News.* **20**(3), 11–13 (1975; reprinted 2006); available at http://ieeexplore.ieee.org/document/4804410/
2. De Micheli, G. Hardware synthesis from C/C++ models. In *Proc. Conf. on Design Automation and Test in Europe (DATE '99)* 80 (ACM, 1999).
3. Deschamps, J.-P., Valderrama, E & Terés, L. *Design Methods* 171–177 (Springer, 2017).
4. Shor, P. W. Algorithms for quantum computation: discrete logarithms and factoring. In *Proc. 35th Ann. Symp. on Foundations of Computer Science (FOCS '94)* 124–134 (IEEE, 1994).
5. Mosca, M. in *Encyclopedia of Complexity and Systems Science* (ed. Meyers, R. A.) 7088–7118 (Springer, 2009).
6. Ambainis, A., Childs, A. M., Reichardt, B. W., Spalek, R. & Zhang, S. Any AND-OR formula of size N can be evaluated in time $N^{1/2+O(1)}$ on a quantum computer. *SIAM J. Comput.* **39**(6), 2513–2530 (2010).
7. Childs, A. M. *et al.* Exponential algorithmic speedup by a quantum walk. In *Proc. 35th Ann. Symp. on Theory of Computing (STOC '03)* 59–68 (ACM, 2003).
8. Hallgren, S. Fast quantum algorithms for computing the unit group and class group of a number field. In *Proc. 37th Ann. Symp. on Theory of Computing (STOC '05)* 468–474 (ACM, 2005).
9. Grover, L. K. A fast quantum mechanical algorithm for database search. In *Proc. 28th Ann. Symp. on Theory of Computing (STOC '96)* 212–219 (ACM, 1996).
10. Whitfield, J. D. *et al.* Simulation of electronic structure Hamiltonians using quantum computers. *Mol. Phys.* **109,** 735–750 (2010).
11. National Institute of Standards and Technology *FIPS PUB 180–4: Secure Hash Standard (SHS)* http://csrc.nist.gov/publications/fips/fips180-4/fips180-4.pdf (US Department of Commerce, 2012).
12. Magniez, F., Santha, M. & Szegedy, M. Quantum algorithms for the triangle problem. In *Proc. 16th Ann. Symp. on Discrete Algorithms (SODA '05)* 1109–1117 (ACM-SIAM, 2005).
13. Aspuru-Guzik, A., Dutoi, A. D., Love, P. J. & Head-Gordon, M. Simulated quantum computation of molecular energies. *Science* **309,** 1704–1707 (2005).
14. Jordan, S. P., Lee, K. S. M. & Preskill, J. Quantum algorithms for quantum field theories. *Science* **336,** 1130–1133 (2012).
15. McClean, J. R., Babbush, R., Love, P. L. & Aspuru-Guzik, A. Exploiting locality in quantum computation for quantum chemistry. *J. Phys. Chem. Lett.* **5,** 4368–4380 (2014).
16. Reiher, M., Wiebe, N., Svore, K. M., Wecker, D. & Troyer, M. Elucidating reaction mechanisms on quantum computers. Preprint at https://arxiv.org/abs/1605.03590 (2016).
17. Peruzzo, A. *et al.* A variational eigenvalue solver on a photonic quantum processor. *Nat. Commun.* **5,** 4213 (2014).
18. O'Malley, P. J. J. *et al.* Scalable quantum simulation of molecular energies. *Phys. Rev. X* **6,** 031007 (2016).
This paper is a good example of the emerging importance of classical-quantum co-processing.
19. Valiron, B. *et al.* Programming the quantum future. *Commun. ACM* **58,** 52–61 (2015).
This paper offers another perspective on quantum programming language design issues.
20. Metodi, T. S., Thaker, D. D., Cross, A. W., Chong, F. T. & Chuang, I. L. A quantum logic array microarchitecture:scalable quantum data movement and computation. In *Proc. 38th Ann. Int. Symp. on Microarchitecture (MICRO)* 305–318 (ACM/IEEE Computer Society, 2005).
21. Thaker, D. D., Metodi, T. S., Cross, A. W., Chuang, I. L. & Chong, F. T. Quantum memory hierarchies: efficient designs to match available parallelism in quantum computing. In *Proc. 33rd Ann. Int. Symp. on Computer Architecture (ISCA)* 378–390 (ACM/IEEE Computer Society, 2006).
22. Balensiefer, S., Kregor-Stickles, L. & Oskin, M. An evaluation framework and instruction set architecture for ion-trap based quantum micro-architectures. *SIGARCH Comput. Archit. News* **33,** 186–196 (2005).
23. Schuchman, E. & Vijaykumar, T. N. A program transformation and architecture support for quantum uncomputation. *SIGARCH Comput. Archit. News* **34,** 252–263 (2006).
24. Isailovic, N., Whitney, M., Patel, Y. & Kubiatowicz, J. Running a quantum circuit at the speed of data. In *Proc. 35th Ann. Int. Symp. on Computer Architecture (ISCA).* 177–188 (2008).
25. Whitney, M. G., Isailovic, N., Patel, Y. & Kubiatowicz, J. A fault tolerant, area efficient architecture for Shor's factoring algorithm. In *Proc. 36th Ann. Int. Symp. on Computer Architecture (ISCA)* 383–394 (2009).
26. Van Meter, R. & Horsman, C. A blueprint for building a quantum computer. *Commun. ACM* **56,** 84–93 (2013).
27. Metodi, T. S., Faruque, A. I. & Chong, F. T. *Quantum Computing for Computer Architects 2nd edn Synthesis Lectures on Computer Architecture* (Morgan & Claypool, 2011).
28. Kudrow, D. *et al.* Quantum rotations: a case study in static and dynamic machine-code generation for quantum computers. In *Proc.40th Ann. Int. Symp. on Computer Architecture (ISCA)* 166–176 (ACM, 2013).
29. Heckey, J. *et al.* Compiler management of communication and parallelism for quantum computation. In *Proc. 20th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* 445–456 (ACM, 2015).

This paper describes the use of a software toolchain to explore architectural designs and optimizations.
30. Devitt, S. J. Performing quantum computing experiments in the cloud. *Phys. Rev. A* **94,** 032329 (2016).
31. Debnath, S. *et al.* Demonstration of a small programmable quantum computer with atomic qubits. *Nature* **536,** 63–66 (2016).
32. Linke, N. M. *et al.* Fault-tolerant quantum error detection. Preprint at https://arxiv.org/abs/1611.06946 (2016).
33. Kelly, J. *et al.* State preservation by repetitive error detection in a superconducting quantum circuit. *Nature* **519,** 66–69 (2015).
34. Lekitsch, B. *et al.* Blueprint for a microwave trapped-ion quantum computer. Preprint at https://arxiv.org/abs/1508.00420 (2015).
35. Fowler, A. G. *et al.* Surface codes: towards practical large-scale quantum computation. *Phys. Rev. A* **86,** 032324 (2012).
36. Monroe, C. *et al.* Large-scale modular quantum-computer architecture with atomic memory and photonic interconnects. *Phys. Rev. A* **89,** 022317 (2014).
37. Hastings, M. B., Wecker, D., Bauer, B. & Troyer, M. Improving quantum algorithms for quantum chemistry. Preprint at https://arxiv.org/abs/1403.1539 (2014).
This paper describes a software toolchain that improves the efficiency 100,000-fold in their quantum chemistry application.
38. Omer, B. *A Procedural Formalism for Quantum Computing: Qcl.* Master's thesis http://tph.tuwien.ac.at/∼oemer/doc/qcldoc.pdf (Technical Physics, TU Vienna, 1998).
39. Green, A. S. *et al.* Quipper: A scalable quantum programming language. In *Proc. 34th SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '13)* 333–342 (ACM, 2013).
This paper describes a quantum programming language incorporating some of the best design practices of functional languages.
40. Lapets, A. *et al.* Quafl: A typed dsl for quantum programming. In *Proc. 1st Ann. Workshop on Functional Programming Concepts in Domain-specific Languages (FPCDSL '13)* 19–26 (ACM, 2013).
41. JavadiAbhari, A. *et al.* ScaffCC: A framework for compilation and analysis of quantum computing programs. In *Proc. 11th ACM Conf. on Computing Frontiers* 1 (ACM, 2014).
42. Lattner, C. & Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. Int. Symp. on Code Generation and Optimization: Feedback-directed and Runtime Optimization* 75–86 (IEEE Computer Society, 2004).
43. Wecker, D. & Svore, K. *Liquid: A Software Design Architecture And Domain-Specific Language For Quantum Computing*. https://www.microsoft.com/en-us/research/project/language-integrated-quantum-operations-liqui/ (2014).
44. Haner, T., Steiger, D. S., Svore, K. & Troyer, M. A software methodology for compiling quantum programs. Preprint at https://arxiv.org/abs/1604.01401 (2016).
This is a good example of a quantum software stack.
45. Steiger, D. S., Hner, T. & Troyer, M. ProjectQ: an open source software framework for quantum computing. Preprint at https://arxiv.org/abs/1612.08091 (2016).
46. Ion Storage Group. *ARTIQ (Advanced Real-Time Infrastructure for Quantum Physics)* http://m-labs.hk/artiq/index.html (NIST, 2017).
47. Smith, R. S., Curtis, M. J. & Zeng, W. J. A practical quantum instruction set architecture. Preprint at https://arxiv.org/abs/1608.03355 (2016).
48. Figgatt, C. *et al.* Complete 3-qubit grover search on a programmable quantum computer. Preprint at https://arxiv.org/abs/1703.10535 (2017).
49. Castelvecchi, D. IBM's quantum cloud computer goes commercial. *Nature* **543,** 159 (2017).
50. Park, J., Esmaeilzadeh, H., Zhang, X., Naik, M. & Harris, W. Flexjava: Language support for safe and modular approximate programming. In *Proc. 10th Joint Meet. on Foundations of Software Engineering (ESEC/FSE 2015)* 745–757 (ACM, 2015).
51. Aho, A. V., Lam, M. S., Sethi, R. & Ullman, J. D. *Compilers: Principles, Techniques, and Tools* 2nd edn (Addison-Wesley Longman, 2006).
52. Allen, F. E. Interprocedural data flow analysis. In *International Federation for Information Processing (IFIP) Congress* 398–402 (1974).
53. Hall, M. W., Murphy, B. R., Amarasinghe, S. P., Liao, S. W. & Lam., M. S. *Interprocedural Analysis for Parallelization* 61–80 (Springer, 1996).
54. Dempster, J. M., Fu, B., Ferguson, D. G., Schuster, D. I. & Koch, J. Understanding degenerate ground states of a protected quantum circuit in the presence of disorder. *Phys. Rev. B* **90,** 094518 (2014).
55. Mavadia, S., Frey, F., Sastrawan, J., Dona, S. & Biercuk, M. J. Prediction and real-time compensation of qubit decoherence via machine learning. *Nature Commun.* **8,** 14106 (2017).
56. Kandala, A. *et al.* Hardware-efficient quantum optimizer for small molecules and quantum magnets. Preprint at https://arxiv.org/abs/1704.05018 (2017).
57. McKay, D. C., Naik, R., Reinhold, P., Bishop, L. S. & Schuster., D. I. High-contrast qubit interactions using multimode cavity qed. *Phys. Rev. Lett.* **114,** 080501 (2015).
58. Homulle, H. *et al.* A reconfigurable cryogenic platform for the classical control of scalable quantum computers. Preprint at https://arxiv.org/abs/1602.05786 (2016).
59. Likharev, K. K. & Semenov, V. K. Rsfq logic/memory family: a new Josephson-junction technology for sub-terahertz-clock-frequency digital systems. *IEEE Trans. Appl. Supercond.* **1,** 3–28 (1991).
60. Wecker, D., Bauer, B., Clark, B. K., Hastings, M. B. & Troyer, M. Gate-count estimates for performing quantum chemistry on small quantum computers. *Phys. Rev. A* **90,** 022305 (2014).
61. Smelyanskiy, M., Sawaya, N. P. D. & Aspuru-Guzik, A. qHiPSTER: the quantum high performance software testing environment. Preprint at https://arxiv.org/abs/1601.07195 (2016).
62. Khammassi, N. *The QX Simulator* http://www.xpu-project.net/qx/download.html (2017).
63. Bravyi, S. & Gosset, D. Improved classical simulation of quantum circuits dominated by clifford gates. *Phys. Rev. Lett.* **116,** 250501 (2016).
64. Chiw, C., Kindlmann, G., Reppy, J., Samuels, L. & Seltzer, N. Diderot: a parallel DSL for image analysis and visualization. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation* 111–120 (ACM, 2012).
65. Alur, R. *et al.* in *Dependable Software Systems Engineering. NATO Science for Peace and Security Series D: Information and Communication Security* (eds Irlbeck, M., Peled, D. A. & Pretschner, A.) Vol. 40, 1–25 (IOS Press, 2015).
66. Selinger, P. & Valiron, B. in *Foundations of Software Science and Computational Structures* 81–96 (Springer Science & Business Media, 2008).
67. Kielpinski, D., Monroe, C. & Wineland, D. J. Architecture for a large-scale ion-trap quantum computer. *Nature* **417,** 709–711 (2002).
68. Wineland, D. J. *et al.* Experimental primer on the trapped ion quantum computer. *Fortschr. Phys.* **46,** 363–390 (1998).
69. Amy, M., Roetteler, M. & Svore, K. M. Verified Compilation of Space-Efficient Reversible Circuits. In *Proc. Computer Aided Verification: 29th Int. Conf. (CAV 2017) Part II*, 3–21 (Springer International, 2017).

Acknowledgements We thank the many collaborators who have helped shape our thinking over the years: K. Brown, I. Chuang, E. Chi, A. Faruque, A. Harrow, J. Heckey, A. Javadi-Abhari, J. Kubiatowicz, D. Kudrow, T. Metodi, M. Oskin, S. Patil, J. Reppy, D. Schuster, M. Suchara and D. Thaker. This work was funded in part by Los Alamos National Laboratory and the US Department of Defense under subcontract 431682, by NSF PHY grant 1660686, and by a research gift from Intel Corporation.

Author Contributions All three authors contributed equally to the survey and conclusions in this article.

Author Information Reprints and permissions information is available at www.nature.com/reprints. The authors declare no competing financial interests. Readers are welcome to comment on the online version of the paper. Publisher's note: Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations. Correspondence should be addressed to F.T.C. (chong@cs.uchicago.edu).

Reviewer Information *Nature* thanks B. Valiron and the other anonymous reviewer(s) for their contribution to the peer review of this work.