Quantum Operating Systems

Henry Corrigan-Gibbs, David J. Wu, and Dan Boneh
Stanford University

ACM Reference format:
Henry Corrigan-Gibbs, David J. Wu, and Dan Boneh *Stanford University*. 2017. Quantum Operating Systems. In *Proceedings of HotOS '17, Whistler, BC, Canada, May 08-10, 2017,* 6 pages.
https://doi.org/10.1145/3102980.3102993

Abstract. If large-scale quantum computers become commonplace, the operating system will have to provide novel abstractions to capture the power of this bizarre new hardware. In this paper, we consider this and other systems-level issues that quantum computers would raise, and we demonstrate that these machines would offer surprising speed-ups for a number of everyday systems tasks, such as unit testing and CPU scheduling.

1 INTRODUCTION

The last few years have seen tremendous progress towards the construction of non-trivial quantum computers [7, 23, 29]. A number of start-ups are working towards commercializing the technology, NIST is standardizing new "post-quantum" cryptosystems [41], and industry giants, including Google [20] and Microsoft [39], are taking steps today to defend their systems against quantum-enabled adversaries in the future. Large-scale quantum computers may exist in our lifetimes.

The first electronic computers—the Mark I, Colossus, and ENIAC—were expensive, cumbersome, and sluggish machines, primarily useful to government code-breakers and weapons designers. Much like the first electronic computers, the first quantum computers will almost certainly be costly and slow and, as with the first electronic computers, the most obvious applications of quantum computers will be to breaking classical cryptosystems, such as RSA [44], and physical simulation.

Fortunately for us, over time classical computing hardware became cheaper and faster, and modern operating systems provided hardware abstractions, virtual memory, and time-sharing to make computers easier, safer, and faster to use. Computers became a universal tool, with applications far beyond the stodgy realms of cryptanalysis and bomb design. If we are lucky, just as classical computers became smaller, cheaper, and faster over time, so will quantum computers. When they do, programmers will again look to the operating system to make these new and unfamiliar devices easier, safer, and faster for everyday people to use.

HotOS '17, May 08-10, 2017, Whistler, BC, Canada
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5068-6/17/05...$15.00
https://doi.org/10.1145/3102980.3102993

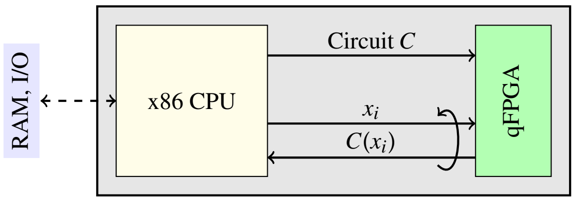

Figure 1: A classical processor connected to a quantum FPGA. The classical processor programs the qFPGA with the a description of a quantum circuit $C : \{0,1\}^n \to \{0,1\}^n$. The CPU can then send inputs $x_i \in \{0,1\}^n$ and receive the output $C(x_i)$ of the quantum circuit applied to x_i.

What sort of computer architecture will our quantum computers employ, and what kind of operating systems will run on these machines? In this paper, we explore the possibility of *quantum operating systems* by asking, and partially answering, a number of questions related to the design of this new type of system:

- What new abstractions could a quantum operating system expose to the programmer?
- How could the power of quantum computers improve the performance of classical software systems?
- What would a distributed system of quantum computers look like? And what new functionality could such a system provide?

This paper is necessarily (and shamelessly) speculative: it is far too early to know exactly what sort of quantum computing hardware will exist in 20, 50, or 100 years. Even so, we can use the existing models of quantum computation to project what will likely be possible if large-scale quantum computers come to exist in the future.

The skeptical reader may—with some cause—view this paper as a collection of negative results: as far as we know, there are not too many exciting things that one can do with a quantum computer that one could not do with a somewhat larger or faster classical machine. Our purpose is to simply take the notion of quantum computing to its logical conclusion and ask: what would happen if we each had one of these magical quantum machines on our desk?

We dedicate the bulk of the paper to sketching three possible architectures for quantum computers, arranged in order from least to most fanciful: we first discuss quantum FPGAs, then quantum x86 machines, and finally, quantum distributed systems. For each, we consider the applications that the machine would have to common systems tasks, such as fuzz testing, CPU scheduling, and parallel programming. We also discuss the systems-level challenges that each architecture would present.

2 BACKGROUND

To set the scene for our discussion, let us first review a few key results from the quantum computing literature. Quantum algorithms, in certain settings, provide surprising speed-ups over classical algorithms. For example, consider the following computational problem:

Problem 1 (Unstructured Search)**.** Given a function $f : \{1,\dots,N\} \to \{0,1\}$ find an $x^* \in \{1,\dots,N\}$ such that $f(x^*) = 1$.

If we treat the function f as a "black box," the best classical algorithm for this problem is just brute-force search: compute $f(1), f(2), f(3)$, and so on, until finding an x^* such that $f(x^*) = 1$. Indeed, any classical algorithm for this problem must invoke f at least $\Omega(N)$ times to succeed with good probability. In contrast, there is a quantum algorithm for solving this problem that invokes f only $O(\sqrt{N})$ times.

INFORMAL THEOREM 2 (GROVER [35]). *There is a quantum algorithm for the Unstructured Search Problem that invokes f at most $O(\sqrt{N})$ times and that succeeds with probability at least* $2/3$.

To explain why such a speed-up is even possible: a classical algorithm that treats f as a black box can only evaluate f at a single input at a time—the classical algorithm works *locally*. In contrast, a quantum algorithm can essentially evaluate a "black-box" function f at a mixture of inputs (a "superposition" of inputs) and can thus learns some *global* information about f with each invocation. After only $O(\sqrt{N})$ invocations of f, the quantum algorithm has explored the entire domain of f, which allows it to recover the target value x^*.

Furthermore, when there are k inputs that cause f to return 1, Grover's algorithm finds one such input with only $O(\sqrt{N/k})$ invocations of f, even if k is not known in advance [19]. As we describe in the following sections, we expect that Grover's algorithm will be one of the most useful tools in the quantum programmer's belt.

An important subtlety of quantum computing is that the only operations that can be performed on a quantum state are those that are *reversible* (or invertible). Thus, to implement Grover's algorithm on a quantum computer, we must first represent the function f in a reversible manner. Even though many classical computations are irreversible (e.g., an AND operation on single-bit inputs), there are standard techniques to make circuits, Turing machines, and even RAM programs reversible [9, 36].

What quantum computers are not. Before diving into the body of our discussion, it is worth emphasizing that *there is no evidence that quantum computers can solve NP-hard problems in polynomial time* [1]. Applying Grover's algorithm to 3SAT, for example, yields a slightly faster-than-brute-force-classical 3SAT algorithm, but both the quantum and classical algorithms still run in exponential time [6]. Furthermore, we know that Grover's algorithm is the best possible quantum algorithm for the Unstructured Search Problem (Problem 1) [10]. As far as we know, it is only for very specific problems, such as integer factorization [44], that quantum computers dramatically outperform their classical counterparts.

3 QUANTUM FPGAS

A plausible architecture for the first generation of general-purpose quantum computers would consist of a classical processor connected over a classical bus to a quantum field programmable gate array, or qFPGA.

Like a conventional FPGA, a qFPGA would essentially be a peripheral device attached to the classical CPU (Figure 1). To use the qFPGA, the programmer would first cook up a quantum circuit C that implements her computation of interest. A quantum circuit is much like a classical Boolean circuit, except that it uses elementary quantum gates (e.g., Toffoli and Hadamard gates [3]) instead of conventional logic gates (AND, OR, NOT).

The programmer would then send the description of a quantum circuit C to the qFPGA over a classical bus. Once programmed with the circuit C, the CPU could send the qFPGA an input x_i. A classical control unit would orchestrate the quantum computation on the qFPGA by applying each quantum gates to the qFPGA's state registers in sequence. The qFPGA would then send the output $C(x_i)$ of the circuit applied to x_i back to the CPU. The inputs x_i and the output $C(x_i)$ would both be classical bitstrings, so the CPU and qFPGA could exchange these values over a classical bus.

As far as we know, we are far far away from being able to construct anything resembling a quantum FPGA. The current generation of quantum computers implement circuits that operate on just a handful of qubits [23, 24, 37], so we are orders of magnitude away from an FPGA-like device that could perform any large-scale quantum computation. But, let us imagine that we had such a device. What could it do?

The obvious application of a qFPGA would be to running Shor's algorithm for factoring integers and computing discrete logarithms [44]. However, by the time qFPGAs are available, the world will long have shifted to quantum-resistant cryptosystems [13]. Instead, we expect that the primary everyday application of these devices would be to run Grover search on programs that could compile down into relatively small circuits. We give two example applications: code fuzzing and password cracking.

3.1 Using qFPGAs

Code fuzzing. The idea of code fuzzing is to run a piece of code on a large number of edge-case inputs to try to identify an input that causes the program to misbehave.

Fuzzing using a qFPGA might be useful for ensuring that tricky optimizations do not introduce correctness bugs [26]. For example, a programmer might have two Boolean circuits for multiplying n-bit integers: a large naïve circuit M_{slow} compiled from native C code, and an optimized hand-designed version M_{fast}. The programmer could construct a circuit $C : \{1,\dots,N\}^2 \to \{0,1\}$ that computes

$$C(x,y) = \begin{cases} 1 & \text{if } M_{\text{slow}}(x,y) \neq M_{\text{fast}}(x,y) \\ 0 & \text{otherwise} \end{cases}.$$

The programmer could ship the qFPGA the program for running Grover search on C.

Using the qFPGA, the programmer could essentially ensure that the two multiplication routines behave identically on 2^{64} possible inputs, at the cost of 2^{32} invocations of M_{slow} and M_{fast}. Furthermore, if there is a severe bug—one that causes the optimized circuit to fail on k out of N inputs, then the qFPGA worker will only need time roughly $\sqrt{N/k}$ to determine the index of the input that caused the optimized circuit to misbehave.

Password cracking. By the time qFPGAs are available, the "quantum apocalypse" will have killed off many of today's quantum-vulnerable cryptographic algorithms. Yet, like the stubborn insects

```
// Type signature of a qthread worker.
typedef int worker(char *args, size_t arglen);

// Initialize a pool of qthreads. All qthreads
// in the pool run the same worker routine.
qpool_t qpool_init(worker *start_routine);

// Feed the specified arguments to a qthread worker
// in the specified pool.
qthread_t qthread_create(qpool_t pool,
                    char *args, size_t arglen);

// Get the ID of a worker that returned > 0.
qthread_t qthread_join(qpool_t pool);

// Get the ID of the qthread that returned the
// largest value.
qthread_t qthread_join_max(qpool_t pool);

// Count how many qthreads returned values > 0.
int qthread_join_count(qpool_t pool);
int qthread_join_count_approx(qpool_t pool);

// Get the sum of the qthread return values.
int qthread_join_sum(qpool_t pool);
int qthread_join_sum_approx(qpool_t pool);
```

Figure 2: The qthreads API.

that survived the extinction of the dinosaurs, we expect the humble human-chosen password to remain a ubiquitous (if universally loathed) feature of computer systems into the quantum era [18].

One nefarious application of qFPGAs, which we have not seen discussed before, would be to the task of password cracking [17, 40]. In a password-cracking attack, an attacker has the image h of a user's password under a cryptographic hash function H and the attacker wants to find a password p such that $h = H(p)$. Since users often pick passwords from a small dictionary of popular phrases D, the attacker can typically recover the user's password by hashing each word in the dictionary until she finds a word $d^* \in D$ such that $h = H(d^*)$.

While a conventional password-cracking attack takes time linear in the size of the dictionary D, a qFPGA-backed password-cracking attack would require only $\sqrt{|D|}$ invocations of the hash function. In concrete terms: there are 95 printable ASCII characters, and there are $95^{10} \approx 2^{66}$ possible ten-character printable ASCII passwords. Even if every user picked her password from this unrealistically high-entropy distribution, an attacker with a qFPGA could invert a password hash with only $\sqrt{2^{66}} = 2^{33}$ invocations of the password-hashing function H. The existence of low-cost qFPGAs would render hashed passwords crackable at low cost.

Modern "memory-hard" password-hashing functions [5, 15, 16, 33, 43] require large classical circuits and thus may be less susceptible to this attack. An interesting open question is whether there are special-purpose better-than-Grover quantum attacks against these hash functions.

4 THE QX86 MACHINE

The qFPGA architecture described in the prior section would be suitable for running quantum computations that are easy to represent as small quantum circuits.

For more ambitious computing tasks, it would be ideal to have an x86 processor connected to a quantum x86 co-processor. The qx86 processor would implement the x86 instruction set, along with a universal quantum instruction [3] that would, for example, apply a quantum operator to the co-processor's eax register. Using these instructions, and a large enough qx86 computer, the quantum assembly programmer could implement any efficient quantum algorithm and could, for example, evaluate an x86 program on a quantum state consisting of a superposition of many program inputs.

Building a qx86 machine seems much harder than building a qFPGA. To support general x86 programs, the machine would have to implement not only the logic necessary to execute x86 instructions, but it would also have to somehow implement a quantum RAM [34]. To give some sense of why this would be difficult to implement: in an n-qubit quantum circuit, the state of the circuit is essentially a mixture (a "superposition") of n-bit strings. A qFPGA might support on the order of $n = 2^{10}$ qubits or so.

In contrast, the state of a qx86 machine would be described by a superposition of M-bit states, such that the entire state of the machine (RAM contents, caches, registers, error flags, etc.) could be described in M bits. To be able to run interesting x86 programs, a modestly-sized qx86 processor would need to support superpositions over states of size $M = 2^{20}$ or so, which would make the machine orders of magnitude larger than a qFPGA.

Even assuming that one could build a qx86 processor, programming it would be a nightmare: it is unlikely that the average programmer would have any idea how to put a universal quantum instruction to good use in her programs. This latter problem, however, seems relatively easy to solve: ideally, future operating systems would provide a higher-level interface that would allow the programmer to exploit the "quantumness" of her computer without having to get her hands dirty.

In particular, we introduce the qthreads API as one convenient way to abstract away the complexity of quantum hardware while still enabling a programmer to avail herself of the power of quantum computation. We first describe the qthreads API and then we describe how to use qthreads to solve common systems problems more efficiently.

4.1 The qthreads API

Figure 2 lists the routines in our proposed API. The API allows the programmer to create a pool of qthreads, where each qthread (like a pthread) takes in an arbitrary blob of data, does some *classical* computation, and returns an integer. Each qthread executes the same piece of code but each qthread takes a different argument as input.

Once the programmer has created a pool of qthreads, she can ask general questions of the pool. For example, the programmer can ask: "Which qthread returned a non-zero value?" or "How many qthreads in the pool returned a non-zero value?" or "Which qthread in the pool returned the largest integer?"

A quantum computer can answer these questions much faster than a classical computer can. For example, to find a qthread that returns a non-zero value, a classical computer would have to execute the qthread worker routine on each candidate argument until it discovered a qthread that returned something other than zero. In a pool of N threads, with each qthread running for at most T time steps (cycles), a classical computer using this strategy would require $\Theta(NT)$ time in the worst case. In contrast, a quantum operating system could use

Grover's algorithm (Informal Theorem 2) to answer this question in roughly $O(\sqrt{N} \cdot T)$ time—roughly a factor of $\sqrt{N}$ speedup.

The importance of abstraction. Using the qthreads API, the programmer can write the code for the qthread worker as if it were a classical program, and the OS takes care of running the qthreads on the quantum hardware. Avoiding the need to write quantum programs directly is important, since quantum programming would be much trickier than classical programming. Normal techniques for finding bugs in programs, such as using `gdb` or `printf` debugging, would corrupt the state of the quantum machine so debugging the program could itself change the behavior of the program. (This follows from the principle that observing a quantum state causes it to collapse down into a classical state.) Note that qthreads *cannot* have side-effects, so they would have to run in an environment without shared memory or I/O.

4.2 Applications of qthreads

"Most-interesting-job-first" scheduling. Qthreads would, for some types of computations, allow the programmer to effectively implement a "most-interesting-job-first" (MIJF) scheduler: each qthread worker returns an integer describing how "interesting" the result of its computation turned out to be. The scheduler can then return the result of the "most interesting" qthread to the programmer without explicitly running all of the qthreads. In a pool of N jobs, a classical scheduler would require time N to find the most interesting one. A quantum qthreads-based scheduler could perform the same task in time roughly $\sqrt{N}$.

One possible application of a MIJF scheduler is in computational genomics. A pervasive problem in that field is the *edit-distance problem*: given two strings find the minimum number of one-character edits (insertions, deletions, and substitutions) necessary to transform one string into the other [47]. A biologist might need to run tens of thousands of edit-distance calculations (e.g., to look for many different genes in a genome) but might be primarily interested in finding likely matches—pairs of input strings with low edit distance.

The biologist could run each of N edit-distance calculations in a qthread worker. The i^{th} worker would return as output the (negated) edit distance of the i^{th} pair of input strings. By invoking the `qthread_join_max` routine, the biologist could find the index of the pair of strings with the smallest edit distance much faster than running all N edit distance calculations.

MapReduce jobs. MapReduce [27, 28] is a popular programming model for distributed data analysis. In scenarios where the `reduce` function computes a sum of mapper outputs, the qthreads API (using `qthread_join_sum`) enables a programmer to locally execute a MapReduce computation over a pool of N mapper instances while only running the `map` function a total of $O(t \cdot \sqrt{N})$ times, where t is the bit-length of each mapper's output value. In contrast, on a classical computer, running a local MapReduce algorithm would require N invocations of the `map` algorithm.

Unit testing. Using qthreads, a programmer could run N unit tests for the price of $\sqrt{N}$ tests. To do so, the programmer would write a qthread worker routine that takes as input a test case—written in a scripting language, for example—and executes it against the codebase. The programmer would then spin up one qthread for each test case. Each test qthread would return a non-zero status code on failure. The programmer would then ask the operating system for a qthread if any tests returned a non-zero value.

4.3 Implementing qthreads

Although the qthreads API gives the programmer the illusion of running many threads of execution in parallel, this is not at all what a qthreads implementation would actually do. Instead, to implement the qthreads API, the operating system would reframe each `qthread_join*` operation as a problem that it could feed to Grover's algorithm (Informal Theorem 2). The OS would then load the code for the appropriate variant of Grover's algorithm into the quantum co-processor, it would send the x86 code for the worker to the quantum co-processor, and it would load each worker's arguments into the co-processor's quantum RAM (qRAM) [34]. The qx86 CPU would then execute Grover's algorithm and return the result to the OS.

By Grover's algorithm, implementing `qthread_join` requires $O(\sqrt{N})$ queries to qRAM and $O(\sqrt{N})$ invocations of the worker function once. In contrast, executing `qthread_join` on a classical processor would require invoking the worker $\Theta(N)$ times, so the quantum computer provides a $\sqrt{N}$ speed-up in this case.

The `qthread_join_count` and `qthread_join_max` routines would invoke the variants of Grover's algorithm for computing the COUNT and MAX functions [4, 21]. We provide an approximate and exact version of the COUNT API: if there are k non-zero qthreads, the approximate version returns an approximation within a factor of 10% in time roughly $\sqrt{N/k}$, while the exact version returns a correct answer in time roughly $\sqrt{kN}$ [21].

The OS would implement the `qthread_join_sum` routine using `qthread_join_count`. To see how, suppose that the output of each qthread instance is a t-bit integer. We can represent the sum of N integers as $\sum_{i=1}^{t} 2^{i-1} \cdot z_i$, where each z_i is the sum of the i^{th}-least significant bits of each of the N integers. The SUM problem now reduces to computing each of the z_i's. But computing a sum of bits (i.e., $0/1$ values) is precisely the same as *counting* the number of 1s that occur. This operation is supported by the qthreads API.

To compute the sum of t-bit integers, we run `qthread_join_count` t times. On the i^{th} run, we modify the qthread workers to return only the i^{th}-least significant bit of their output. This process yields the values $z_1, \dots, z_t$ (where z_i is the count from the i^{th} thread pool) Finally, we compute $\sum_{i=1}^{t} 2^{i-1} \cdot z_i$ to obtain the sum of the qthread return values. Approximate sums can be computed similarly (using `qthread_join_count_approx`).

4.4 Persistent storage

Ideally, it would be possible to take the state of a qx86 program and save it to disk in a way that would allow restoring the state of the program later on. Unfortunately, many complications arise in the quantum setting. The "no-cloning theorem" [31, 48] says that it is impossible to make copies of an unknown quantum state. Thus, even a basic task such as backing up your quantum data, by saving a copy of it in a safe location, would be essentially impossible. Moreover, writing a quantum state out to a classical disk won't fly either: there is no way to read out the entire quantum state, since measuring a quantum state causes it to immediately collapse back into a classical one. And even if you could somehow read the state without destroying it, a quantum state would require exponentially many classical bits to store on disk.

Given that classical disks are not an option, one might be tempted to ask for some sort of quantum persistent storage. This would be asking for a lot: today's physical qubits remain coherent only for small fractions of a second, and these storage media are nowhere near non-volatile [42]. Yet, since we are optimists, let us consider one interesting application that persistent quantum storage would enable.

Tamper-evident storage. Say that a computer owner Alice drops off her laptop at the tech support desk at her workplace. The technicians may need to access arbitrary files on Alice's computer, but Alice wants to be able to detect after the fact whether they have accessed any of her family photos. With a conventional computer, there is no way for Alice to detect whether the officials have copied the files off of her hard drive. On a conventional file system, Alice can inspect the file's last-read timestamp (`atime`) to check whether a file was accessed. Of course, a clever administrator could just alter the timestamp to hide her tracks. When using a quantum hard drive, we *can* ensure that Alice can detect that someone else has accessed her files.

We borrow an idea from quantum key-distribution protocols [11], whose security rests on the principle that it is infeasible to clone an arbitrary unknown quantum state [48]. To sketch the idea: when creating a file, Alice chooses two 256-bit keys: k_{enc} and k_{tamper}. Alice encrypts her file with k_{enc} (using AES, for example). Then, using k_{tamper}, she encodes the bits of the encryption key k_{enc} into a quantum state, which she stores in the file header. (We assume here that the disk can maintain a quantum state over a period of time.)

Now, access to the file requires reading all of k_{enc}. Anyone who reads the header can decrypt the file, but when anyone other than Alice—i.e., anyone who does not know k_{tamper}—measures the quantum state of k_{enc}, they will collapse the quantum state and will not be able to restore it (with high probability). When Alice later inspects the file header, she will, using ideas from quantum key distribution [11], be able to detect that someone other than her tried to measure k_{enc}.

5 DISTRIBUTED SYSTEMS

The last architecture we consider, which is by far the most speculative, is one in which we have a network of quantum computers connected by qubit-carrying network links. In this section, we explore what we could build with such systems.

Prefetching with superdense coding. Many network workloads are bursty. For example, a web browser spends most of its time transferring nothing over the network, but on every page load, the browser downloads megabytes of content as quickly as possible.

One technique to make traffic less bursty is to use link prefetching [46]: the browser tries to guess the next link the user will click and downloads that content in the background before the user clicks the link. A quantum phenomenon known as *superdense coding* allows a quantum client to prefetch data from a quantum server, even if the client has no idea what data it will need in the future.

Fact 3 (Superdense coding [12]). If a client and server share a single entangled qubit (i.e., a preshared qubit), the server can transmit two classical bits of information to the client by sending a single qubit to the client.

To implement prefetching using superdense coding, the server would continuously produce pairs of entangled qubits and would send one of the two entangled qubits to the client. When the client wants to download some data from to the server, the server could use the entangled qubits it has preshared with the client to transmit data to the client at *twice* the bitrate of the network link.

Superdense coding could improve network performance even when the client is downloading data from a server at a constant rate. Say that the client and server are connected over a network link that has a maximum transfer rate of 100 Mbps in either direction. Using superdense coding, the client could download data from the server at *200 Mbps*. To do so, the client would send entangled qubits to the server at the rate of 100 Mbps while the server would send coded qubits back to the client at a rate of 100 Mbps. The client could extract information from the channel at 200 Mbps, turning a 100 Mbps bidirectional link into a 200 Mbps unidirectional link.

Remote search. Computers connected by a quantum link could run Grover's algorithm across a network. To see one possible application of this idea: let us say that a client wants to search for a special element in a large dataset of N items $(x_1, \ldots, x_N)$ stored at a server (e.g., terabytes of video data). In particular, the client has a classifier $f : \{0,1\}^* \to \{0,1\}$ and wants to find an index i^* on the server such that $f(x_{i^*}) = 1$. The client does not want to download the entire dataset from the server, nor does the client want to upload the code for its classifier to the server—the classifier f might be too large to send or it could involve some secret inputs or algorithms.

The client and server could run Grover search over a network to allow the client to find the matching value x_{i^*} while exchanging a mere $\tilde{O}(\sqrt{N})$ qubits, instead of $\Omega(N)$ classical bits. Of course, the computational burden at both the client and server would be substantial: the client would have to run the classifier roughly $\sqrt{N}$ times and the server would have to execute a qRAM query roughly $\sqrt{N}$ times. Even so, the network traffic savings might be worth the computational cost.

6 RELATED WORK

Richard Feynman initiated the study of quantum computers by pointing out that classical computers seem too weak to efficiently simulate quantum physical systems [32]. Computer scientists have demonstrated that quantum computers can provide speedups for a number of problems [14, 30, 45], including factorization of integers [44], inversion of one-way functions [35], and certain machine learning problems [38] (though these latter algorithms come with caveats [2]). Known applications of distributed quantum computing systems include unconditionally secure key agreement over authenticated quantum channels [11] and superdense coding [12] (see Section 5). More recent work has investigated quantum consensus [8] and quantum secure multi-party computation [25]. Broadbent and Tapp survey these and related results [22].

7 CONCLUSION

If quantum computers are indeed on their way, the operating system will have an important role in enabling constructive applications of this potentially destructive technology. As Richard Feynman said of quantum computing, “It’s a wonderful problem, because it doesn’t look so easy” [32].

REFERENCES

[1] Scott Aaronson. 2008. The limits of quantum computers. *Scientific American* 298, 3 (2008), 62–69.

[2] Scott Aaronson. 2015. Read the fine print. *Nature Physics* 11, 4 (2015), 291–293.

[3] Dorit Aharonov. 2003. A simple proof that Toffoli and Hadamard are quantum universal. *arXiv preprint quant-ph/0301040* (2003).

[4] Ashish Ahuja and Sanjiv Kapoor. 1999. A quantum algorithm for finding the maximum. *arXiv preprint quant-ph/9911082* (1999).

[5] Joël Alwen, Binyi Chen, Chethan Kamath, Vladimir Kolmogorov, Krzysztof Pietrzak, and Stefano Tessaro. 2016. On the Complexity of scrypt and Proofs of Space in the Parallel Random Oracle Model. In *EUROCRYPT 2016*. Springer, 358–387.

[6] Andris Ambainis. 2004. Quantum search algorithms. *ACM SIGACT News* 35, 2 (2004), 22–35.

[7] R. Barends, J. Kelly, A. Megrant, A. Veitia, D. Sank, E. Jeffrey, T. C. White, J. Mutus, A. G. Fowler, B. Campbell, Y. Chen, Z. Chen, B. Chiaro, A. Dunsworth, C. Neill, P. O/’Malley, P. Roushan, A. Vainsencher, J. Wenner, A. N. Korotkov, A. N. Cleland, and John M. Martinis. 2014. Superconducting quantum circuits at the surface code threshold for fault tolerance. *Nature* 508, 7497 (04 2014), 500–503.

[8] Michael Ben-Or and Avinatan Hassidim. 2005. Fast quantum Byzantine agreement. In *STOC*. ACM, 481–485.

[9] Charles H Bennett. 1973. Logical reversibility of computation. *IBM journal of Research and Development* 17, 6 (1973), 525–532.

[10] Charles H Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. 1997. Strengths and weaknesses of quantum computing. *SIAM journal on Computing* 26, 5 (1997), 1510–1523.

[11] Charles H. Bennett and Gilles Brassard. 1984. Quantum Cryptography: Public Key Distribution and Coin Tossing. In *International Conference on Computers, Systems & Signal Processing*.

[12] Charles H Bennett and Stephen J Wiesner. 1992. Communication via one-and two-particle operators on Einstein-Podolsky-Rosen states. *Physical Review Letters* 69, 20 (1992), 2881.

[13] Daniel J. Bernstein. 2009. Introduction to post-quantum cryptography. In *Post-quantum cryptography*, Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen (Eds.). Springer, 1–14.

[14] Ethan Bernstein and Umesh Vazirani. 1997. Quantum complexity theory. *SIAM J. Comput.* 26, 5 (1997), 1411–1473.

[15] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. 2016. Argon2: new generation of memory-hard functions for password hashing and other applications. In *European Symposium on Security and Privacy*. IEEE, 292–302.

[16] Dan Boneh, Henry Corrigan-Gibbs, and Stuart Schechter. 2016. Balloon Hashing: a Provably Memory-Hard Function with a Data-Independent Access Pattern. In *ASIACRYPT*.

[17] Joseph Bonneau. 2012. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *Symposium on Security and Privacy*. IEEE, 538–552.

[18] Joseph Bonneau, Cormac Herley, Paul C. Van Oorschot, and Frank Stajano. 2012. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Symposium on Security and Privacy*. IEEE, 553–567.

[19] Michel Boyer, Gilles Brassard, Peter HÃÿyer, and Alain Tapp. 1998. Tight Bounds on Quantum Searching. *Fortschritte der Physik* 46, 4-5 (1998), 493–505.

[20] Matt Braithwaite. 2016. Experimenting with Post-Quantum Cryptography. https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html. (July 7, 2016). Accessed 21 January 2017.

[21] Gilles Brassard, Peter Høyer, and Alain Tapp. 1998. Quantum counting. In *International Colloquium on Automata, Languages, and Programming*. Springer, 820–831.

[22] Anne Broadbent and Alain Tapp. 2008. Can quantum mechanics help distributed computing? *ACM SIGACT News* 39, 3 (2008), 67–76.

[23] Davide Castelvecchi. 2016. Quantum computers ready to leap out of the lab in 2017. *Nature News* 541, 7635 (January 3, 2016).

[24] Jamie Condliffe. 2016. Google’s Quantum Dream May Be Just Around the Corner. https://www.technologyreview.com/s/602283/googles-quantum-dream-may-be-just-around-the-corner/. (September 1 2016).

[25] Claude Crépeau, Daniel Gottesman, and Adam Smith. 2002. Secure multi-party quantum computation. In *STOC*. ACM, 643–652.

[26] CVE-2014-3570 2014. CVE-2014-3570: Bignum squaring may produce incorrect results. (May 2014).

[27] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*. 137–150.

[28] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[29] S. Debnath, N. M. Linke, C. Figgatt, K. A. Landsman, K. Wright, and C. Monroe. 2016. Demonstration of a small programmable quantum computer with atomic qubits. *Nature* 536, 7614 (08 2016), 63–66.

[30] David Deutsch and Richard Jozsa. 1992. Rapid solution of problems by quantum computation. In *Proceedings of the Royal Society of London*, Vol. 439. The Royal Society, 553–558.

[31] DGBJ Dieks. 1982. Communication by EPR devices. *Physics Letters A* 92, 6 (1982), 271–272.

[32] Richard P Feynman. 1982. Simulating physics with computers. *International Journal of Theoretical Physics* 21, 6 (1982), 467–488.

[33] Christian Forler, Stefan Lucks, and Jakob Wenzel. 2014. Memory-demanding password scrambling. In *ASIACRYPT*. Springer, 289–305.

[34] Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. 2008. Quantum random access memory. *Physical Review Letters* 100, 16 (2008), 160501.

[35] Lov Grover. 1996. A fast quantum mechanical algorithm for database search. In *STOC*. ACM, 212–219.

[36] Rolf Landauer. 1961. Irreversibility and heat generation in the computing process. *IBM journal of research and development* 5, 3 (1961), 183–191.

[37] Chris Lee. 2016. How IBM’s new five-qubit universal quantum computer works. https://arstechnica.com/science/2016/05/how-ibms-new-five-qubit-universal-quantum-computer-works/. (May 4 2016).

[38] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. 2013. Quantum algorithms for supervised and unsupervised machine learning. *arXiv preprint arXiv:1307.0411* (2013).

[39] Microsoft 2016. Microsoft Lattice Cryptography Library. https://www.microsoft.com/en-us/research/project/lattice-cryptography-library/. (April 19, 2016). Accessed 21 January 2017.

[40] Robert Morris and Ken Thompson. 1979. Password security: A case history. *Commun. ACM* 22, 11 (1979), 594–597.

[41] National Institute of Standards and Technology. 2016. Post-Quantum Crypto Project. http://csrc.nist.gov/groups/ST/post-quantum-crypto/. (December 15, 2016). Accessed 21 January 2017.

[42] Nissim Ofek, Andrei Petrenko, Reinier Heeres, Philip Reinhold, Zaki Leghtas, Brian Vlastakis, Yehan Liu, Luigi Frunzio, SM Girvin, L Jiang, et al. 2016. Extending the lifetime of a quantum bit with error correction in superconducting circuits. *Nature* (2016).

[43] Colin Percival. 2009. Stronger key derivation via sequential memory-hard functions. In *BSDCan*.

[44] Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.* 26, 5 (1997), 1484–1509.

[45] Daniel R Simon. 1997. On the power of quantum computation. *SIAM J. Comput.* 26, 5 (1997), 1474–1483.

[46] W3C. 2016. Resource hints. https://www.w3.org/TR/resource-hints/#prefetch. (Dec. 2016). Accessed 21 January 2017.

[47] Robert A Wagner and Michael J Fischer. 1974. The string-to-string correction problem. *J. ACM* 21, 1 (1974), 168–173.

[48] William K Wootters and Wojciech H Zurek. 1982. A single quantum cannot be cloned. *Nature* 299, 5886 (1982), 802–803.