Jared Stillford
16 November 2015
CS4960-001
Dr. Melanie Martin

Quantum Computing: A Matter of Time

Within the last few decades, computer programming has grown exponentially in both its output of newer technologies and the subsequent adoption by the general population. The evolution of the computer is nearing another milestone: the quantum computer. The idea of quantum computing can conjure many images of nearly limitless power and instantaneous answers with minimal work on the part of the human counterpart using it. While these examples are highly unlikely, at least in any foreseeable future, the real power of the quantum computer comes not only from what it can offer in terms of functionality but also in what is learned by the search itself.

The discussion is far too lengthy for any one paper, and new ideas are created almost daily for a shot at being the next leap forward. What will be attempted in the course of this paper is a comparison between a pair of classical and quantum components that may be considered the building blocks of the enterprise. These components are the storage of quantum data itself and the current approach toward a unifying language for the machines. Additionally, a detailed look at the quantum language Quipper will be applied toward these two ideas to see how problems with both are actually being tested.

Introduction

At least with regard to their first generation counterparts, quantum computers will not appear to be too different from current computers in terms of necessary hardware. There will still

need to be a source of energy, a storage component, a processing unit, and many other pieces with varying ways to set them up. The transistors for storage will still be composed of metalloid semiconductor compounds – although changes in the exact elements chosen will have additional requirements (e.g., decay lifetimes) to compensate for the additional stress (Vyavahare, 2011 and Tribollet. 2005). Transistors are a good way to begin the discussion as they have representations both in the physical and digital circuitry of the computer.

Different languages will inevitably be used by the multitude of quantum computers yet to come. The true difficulty will be in establishing the first few languages to help make this leap to a medium in which programmers are still unsure of the full potential, or limits, of the machines. It is especially difficult to decide upon a quantum language when classical languages are still hotly debated about their own efficiencies. Realistically, the most cutting-edge designers use combinations of languages in layered hierarchies, but that is a complication left to further personal study.

Classical to Quantum: Storage

The basic components required of a quantum computer are analogous to those needed by their classical counterparts. While there may be numerous ways to do so, storage of current media occurs primarily through magnetic storage (e.g., hard disks) or flash memory. High intensity fields of either electric or magnetic origin are used to write data in such a manner that lower energy fields may then read it. This is the driving force behind the physical nature of transistors, circuits, and more complex states with each building on the one before it. The interpretation of these structures results in the bits more commonly known by computer programmers.

Bits are simplified theoretical models that are used to show how the varying parts interact. They are great for showing the many binary relations programmers deal with while coding; for example, Booleans exist as true or false while the different types of numbers are expressed as positive or negative. This is done through the high or low energy states of the transistors with a third state of some intermediate energy being merely the transition between the two. As it stands, that transitional energy cannot be used by the registers to allow for any additional storage state.

This in-depth look at bits and their physical nature is what quantum programmers are trying to recreate with slight modifications. A quantum computer is assumed to need similar functionality and current quantum languages are calling them quantum bits – or qbits, for short. They are similar to the classical bits in that they represent the energy states of some physical storage but are able to take it one step further. A qbit has the possibility to be assigned to either state of a classical bit as well as a new state tied directly to the transitional energy. For a Boolean this is synonymous with being true, false, or both.

Any knowledge of simple logic can show the inaccuracy of the previous statement, but in terms of storage it is without equal for efficiency. Consider a circuit that takes extremely large numbers of inputs and can calculate a result, or many meaningful outputs, in real-time. This is the potential of qbits that programmers are trying to harness, but these quantum circuits are not without their own requirements to be able to work in the first place as will be discussed later. However, should they succeed, quantum theoretical models have already shown dozens of algorithms that work at exponentially faster rates on quantum computers (Shor, 1997). For outputs requiring many registers to change from one energy state to the other, it is logical that the change is both faster and less costly if the inputs are already held in some state between the two.

Classical to Quantum: Language

There are two vastly different approaches in considering what makes up a good programming language: the imperative and the functional. Classical imperative languages such as Java and Python require the user to explicitly state at each step what the program will do. It is essentially a command with its limitations based primarily on the capacity of the user to either complicate or simplify the path taken toward a result. The benefit is how intuitive this really is; by stating what occurs along the way, users may track the desired function more readily and know precisely what it interacts with. However, this dependence can slow down both run time and desired modifications to the code.

Functional languages like Haskell and Lisp allow a degree of flexibility for the computer. Libraries are loaded into the program while the user specifies initial parameters for variables and different end results. Once started, the program may choose its own path from beginning to completion as defined within the libraries and boundaries of variables. Although there are other aspects to these languages, this is the primary reason for choosing them and their logic counterparts as early languages for artificial intelligence. The drawback to these languages is that less micromanagement by the user can prove difficult to the debugging process if answers are vastly different than those desired.

What is the best language for a quantum architecture? That is still up for debate, but both of the aforementioned language families are just as likely to be the first implemented when hardware catches up. While the first quantum language to be relatively complete was imperative and known as Quantum Computing Language (QCL), forerunners are now overwhelmingly functional (Gay, 2006). Haskell has been scaled-up to have its data types refer to quantum data types through its own type-class mechanism. A drawback of previous functional languages had

been that they could not separate data creation from evaluation, but Haskell's "Template Haskell" allows for the program to manipulate its own data accordingly (Sabry, 2003). This could be useful as our own limitations are taken out of quantum computers to allow them to create and alter code for increasingly better, or faster, results.

Quipper

Now for a look at a language that attempts to bridge the disconnect between these two ideas while providing a framework with which future quantum programmers may build upon. Quipper utilizes the previously mentioned functionality offered by Haskell and is implemented "as a deeply embedded domain-specific language" (Valiron, 2015). What this means is Quipper is not only developed to use the syntax of its host language for a majority of its computations, but a deeply embedded DSL means that it may have other infrastructure used to support or modify the existing language. This is useful for pulling out functions to be used on other sections of code (Gill, 2014). As an example, Quipper can use this section of code by transforming it (e.g., optimization, recursion, repeating block structures) before implementing it in another area.

For a classical computer this is where the usefulness ends, but for quantum computing it is a gateway to solving even larger problems. One of the biggest obstacles concerning qbits is observing the state of its transistors and, consequently, the inputs for the circuits being read by the quantum processor. The superposition of one semiconductor used for the data storage is not entirely independent of the other ones around it or even the influence of the machine, or user, through observation. Loosely this means that by observing a qbit mid-run we would force it out of its superposition and into either the "high" or "low" state at random, causing the output to immediately by incorrect. The deeply embedded nature of Quipper comes in handy here since we

may then extract the body of the function to observe how it is being manipulated without actually affecting the qbit itself.

A simpler, but by no means less useful, ability of Quipper is in its ability to both represent and orient circuits via block structures. By adding the ability for functional circuits to be read and stored as block structures, the program may again call on them in such a way that they are reused as subroutines. This can save both in time and memory allocation by building larger structures from simpler repeating ones. Another aspect of these repetitions is the simplicity in their recursion. Again, by creating a command to reverse the function passed to it this allows for complicated structures to be flipped by iterating through and reversing the smaller blocks individually.

This touches briefly upon another characteristic of quantum algorithms in their inability to run at all unless done within an entirely reversible circuit (Saeedi and Markov, 2013).Consider a simple example in which a circuit takes inputs x, y, and z as Booleans while outputting x', y', and z'; not only do fully reversible circuits have the same number of inputs as outputs, they must also be one-way circuits while avoiding loops. Unlike standard circuits which may derive an answer from its inputs but not the other way around, reversible circuits allow for the determination of inputs from its outputs. This ties back to the nature of the qbits and their storage within the superposition as mentioned before.

Conclusion

Now is also a good time to mention the simplified nature of transistors and quantum particles mentioned within this paper. Additional research will be required on behalf of the more specific disciplines, such as the engineers in charge of manufacturing the hardware and security

analysts who will have to eventually develop quantum encryptions without loss of data for qualified users. In order to reach those points, however, information will have to be obtained and passed along so that others may add to the conversation.

While a majority of this discussion is based on theoretical design, the development of such protocols is still a worthwhile venture for exploration in the computing world – especially for the generations growing up within an already developed classical model of programming. Although the examples of storage and language are only two of many problems to be solved in a transition toward quantum computing, the decision of how they are approached could set the precedent for the rest. Languages like Quipper help to create new goals by understanding both the positives and negatives of that specific attempt so there will no doubt be many after it. Quantum theories are still being explored even outside of computer science, so the chance for the next big breakthrough could be just around the corner.

Works Cited

Andy Gill. 2014. Domain-specific Languages and Code Synthesis Using Haskell. *Queue* 12, 4, pages 30 (April 2014), 14 pages.

Jerome Tribollet. 2005. Globally controlled artificial semiconducting molecules as quantum computers. *Quantum Info. Comput.* 5, 7 (November 2005), 561-572.

Benoît Valiron, Neil J. Ross, Peter Selinger, D. Scott Alexander, and Jonathan M. Smith. 2015. Programming the quantum future. *Commun. ACM* 58, 8 (July 2015), 52-61.

Daniel Kudrow, Kenneth Bier, Zhaoxia Deng, Diana Franklin, Yu Tomita, Kenneth R. Brown, and Frederic T. Chong. 2013. Quantum rotations: a case study in static and dynamic machine-code generation for quantum computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (ISCA '13). ACM, New York, NY, USA, 166-176.

A. Ajit Vyavahare, Amit Patil, and S. Prashant Hadapad. 2011. Quantum size effect in nanosize semiconductor materials. In *Proceedings of the International Conference & Workshop on Emerging Trends in Technology* (ICWET '11). ACM, New York, NY, USA, 1374-1374.

Amr Sabry. 2003. Modeling quantum computing in Haskell. In Proceedings of the 2003 ACM SIGPLAN workshop on Haskell (Haskell '03). ACM, New York, NY, USA, 39-49.

Mehdi Saeedi and Igor L. Markov. 2013. Synthesis and optimization of reversible circuits – a survey. *ACM Comput. Surv*. 45, 2, Article 21 (March 2013), 34 pages.

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

Simon J. Gay. 2006. Quantum programming languages: survey and bibliography. *Mathematical. Structures in Comp. Sci.* 16, 4 (August 2006), 581-600.