Rapid Prototyping with Symbolic Computation: Fast Development of Quantum Annealing Solutions

Mark Hodson, Duncan Fletcher, Dan Padilha, Tristan Cook
QxBranch LLC
Washington DC, USA
mark.hodson@qxbranch.com, duncan.fletcher@qxbranch.com, dan.padilha@qxbranch.com, tristan.cook@qxbranch.com

Quantum computing promises to improve the speed and scalability of computations over that of classical computing hardware. At this early stage of quantum computer hardware development, software frameworks which support rapid prototyping of quantum solutions on small-scale hardware or simulators are necessary to explore the application of quantum algorithms to hard computational problems. We present a software library, "QxLib," which incorporates symbolic computation of optimization functions for quantum annealers as one means to enable rapid prototyping. We demonstrate its effectiveness on integer linear programming and integer factorization problems.

Keywords—Ising model; Python; quantum annealing; quantum computing; symbolic computation; universal quantum computation

I. Introduction

Quantum computing promises to improve the speed and scalability of computations that are hard to perform on classical computing hardware. This approach to computing is being developed through two computational models: quantum annealing (QA) and universal quantum computation (UQC).

QA finds solutions to optimization problems that exist as the ground states of a Hamiltonian expressed in the form of an Ising model [1]. UQC acts by initializing, manipulating and measuring quantum states directly through sequences of quantum gate operations [2]. The speed and scalability of QA is being assessed experimentally by operating on early-stage hardware including the commercially available annealers from D-Wave Systems [3], supported by theoretical characterization work such as [4]. The speed and scalability of UQC is established theoretically by the application of quantum information theory to quantum algorithms such as Shor's [5] and Grover's [6], while the engineering efforts to develop UQC hardware are ongoing.

To map computational problems onto a quantum computer we desire layered software abstractions. These abstractions bring the benefits of the quantum computer hardware into the domain of existing, classical software systems. Given the current early stage of quantum algorithm exploration and development, we believe these software abstractions should enable rapid prototyping and experimentation. When hardware is not available or we seek to test concepts that do not fit on existing hardware, simulators can be used.

In section II we present a quantum computer system model that aligns with the requirements of an operational system, enabling a smooth transition from research and development into operation. In section III we present our software library QxLib; its architecture in the context of the system model and its application to symbolic computation of two hard problems expressed as QA optimization functions. A summary of our findings is provided in section IV.

II. Quantum Computing System Model

Just as the OSI (Open Systems Interconnect) model [7] defines a protocol stack with seven abstraction layers designed to bring the benefits of telecommunications technologies to application developers, a quantum computing system model (Fig. 1) defines a hardware-software stack designed to bring the benefits of quantum computing technologies to the same audience. Future standards for implementation of each layer then enable industry collaboration and technology interchange in achieving desired application outcomes.

Establishing a quantum computer system model early in the quantum hardware development process focuses the software development efforts towards architectures that are ready for the transition to an operational system. Architectures developed atop early-stage hardware or simulators (Fig. 2) enable real-world applications to be tested, providing input to the final architecture requirements. This approach is consistent with how improvements to classical computing technologies such as Intel processors are brought to market, through tools such as the Intel Software Development Emulator[1].

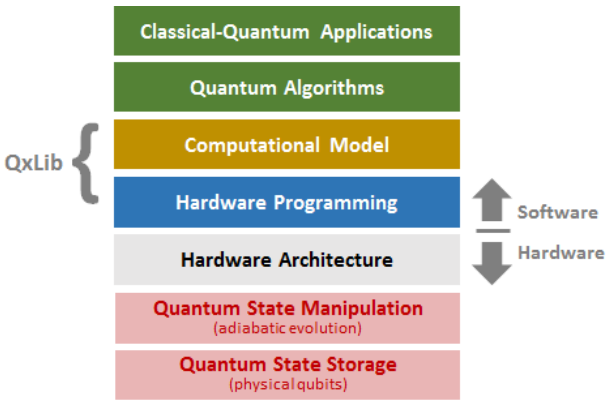

Fig. 1. Quantum Computer System Model based on Hardware

[1] Online documentation available at: https://software.intel.com/en-us/articles/intel-software-development-emulator

M. Hodson, D. Fletcher, D. Padilha and T. Cook, "Rapid prototyping with symbolic computation: Fast development of quantum annealing solutions," 2016 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, 2016, pp. 1-5.
doi: 10.1109/HPEC.2016.7761632
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7761632&isnumber=7761574

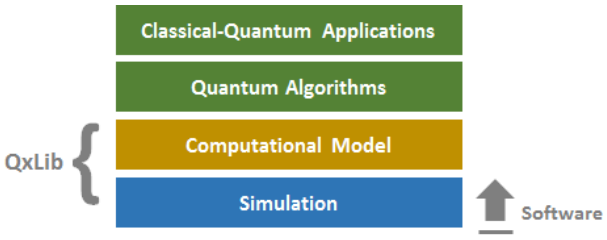

Fig. 2. Quantum Computer System Model based on Simulation

Current early-stage developments towards this goal include D-Wave Systems' QA software architecture [8]. For UQC systems, development of technologies underpinning elements of the system model such as quantum circuit compilation and error correction codes are underway.

The computational model layer is of particular interest in early-stage problem exploration and prototype development as it provides an Application Programming Interface (API) for which quantum algorithms and hybrid classical-quantum applications can be developed. Ease of use in this layer aids rapid experimentation and knowledge generation. We have focused our development efforts in this layer and its connection to hardware programming and simulation.

III. QxLib Software Library

A. Overview

We present a software library written in Python and C++ called QxLib that supports rapid prototyping of quantum algorithms based on the QA computational model (Fig. 3).

The API for QxLib is in Python. Python is a common high-level language with third-party support for data analytics and machine learning at sufficient scale to support prototype, hybrid classical-quantum applications. Performance-critical functions are implemented in C++.

When quantum computing hardware is not available for experimentation, quantum computer simulation can be used in its place. Quantum computer simulators are an effective means to establish and evaluate quantum algorithms before hardware is available at the necessary scale, and supporting the necessary features [9]. QxLib supports simulation as a surrogate for hardware programming. This provides an effective, software-only environment enabling assessment of feasibility and debugging of quantum algorithms.

Simulators vary in fidelity. Some act only to provide a representative result using established classical methods, such as using simulated annealing as a substitute for a quantum annealer [10]. Others model the specific way in which the state evolves within the quantum computer [11] [12]. The simulator is selected to support the experimental outcomes required by the end-user.

QxLib interfaces with the D-Wave Systems' SAPI [8] for access to a D-Wave 2X™ quantum annealer or one of its simulators (*qxdwave*). QxLib also provides simulators based on QA solvers by Sergei Isakov et al [10] (*qxisakov*), Alex Selby [13] (*qxhfs*) and a custom brute-force solver (*qxbrute*).

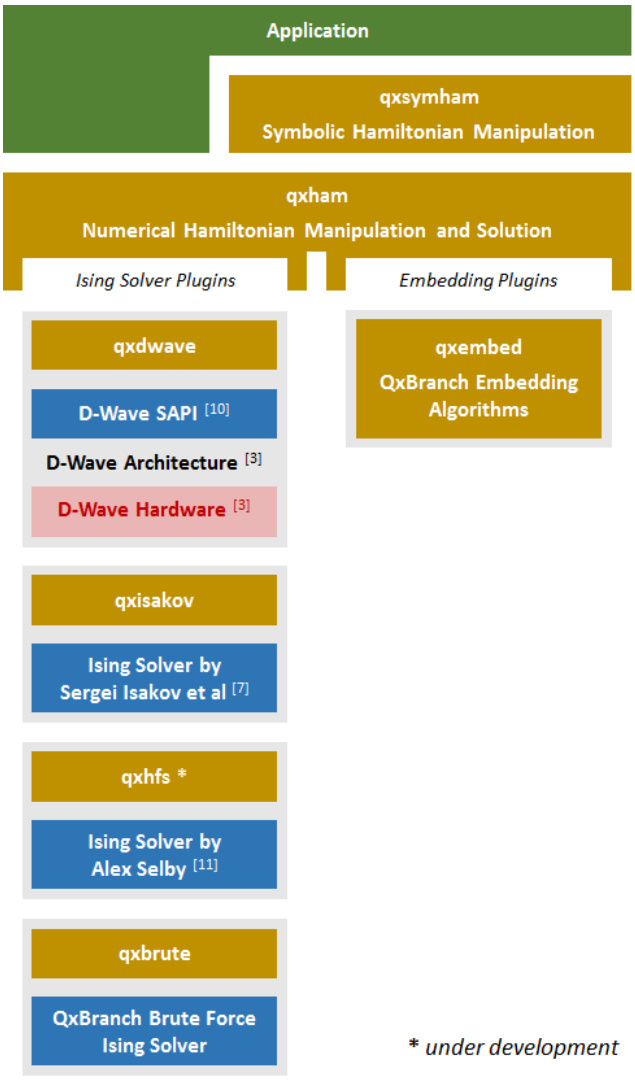

Fig. 3. QxLib Architecture with Ising Solver and Embedding plugins

B. Numeric Computation of the Hamiltonian

The computational model of current QA systems can be expressed as an Ising model of N variables (1) whose lowest-energy solution to Hamiltonian H solves for binary variables s_i (2). The optimal solution depends on single-variable biases h_i and two-variable interaction weights J_{ij}.

$$H = \sum_{i}^{N} h_i s_i + \sum_{i,j}^{N} J_{ij} s_i s_j \quad (1)$$

$$s_i \in \{-1, +1\} \quad (2)$$

The QxLib component *qxham* is concerned with the expression and solution of numeric Ising model problems. Solutions are generated by Ising Solver Plugins encapsulating hardware interfaces or simulators as described above.

Some of these plugins do not solve a fully-connected Ising model; they have connectivity that restrict specific J_{ij} terms to be zero. One example is D-Wave Systems' Chimera – a non-planar graph of at most degree six. These plugins require a

support process known as minor embedding to transform a general Ising model into a form that is executable by the hardware or simulator. Embedding algorithms are also provided within QxLib (*qxembed*).

C. Symbolic Computation of the Hamiltonian

Many optimization problems that are computationally hard can be transformed into the Ising model but are not naturally expressed in this form. Symbolic computation can assist end-users in transforming their optimization function into the Ising model form. The only constraint is that the problem be expressed using sums and products. Symbolic computation is supported by products such as Mathworks Matlab[2], Wolfram Mathematica[3], and the open source Python library SymPy[4].

The QxLib component *qxsymham* is concerned with the expression and solution of symbolic Ising model problems. It adapts SymPy and NumPy[5] for the purpose of transforming a compatible symbolic Hamiltonian into a numeric Ising model for execution by *qxham*.

Symbolic computation of a Hamiltonian involves 3 steps:

1. Expansion and collection of terms involved in sums and products.
2. Reduction of multiple-variable interactions to two-variable interactions using ancillary variables.
3. Simplification using Ising model identities.

Multi-variable interactions can be reduced by introduction of ancillary variables as in [14] or [15]. Simplifications include substitution of the identity (4).

$$s^2 = 1 \tag{4}$$

Symbolic computation affords additional freedoms of expression. Parameter placeholder variables may be used and substituted just before execution of the solver as a way to reuse the transformation result for many problems of the same form. Composite variables may be defined that are ultimately reduced to binary Ising variables (2).

The simplest example is the conversion (5) to a Quadratic Unconstrained Binary Optimization (QUBO) variable (6).

$$x_i = \frac{s_i + 1}{2} \tag{5}$$

$$x_i \in \{0, 1\} \tag{6}$$

More complex variables such as an unsigned integer of bit depth *K* can be described as in (7), yielding a variable with a range on an interval as in (8).

$$v_i = \sum_{k=0}^{K-1} 2^k \frac{s_{i,k} + 1}{2} \tag{7}$$

$$v_i \in [0, 2^K - 1] \tag{8}$$

The ability to freely express sums, products, multi-variable interactions and composite variables as symbolic computations improves end-user productivity when defining and executing prototype, hybrid classical-quantum applications.

The following examples show how the natural form of two common, computationally hard problems may be expressed using the symbolic computation facility in QxLib.

1) Integer Linear Programming

Integer Linear Programming (ILP) is an NP-hard problem that asks: what is the largest value of *v* (8) that meets constraint (9)?

$$v = \mathbf{c} \cdot \mathbf{x} \tag{8}$$

$$\mathrm{S}\mathbf{x} = \mathbf{b} \tag{9}$$

This can be expressed for binary variables *x* (6) as the Hamiltonian (10) for *N* variables and *M* constraints [16].

$$H = A \sum_j^M \left(b_j - \sum_i^N S_{ji} x_i \right)^2 - B \sum_i^N c_i x_i \tag{10}$$

This can be easily extended using symbolic computation to a more general ILP form that allows unsigned integers of bit depth *K* by replacing each x_i (6) with the definition (7). The implementation and execution of this problem using QxLib takes on an intuitive form in software code (Fig. 4). QxLib transforms the Hamiltonian without laborious hand expansion, supporting experimentation on how ILP problems can best be solved by quantum annealing [17].

2) Integer Factorisation

Integer factorization is unlikely, although unproven, to be NP-Complete, but is nevertheless considered a "hard" classical problem with a known, efficient (BQP) quantum solution [18].

For integers that are a product of two factors, *p* and *q*, the factors of *N* can be found by solving $N = pq$. This equation can be expressed as the Hamiltonian of (11) [19]:

$$H = (N - pq)^2 \tag{11}$$

For a known *N*, bounds on the possible size of non-trivial factors *p* and *q* can be set [18]. An integer interval variable spanning a range [*A*, *B*] (14) can then be used to generalize the possible solutions (13) by first considering the required number of QUBO variables *K* needed to support it (12). This variable uses the "log trick" proposed in [16].

$$K = \lfloor \log_2(B - A) \rfloor + 1 \tag{12}$$

$$v_i = A + \left(B - A + 1 - 2^{K-1}\right) x_{i,K-1} + \sum_{k=0}^{K-2} 2^k x_{i,k} \tag{13}$$

$$v_i \in [A, B] \tag{14}$$

---

[2] Refer to the symbolic toolbox documentation online at: http://www.mathworks.com/products/symbolic

[3] Refer to the symbolic computation documentation online at: https://reference.wolfram.com/language/tutorial/SymbolicComputation.html

[4] Refer to the SymPy documentation online at: http://www.sympy.org/en/index.html

[5] Refer to the NumPy documentation online at: http://www.numpy.org/

```
# DEFINE A PARAMETERIZED SYMBOLIC PROBLEM #
N = 10     # ... number of variables
M = 3      # ... number of constraints
K = 4      # ... integer bit depth
sp = qxsymham.SymbolicProblem()

# Define the array of integer variables
# that form the solution.
x = sp.new_variable_array(
      qxsymham.UnsignedIntegerVariable,
      "x", (N,), K)

# Define parameters for the problem that
# are set per-instance.
S = sp.new_variable_array(
      qxsymham.Parameter, "S", (M,N))
b = sp.new_variable_array(
      qxsymham.Parameter, "b", (M,))
c = sp.new_variable_array(
      qxsymham.Parameter, "c", (N,))
A = sp.new_variable(
      qxsymham.Parameter, "A")
Z = sp.new_variable(
      qxsymham.Parameter, "Z")

# The optimization term of the Hamiltonian.
H_opt = -np.dot(c, x)

# The penalty term of the Hamiltonian.
H_pen = A*sum((b - np.dot(S, x))**2)

# Set the Hamiltonian expression.
sp.set_expression(H_opt + H_pen)

# CREATE AN INSTANCE OF THE PROBLEM #
# Numerical values must be assigned to
# all parameters in order to generate a
# problem instance that can be solved.
# An example with arbitrary values:
parameter_dict = {}
parameter_dict[sp.A] = 5.0
for i in range(self.N):
    parameter_dict[sp.c[i]] = 1.0
for i in range(self.M):
    for j in range(self.N):
        parameter_dict[sp.S[i,j]] = 2.5
# etc.

# Retrieve the IsingProblem instance given
# these parameters.
ising_problem =
sp.get_ising_problem(parameter_dict)

# SOLVE AN INSTANCE OF THE PROBLEM #
solver  = qxbrute.BruteCompleteSolver()
results = solver.run(ising_problem)
optimal_energy =
  ising_results[0].energy_scaled
optimal_solution =
  ising_results[0].solution
```

Fig. 4. Integer Linear Programming example Python code using QxLib

```
# DEFINE AN INTEGER INTERVAL VARIABLE #
class IntervalVariable(qxsymham.Variable):
  def __init__(self, name, min, max):
    K = int(math.floor(math.log(
            max - min, 2)) + 1)
    bits = [qxsymham.QuboVariable(
        name + "[{0}]".format(k))
        for k in range(bit_depth)]
    expr = min +
       (max - min + 1-2**(bit_depth-1))
       * bits[bit_depth-1]
       + sum([2**k * bits[k]
           for k in range(bit_depth-1)])
    super(IntervalVariable, self).
      __init__(self, name, expr, bits)
```

Fig. 5. Integer Interval Variable example Python code using QxLib

Implementation of this variable using QxLib is shown in (Fig. 5).

Integer factorization is particularly suited to symbolic computation as the expansion is tedious, problem-dependent and involves up to 4-variable interactions. Solving this problem using QxLib follows the same structure as (Fig. 4), injecting use of the integer interval variable for *p* and *q*.

D. Hardware and Simulation Results

Using QxLib, the same symbolic problem can be executed on a variety of hardware or simulator technologies, enabling experimentation on the effect of parameter setting and on the comparative performance of the underlying technologies. An example that the authors performed investigated the effect of embedding on the performance of Integer Linear Programming problems on the D-Wave 2 hardware [17]. This was supported by experimentation using the Isakov et al Ising solver [10] to ensure the program was correct before submitting problems to the limited D-Wave 2 hardware resource. Though the solution probabilities returned from a simulated annealing algorithm are quite different to those of a quantum annealer, we found this type of simulator to be sufficient to find the highest probability solution and so enable debugging of program code.

IV. Summary

In this paper we have shown a direct mapping of high-level Hamiltonian descriptions into software supported by QxLib for two computationally hard problems that can then be executed on a quantum annealer. We have also shown how this approach fits into the quantum computer system model needed to operationalize QA quantum algorithms; a conceptual model that applies equally to UQC.

References

[1] G.E. Santoro, E. Tosatti, “Optimization using quantum mechanics: quantum annealing through adiabatic evolution,” J. Phys. A 39, R393, 2006.

[2] N. Gershenfeld, I.L. Chuang, “Quantum computing with molecules,” Scientific American, June 1998.

[3] M.W, Johnson et al, “Quantum annealing with manufactured spins,” Nature 473.7346, 2011, pp. 194-198.

[4] D.S. Steiger, T.F. Rønnow, M. Troyer, “Heavy tails in the distribution of time-to-solution for classical and quantum annealing,” arXiv: 1504.0799, 2015.

[5] P.W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," SIAM J. Comput. 26 (5), 1997, pp. 1484-1509.

[6] L.K. Grover, “A fast quantum mechanical algorithm for database search,” Proceedings, 28th Annual ACM Symposium on the Theory of Computing, May 1996, p. 212.

[7] International Organization for Standardization, “ISO/IEC 7498-4:1989 Information technology- Open Systems Interconnection basic reference model: Naming and addressing,” 1989.

[8] M. Booth, E. Dahl, M. Furtney, S.P. Reinhardt, “Abstractions considered helpful: A tools architecture for quantum annealers,” unpublished.

[9] T. Häner, D.S. Steiger, M. Smelyanskiy, M. Troyer, “High performance emulation of quantum circuits,” arXiv:1604.06460, 2016.

[10] S.V. Isakov, I.N. Zintchenko, T.F. Ronnow, M. Troyer, “Optimized simulated annealing for Ising spin glasses,” arXiv:1401.1084, 2015.

[11] D. Wecker, K.M. Svore, “LIQUi|>: A software design architecture and domain-specific language for quantum computing,” arXiv:1402:4467, 2014.

[12] A.S. Green, P.L. Lumsdaine, N.J. Ross, P. Selinger, B. Valiron, “Quipper: A scalable quantum programming language,” arXiv: 1304.3390, 2013.

[13] A. Selby, “Efficient subgraph-based sampling of Ising-type models with frustration,” arXiv:1409.3934, 2014.

[14] Z. Bian, F. Chudak, W. G. Macready, L. Clark, F. Gaitan, “Experimental determination of Ramsey numbers,” Physical review letters 111.13, 2013, p. 130505.

[15] R. Babbush, B. O’Gorman, A. Aspuru-Guzik, “Resource efficient gadgets for compiling adiabatic quantum optimization problems,” arXiv:1307.8041, 2013.

[16] A. Lucas, “Ising formulations of many NP problems,” arXiv:1302.5843, 2014.

[17] M. Hodson, K.M. Zick, K. Jones, D. Padilha, “A novel embedding technique for optimization problems of fully-connected integer variables,” Proceedings, fourth conference in adiabatic quantum computing, June 2015.

[18] D. Padilha. “Solving NP-Hard problems on an adiabatic quantum computer.” UNSW School of Mechanical and Manufacturing Engineering, 2014.

[19] X. Peng et al, “Quantum adiabatic algorithm for factorization and its experimental implementation,” Physical review letters 101.22, 2008, p. 220405.

[20] International Organization for Standardization, "ISO/IEC 7498-4:1989 Information technology- Open Systems Interconnection basic reference model: Naming and addressing", 1989.