Targeting Classical Code to a Quantum Annealer

Scott Pakin
pakin@lanl.gov
Los Alamos National Laboratory
Los Alamos, NM, USA

Abstract

From a compiler's perspective, a *quantum annealer* represents a fundamentally different hardware target from a CPU, GPU, or other von Neumann architecture. Quantum annealers are special-purpose computers that use quantum effects to heuristically determine the set of Boolean variables that minimize a quadratic pseudo-Boolean function (an NP-hard problem). Natively programming such systems involves supplying them with a vector of function coefficients and receiving a vector of function-minimizing Booleans in return.

The contribution of this work is to demonstrate how to compile conventional code into a minimization problem for solution on a quantum annealer. The resulting code can run either forward (from inputs to outputs) or backward (from outputs to inputs). We show how this capability can be exploited to simplify the expression and solution of problems in the NP complexity class.

Keywords Verilog, quantum annealing, quantum computing, D-Wave, EDIF

ACM Reference Format:
Scott Pakin. 2019. Targeting Classical Code to a Quantum Annealer. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3297858.3304071

1 Introduction

A quantum computer exploits properties of quantum mechanics to solve computational problems, in some cases asymptotically faster than is possible on a conventional, classical computer. There are two main types of quantum computers in use today outside of research laboratories. *Gate-model quantum computers* support general-purpose quantum computation but are challenging to scale to large qubit (quantum bit) counts. At the time of this writing, the largest such system is Google's, which contains 72 qubits [18]. This paper focuses on the other main type of quantum computer, a *quantum annealer*, which is a special-purpose device. Quantum annealers are less well-studied than gate-model quantum computers and have not yet been shown capable of delivering asymptotic speedup over the best classical implementation of any real problem. However, they are a more scalable technology, with the largest instance at the time of this writing being D-Wave Systems's D-Wave 2000Q, with up to 2048 qubits [29]. We use this system for our experiments.

The one problem that a quantum annealer solves is finding a set of Boolean values that minimize a quadratic pseudo-Boolean function.[1] (Physicists call this a 2-local Ising-model Hamiltonian function.) Finding these values is an NP-hard problem [2], which implies that if quantum annealing *can* provide asymptotic speedup, all of the problems in the NP complexity class would see comparable speedups.

As with all new technologies, early programming models are low-level and correspond directly to the underlying hardware's capabilities. In the case of quantum annealers, this means that a computational problem must normally be expressed in terms of function minimization. This is clearly a substantially different way to reason about programming than that which is required for a conventional CPU or GPU.

To make quantum annealers more accessible to classically trained programmers, we ask the question, *Is it possible to target classical code to a quantum annealer?* That is, can we mechanically compile programs written to a classical programming model into the form of a quadratic pseudo-Boolean function such that the Boolean variables that minimize that function correspond to the program's output? In this paper, we argue that the answer is yes. Our evidence is a presentation of a compilation process that transforms code written in a (somewhat) conventional language through a series of successively lower-level abstractions until it takes the form of function minimization, with the results of program execution mapped back to the higher-level form expressed by the programmer.

This paper makes two key contributions. The first is the compiler technology that can be used to reduce classical code into an optimization problem for quantum-annealing solution. The second contribution is the placing of this technology in context, showing where it is likely to yield productivity benefits.

The rest of the paper is structured as follows. Section 2 provides the requisite background knowledge needed to understand the context of this work. Related work in programming tools for quantum computers is discussed in Section 3. In Section 4 we describe our approach in a top-down manner, from a programmer's view of a program down to the actual code that runs on the hardware. The implications of compiling conventional languages to a quantum annealer are discussed

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304071

[1] A pseudo-Boolean function is defined as being of type $\mathbb{B}^N \mapsto \mathbb{R}$.

in Section 5 through the presentation of a set of examples. In Section 6, one of these is analyzed in extra detail. Finally, we draw some conclusions from our work in Section 7.

2 Background

Quantum annealing [27, 28, 37] is a physical, hardware analogue of simulated annealing [40]. Both seek the input values that minimize a given function, but a quantum annealer can exploit *quantum tunneling* to find those values with higher probability than simulated annealing given the same search time [37].

A quantum annealer cannot minimize an arbitrary function but only those taking a particular form. The quantum annealers we consider in this paper, the ones for which scalable hardware exists, solve

$$\underset{\bar{\sigma}}{\operatorname{argmin}} \mathcal{H}(\bar{\sigma}) \tag{1}$$

where

$$\mathcal{H}(\bar{\sigma}) = \sum_{i=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i \sigma_j \,. \tag{2}$$

In that expression, each variable σ_i is a "physics Boolean", with False and True represented respectively by -1 and $+1$ rather than by 0 and 1. (0 and 1 can be used—and typically are in the operations-research community—but doing so would complicate much of this paper's exposition.) Each linear coefficient h_i and each quadratic coefficient $J_{i,j}$ is real-valued.

A program for a quantum annealer is nothing more than a set of h_i and $J_{i,j}$ coefficients. Programs have no inputs, but they output a set of σ_i values. Clearly, there is a huge semantic gap between conventional software and Equation (2), which is the topic we address in Section 4.

Quantum annealers work by representing each σ_i with a physical qubit. An h_i is specified as the application of an external field to the associated qubit, and a $J_{i,j}$ is specified by controlling an inter-qubit coupling strength. The annealing process starts by bombarding the system with a strong transverse field, which puts all qubits into an indeterminate state—a so-called *superposition* of 0 and 1. The transverse field is gradually lessened, thereby letting the qubits stabilize to either 0 or 1, subject to biases imposed by the external fields and couplers. The adiabatic theorem [9] describes how slowly the transverse field must be removed for the system to wind up in a ground state (i.e., a solution to Equation (1)) with near-certainty. Unfortunately, this rate is problem-dependent, which makes it difficult to reason about asymptotic performance in the general case.

A point to note is that there is nothing quantum about Equation (2), as the Hamiltonian contains only stoquastic terms [10]. A quantum annealer uses quantum effects *implicitly* to solve a *classical* problem. An implication is that the compilation approach we present in this paper is as applicable to classical annealers such as Hitachi's simulated quantum annealer [48], Hitachi's CMOS annealer [64], and Fujitsu's Digital Annealer [43, 60] as it is to quantum annealers. In fact, the generated $\mathcal{H}(\bar{\sigma})$ can be minimized in software on conventional computers using, e.g., simulating annealing [40] or a constraint-programming solver such as Chuffed [15]—or even minimized in software on gate-model quantum computers using, e.g., the quantum approximate optimization algorithm (QAOA) [26].

As an aside, it has been proven that hardware implementing an efficient solution to an enhanced, non-stoquastic, version of Equation (2),

$$\mathcal{H}(\bar{\sigma}) = \sum_{i=0}^{N-1} h_i \sigma_i^z + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i^z \sigma_j^z + \sum_{i=0}^{N-1} \Delta_i \sigma_i^x + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} K_{i,j} \sigma_i^x \sigma_j^x \,, \tag{3}$$

which includes linear and quadratic terms in both the σ^z and σ^x directions—roughly analogous to real and imaginary axes—*would* be more powerful than classical [1, 5, 45]. Specifically, minimizing Equation (3) is QMA-complete [5]. To date, no hardware natively implementing Equation (3) has been developed, however.

While Equations (1) and (2) represent the *logical* problem a quantum annealer solves, engineering reality imposes a set of constraints on the physical $\mathcal{H}(\bar{\sigma})$. D-Wave Systems's current product, the D-Wave 2000Q, provides a nominal 2048 qubits, although there is inevitably some drop-out. Hence, in Equation (2), $N \leq 2048$. We remind the reader that because *all* quantum computers return classical, Boolean values, the implication is that the entire D-Wave 2000Q has the equivalent of at most 2048 *bits* of memory. This constraint plays a role in our choice of language in Section 4.

Although, nominally, $h_i \in \mathbb{R}$ and $J_{i,j} \in \mathbb{R}$, engineering limitations restrict the actual range to $h_i \in [-2.0, 2.0]$ and $J_{i,j} \in [-2.0, 1.0]$. (The asymmetry in $J_{i,j}$ is due to the rf-SQUID physics of the inter-qubit couplers, which are tuned via a flux bias applied to a compound Josephson junction loop [34].) Furthermore, because the system is analog rather than digital, there is limited precision within each of those ranges. The user-specified annealing time ranges from 1–2000µs, which may be shorter than what the adiabatic theorem requires to minimize $\mathcal{H}(\bar{\sigma})$ with near-certainty.

The most severe hardware limitation in practice is that the on-chip network lacks all-to-all connectivity. The physical topology is called a *Chimera graph* and is a 2-D mesh of 8-qubit bipartite graphs, called *unit cells* [12]. A fragment of this topology is illustrated in Figure 1. Each of the four qubits in a horizontally oriented partition connects to its peer in the unit cells to the north and south, and each of the four qubits in a vertically oriented partition connects to its peer in the unit cells to the east and west. A D-Wave 2000Q is laid out as a C16 Chimera graph, which denotes a 16×16 mesh of unit cells.

As on a conventional network on chip, non-local communication requires routing. For example, the hardware does not accept a $J_{4,139}$ coefficient for Equation (2) so one might use coefficients $J_{4,12}$, $J_{12,11}$, and $J_{11,139}$ to compensate. As in an FPGA, the place-and-route step is performed statically at compile time. Unlike in the FPGA context, this step consumes not only couplers (≈ wires) but also qubits (≈ configurable logic blocks), both of which are scarce resources. In Section 5 we see how the sparseness of the topology affects the scale of problem that can be expressed.

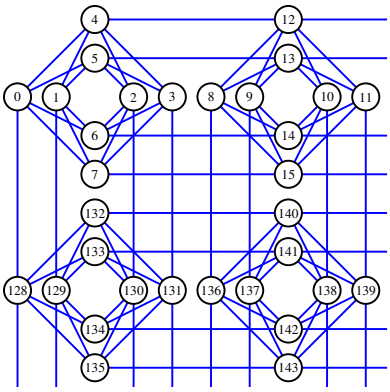


Figure 1. The upper-left 2×2 array of unit cells in a D-Wave 2000Q system

3 Related Work

The most related work to ours is QA Prolog [50]. That work is built on the same underlying software framework as we describe in this paper. However, QA Prolog targets a different end goal—fully parallel constraint logic programming—so the corresponding paper focuses more on that than on the description and application of the underlying software framework.

Most programming tools for D-Wave systems are substantially lower-level than what we discuss in this paper. For example, programs written in the qbsolv [23] and QMASM [49] languages are essentially just lists of h_i and $J_{i,j}$ coefficients for Equation (2) (with $\sigma_i \in \{0,1\}$ for qbsolv and either $\sigma_i \in \{0,1\}$ or $\sigma_i \in \{-1,+1\}$ for QMASM). ThreeQ.jl [47] improves slightly upon this model by embedding the construction of $\mathcal{H}(\bar{\sigma})$ into the Julia language [4]. Julia loops, conditionals, functions, and other higher-level constructs can be used to generate $\mathcal{H}(\bar{\sigma})$, although the core abstraction is no higher-level than that.

To the best of our knowledge, the situation is no different for classical annealers, such as Hitachi's simulated quantum annealer [48], Hitachi's CMOS annealer [64], and Fujitsu's Digital Annealer [43, 60]. All are also currently programmed by specifying lists of h_i and $J_{i,j}$ coefficients via a general-purpose-language library.

Due in part to the lack of higher-level tools, expressing computational problems as quadratic pseudo-Boolean functions is traditionally performed manually, often requiring a fair amount of ingenuity. Nevertheless, the research community has managed to express a large number of problems as quadratic pseudo-Boolean functions, as surveyed by Lucas [42] and Kochenberger et al. [41].

More recently, the D-Wave Ocean software suite [22] has incorporated a few higher-level programming abstractions, although, unlike in our work, these are not expressed as standalone programming languages. D-Wave NetworkX is a Python library based on the NetworkX [32] graph library. D-Wave NetworkX provides a set of functions for solving various NP-complete and NP-hard graph problems on a D-Wave system. One can conceive of the D-Wave NetworkX programming model as comprising the transformation of a problem into one of the provided algorithms, execution of the code on a D-Wave system, and transformation of the solution back to a solution to the original problem. D-Wave Binary CSP, also a Python library, lets programmers reason about programming in terms of constraint-satisfaction problems. The programmer specifies constraints on a set of Boolean variables, and the tool compiles these into an $\mathcal{H}(\bar{\sigma})$ for D-Wave satisfaction.

Gate-model quantum computers offer a fundamentally different native programming model to that offered by quantum annealers. To date, most programming tools for gate-model quantum computers also present the programmer with only a low level of abstraction. This low-level abstraction—partially ordering a set of quantum gates on a timeline—is sometimes embodied in a standalone programming language, as in Quil [57], Q# [44], and Scaffold [36], and is sometimes implemented as an embedded domain-specific language, as in Quipper [31] (Haskell), LIQ$Ui|\rangle$ [61] (F#), and Cirq [59], ProjectQ [58], and pyQuil [57] (Python). Some of the preceding language environments include implementations of various building-block algorithms, making these tools a bit higher-level, in the sense of D-Wave NetworkX, as described above. Qumin [56] is arguably one of the few exceptions to the low-level-abstraction rule. In addition to gate-by-gate placement in a partial time ordering, it also lets programmers use functional-language constructs to define gate-model quantum programs in mathematical terms.

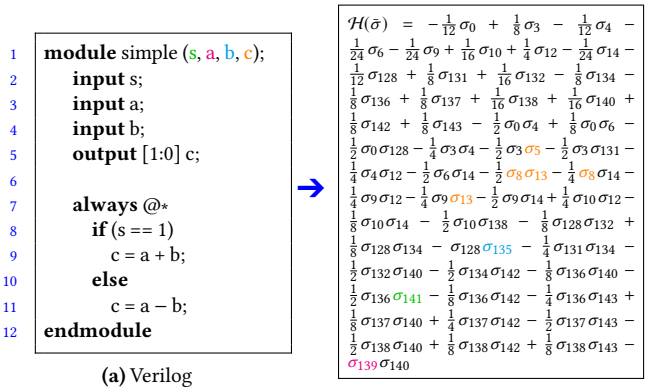
There exists work in the program-verification community, specifically in bounded model checking (BMC) [7], for compiling classical code to satisfiability (SAT) problems. However, while BMC is similar on the surface to what we outline in Section 4 in that both techniques generate Boolean expressions, the fundamental difference is that our approach involves lowering code all the way to bitwise operations while BMC tools lower code only as far as mechanically verifiable expressions, such as $x_4 \leq 3$ [16, 17], which is too high-level to directly produce $\mathcal{H}(\bar{\sigma})$.

4 Approach

It may seem like a formidable task to compile classical code to the form of Equations (1) and (2). In this section, we describe our methods and techniques in top-down fashion, from the highest level of abstraction (classical code) to the lowest level of abstraction (a hardware-specific instantiation of $\mathcal{H}(\bar{\sigma})$).

4.1 Inputting classical code

The choice of programming language has substantial ramifications on the ease with which a compiler can translate programs into a quadratic pseudo-Boolean function. After careful consideration we settled on Verilog [35]. Verilog is a hardware-description language, normally used for the design of digital circuits. While Verilog may seem like an odd choice of language for programming a quantum annealer, it has two key characteristics that make it a suitable source language:



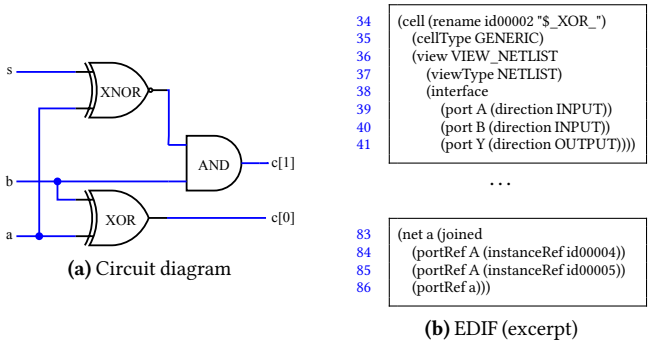
```
1  module simple (s, a, b, c);
2      input s;
3      input a;
4      input b;
5      output [1:0] c;
6
7      always @*
8        if (s == 1)
9          c = a + b;
10       else
11         c = a - b;
12 endmodule
```

(a) Verilog

$$\mathcal{H}(\bar{\sigma}) = -\tfrac{1}{12}\sigma_0 + \tfrac{1}{8}\sigma_3 - \tfrac{1}{12}\sigma_4 - \tfrac{1}{24}\sigma_6 - \tfrac{1}{24}\sigma_9 + \tfrac{1}{16}\sigma_{10} + \tfrac{1}{4}\sigma_{12} - \tfrac{1}{24}\sigma_{14} - \tfrac{1}{12}\sigma_{128} + \tfrac{1}{8}\sigma_{131} + \tfrac{1}{16}\sigma_{132} - \tfrac{1}{8}\sigma_{134} - \tfrac{1}{8}\sigma_{136} + \tfrac{1}{8}\sigma_{137} + \tfrac{1}{16}\sigma_{138} + \tfrac{1}{16}\sigma_{140} + \tfrac{1}{8}\sigma_{142} + \tfrac{1}{8}\sigma_{143} - \tfrac{1}{2}\sigma_0\sigma_4 + \tfrac{1}{8}\sigma_0\sigma_6 - \tfrac{1}{2}\sigma_0\sigma_{128} - \tfrac{1}{4}\sigma_3\sigma_4 - \tfrac{1}{2}\sigma_3\sigma_5 - \tfrac{1}{2}\sigma_3\sigma_{131} - \tfrac{1}{4}\sigma_4\sigma_{12} - \tfrac{1}{2}\sigma_6\sigma_{14} - \tfrac{1}{2}\sigma_8\sigma_{13} - \tfrac{1}{4}\sigma_8\sigma_{14} - \tfrac{1}{4}\sigma_9\sigma_{12} - \tfrac{1}{4}\sigma_9\sigma_{13} - \tfrac{1}{2}\sigma_9\sigma_{14} + \tfrac{1}{4}\sigma_{10}\sigma_{12} - \tfrac{1}{8}\sigma_{10}\sigma_{14} - \tfrac{1}{2}\sigma_{10}\sigma_{138} - \tfrac{1}{8}\sigma_{128}\sigma_{132} + \tfrac{1}{8}\sigma_{128}\sigma_{134} - \sigma_{128}\sigma_{135} - \tfrac{1}{4}\sigma_{131}\sigma_{134} - \tfrac{1}{2}\sigma_{132}\sigma_{140} - \tfrac{1}{2}\sigma_{134}\sigma_{142} - \tfrac{1}{8}\sigma_{136}\sigma_{140} - \tfrac{1}{2}\sigma_{136}\sigma_{141} - \tfrac{1}{8}\sigma_{136}\sigma_{142} - \tfrac{1}{4}\sigma_{136}\sigma_{143} + \tfrac{1}{8}\sigma_{137}\sigma_{140} + \tfrac{1}{4}\sigma_{137}\sigma_{142} - \tfrac{1}{2}\sigma_{137}\sigma_{143} - \tfrac{1}{2}\sigma_{138}\sigma_{140} + \tfrac{1}{8}\sigma_{138}\sigma_{142} + \tfrac{1}{8}\sigma_{138}\sigma_{143} - \sigma_{139}\sigma_{140}$$

(b) Physical Hamiltonian function

Figure 2. End-to-end transformation of a simple function



(a) Circuit diagram

(b) EDIF (excerpt)

Figure 3. Digital circuit corresponding to Figure 2(a)

1. It provides precise control over bit widths. With current quantum annealers providing on the order of 2000 qubits [29]—wasting qubits would be unacceptable.
2. It can be compiled to a small set of primitives such as Boolean operators, multiplexers, and flip-flops. We will see shortly why this is helpful when targeting a quantum annealer.

However, Verilog also exhibits a couple of important shortcomings:

1. Its hardware-centric semantics feel awkward to programmers accustomed to "conventional" languages such as Python, C++, Java, etc.
2. It has limited support for data structures (e.g., arrays and records), floating-point arithmetic, recursion, and loops whose trip count is unknown at compile time.

That said, Verilog does support multi-bit arithmetic and relational operators, conditionals, functions, modules, and other near-necessities.

As a simple example to make the goal of this work more concrete, Figure 2(a) presents a (not particularly useful) Verilog program that inputs three 1-bit variables s, a, and b and outputs a 2-bit variable c, which is assigned $a + b$ if s is 1 and $a - b$ if s is 0. Figure 2(b) shows one possible compilation of that code to a hardware-specific, quadratic pseudo-Boolean function. In this formulation of $\mathcal{H}(\bar{\sigma})$, variable s is mapped to physical qubit 141 (i.e., σ_{141} in the figure), a to physical qubit 139, b to 135, and c to 5 (high-order bit) and both 8 and 13 (low-order bit). $\mathcal{H}(\bar{\sigma})$ is minimized exactly when s, a, b, and c, correspond to a valid relation of inputs and outputs, for example at $\{s=0, a=1, b=0, c=01\}$ or $\{s=1, a=1, b=1, c=10\}$ but not at $\{s=1, a=0, b=0, c=11\}$. In Section 4.3.6 we will see how to pass arguments to $\mathcal{H}(\bar{\sigma})$.

4.2 Lowering Verilog to EDIF

The first step in our compilation process is to compile Verilog code to a digital circuit. We use Yosys [63] as our hardware-synthesis tool (i.e., a compiler for hardware) with ABC [3, 11] providing additional code optimizations. Together, these tools compile Verilog source such as that shown in Figure 2(a) into a digital circuit such as that shown in Figure 3(a).

More precisely, we instruct Yosys to generate a machine-readable *netlist* for the next step of our compilation process. A netlist is a precise specification of gates and the wires that connect them. We specify EDIF (Electronic Design Interchange Format) [38] as the netlist format for Yosys to output. An EDIF netlist is represented by a single, large s-expression [54], which makes it easy to parse mechanically. Figure 3(b) shows an excerpt of the 112-line EDIF netlist corresponding to our simple circuit. The first stanza of the excerpt indicates that an XOR gate provides inputs A and B and output Y. The second stanza indicates that the circuit's input port a fans out to gate id00004's input A and gate id00005's input A.

4.3 Lowering EDIF to QMASM

The next step of the compilation process is the most critical as well as the most specific to our goal of targeting classical code to a quantum annealer. This is where we compile EDIF code into a *logical* quadratic pseudo-Boolean function. That is, we generate a function of the form shown in Equation (2) but with no restrictions on the ranges of the h_i and $J_{i,j}$ coefficients and no restrictions on the pairs of variables that can appear in a quadratic term.

Our approach is to target QMASM [49], which is a "quantum macro assembler" in the sense that it provides small but important programming conveniences over the raw hardware model. Just as it is more convenient to express an x86 addition instruction symbolically as `addl %esi, %eax` than as the binary `0000 0001 1111 0000`, QMASM lets programmers write functions symbolically, as in Listing 1, as opposed to numerically, as in $\frac{1}{10}\sigma_0 - \frac{1}{40}\sigma_2 - \frac{1}{40}\sigma_4 - \frac{1}{4}\sigma_0\sigma_4 + \frac{1}{2}\sigma_0\sigma_5 - \frac{1}{4}\sigma_0\sigma_7 - \frac{1}{8}\sigma_1\sigma_4 - \sigma_1\sigma_5 - \frac{1}{4}\sigma_1\sigma_7 - \sigma_2\sigma_4 - \frac{1}{8}\sigma_2\sigma_5 + \frac{1}{2}\sigma_2\sigma_7$, with all coefficient limitations manually accounted for in that formulation.

Listing 1. Sample QMASM code

```
1  A      -1
2  D       2
3  A B    -5
4  B C    -5
5  C D    -5
6  D A    -5
7  A C    10
8  B D    10
```

Some relevant QMASM language features are that it

- lets programs refer to variables symbolically,
- accepts arbitrary values for the function coefficients and automatically maps those onto what is accepted by the underlying hardware,
- provides shortcut syntax for biasing two variables to have the same value (or, respectively, the opposite value) in the solution to Equation (1),
- supports macros to facilitate code reuse, and
- allows sets of macros to appear in a separate file that can be included into a main routine.

In addition, as a tool, `qmasm`

- can not only execute programs directly on a D-Wave quantum annealer but can also convert them to various other formats for classical solution (e.g., a constraint problem for solution with MiniZinc [46]), or run them indirectly through qbsolv [8, 21], which can split large problems into sub-problems that fit on the D-Wave hardware,
- reports the solution to Equation (1) in terms of the program-specified symbolic names rather than as physical qubit numbers,
- accepts a command-line option to bias specified variables toward True or False, and
- can run a program arbitrarily many times and report statistics on the results—important because all quantum computers are fundamentally stochastic.

The challenge we face in this step of the compilation process is how to convert an EDIF netlist to QMASM code. Our approach involves establishing a mapping from each gate type that can appear in a netlist to a relatively small quadratic pseudo-Boolean function, which is expressed as a QMASM macro. These are instantiated for each *cell* (gate) specified by the netlist. A *net* (wire) between cells is expressed as a bias for the two connected variables to have the same value. We build up from those steps to a complete digital circuit. The rest of Section 4.3 explains how each of the transformation steps is accomplished.

4.3.1 Nets

One can think of a net that connects one cell's output Y to another cell's input A as an assertion that $A = Y$. Letting $\bar{\sigma}$ be the two-variable vector $[\sigma_A\, \sigma_Y]$, Equation (2) expands to $\mathcal{H}(\sigma_A,\sigma_Y) = h_A\sigma_A + h_Y\sigma_Y + J_{A,Y}\sigma_A\sigma_Y$. As Table 1 indicates, assigning $h_A = h_Y = 0$ and $J_{A,Y} = -1$ ensures that $\mathcal{H}(\sigma_A,\sigma_Y)$ is minimized exactly where $\sigma_A = \sigma_Y$.[2] Remember, as per Equation (1), a quantum annealer will settle into a state that assigns variables whatever ±1 values minimize $\mathcal{H}(\bar{\sigma})$.

Larger nets follow the same pattern as nets that connect two endpoints. For example, a net that represents a fan-out of one cell's output Y to other cells' inputs A, B, C, and D can be expressed as $\mathcal{H}(\sigma_Y,\sigma_A,\sigma_B,\sigma_C,\sigma_D) = -\sigma_Y\sigma_A - \sigma_Y\sigma_B - \sigma_Y\sigma_C - \sigma_Y\sigma_D$.

[2] In fact, any negative number will suffice, but recall that all coefficients must be scaled to the range supported by the hardware and that precision is limited (see Section 2).

Table 1. A two-ended net expressed as a quadratic pseudo-Boolean function

σ_A	σ_Y	$-\sigma_A\sigma_Y$	Min.?
−1	−1	−1	✓
−1	+1	+1	
+1	−1	+1	
+1	+1	−1	✓

4.3.2 Cells

A quantum-annealing version of a cell is a quadratic pseudo-Boolean function that is minimized exactly when provided a valid relation of cell inputs and outputs. For example, we want to compile an and gate, representing $Y = A \wedge B$, to a quadratic pseudo-Boolean function $\mathcal{H}_\wedge(\sigma_Y,\sigma_A,\sigma_B)$ such that $\mathcal{H}_\wedge(-1,-1,-1) = \mathcal{H}_\wedge(-1,-1,+1) = \mathcal{H}_\wedge(-1,+1,-1) = \mathcal{H}_\wedge(+1,+1,+1)$ and that this value is strictly less than what would arise by passing $\mathcal{H}_\wedge$ any other arguments.

With the three arguments σ_Y, σ_A, and σ_B, Equation (2) expands to $\mathcal{H}_\wedge(\sigma_Y,\sigma_A,\sigma_B) = h_Y\sigma_Y + h_A\sigma_A + h_B\sigma_B + J_{Y,A}\sigma_Y\sigma_A + J_{Y,B}\sigma_Y\sigma_B + J_{A,B}\sigma_A\sigma_B$. The challenge is to find suitable values of the h and J coefficients to enforce the properties we require of $\mathcal{H}_\wedge$. Our approach is to set up and solve a system of inequalities (using, e.g., MiniZinc [46]), as shown in the first five columns of Table 2. Table 2 presents a complete three-column truth table, with $\mathcal{H}_\wedge(\sigma_Y,\sigma_A,\sigma_B)$ evaluated for each row. This leads to an expression in terms of only the h and J coefficients. We constrain the expression to equal k, the as-yet-unknown minimum value of $\mathcal{H}_\wedge$, on all valid rows of an and truth table and to be greater than k on all invalid rows. In this case, one solution—there are infinitely many—is $\mathcal{H}_\wedge(\sigma_Y,\sigma_A,\sigma_B) = 2\sigma_Y - \sigma_A - \sigma_B - 2\sigma_Y\sigma_A - 2\sigma_Y\sigma_B + \sigma_A\sigma_B$, with $k = -3$. The sixth column of Table 2, labeled *Example*, lists what this solution function evaluates to for each set of σ_Y, σ_A, and σ_B arguments.

Unfortunately, this approach does not always succeed. Considering only 2-input, 1-output Boolean functions, there are eight inequalities applied to only six variables, leading to a potentially overconstrained system. Fortunately, of the 16 possible 2-input, 1-output Boolean functions, only xor and xnor lead to an unsolvable system of inequalities [62]. This does not imply that xor and xnor cannot be implemented. A technique for making unsolvable systems solvable is to introduce ancilla variables. The extra columns these add to the truth table can provide sufficient additional degrees of freedom to make an unsolvable system of inequalities solvable.

In the case of xor and xnor a single ancilla suffices. Table 3 presents one of the eight possible ways to augment the truth table for xor to make the corresponding system of inequalities solvable. One solution, with σ_a representing the

Table 2. System of inequalities corresponding to an AND gate ($Y = A \wedge B$)

σ_Y	σ_A	σ_B	$\mathcal{H}_\wedge(\sigma_Y,\sigma_A,\sigma_B)$	Constraint	*Example*
−1	−1	−1	$-h_Y-h_A-h_B+J_{Y,A}+J_{Y,B}+J_{A,B}$	$=k$	−3
−1	−1	+1	$-h_Y-h_A+h_B+J_{Y,A}-J_{Y,B}-J_{A,B}$	$=k$	−3
−1	+1	−1	$-h_Y+h_A-h_B-J_{Y,A}+J_{Y,B}-J_{A,B}$	$=k$	−3
−1	+1	+1	$-h_Y+h_A+h_B-J_{Y,A}-J_{Y,B}+J_{A,B}$	$>k$	1
+1	−1	−1	$+h_Y-h_A-h_B-J_{Y,A}-J_{Y,B}+J_{A,B}$	$>k$	9
+1	−1	+1	$+h_Y-h_A+h_B-J_{Y,A}+J_{Y,B}-J_{A,B}$	$>k$	1
+1	+1	−1	$+h_Y+h_A-h_B+J_{Y,A}-J_{Y,B}-J_{A,B}$	$>k$	1
+1	+1	+1	$+h_Y+h_A+h_B+J_{Y,A}+J_{Y,B}+J_{A,B}$	$=k$	−3

ancilla variable, is $\mathcal{H}_\oplus(\sigma_Y,\sigma_A,\sigma_B,\sigma_a) = -\sigma_Y+\sigma_A-\sigma_B+2\sigma_a-\sigma_Y\sigma_A+\sigma_Y\sigma_B-2\sigma_Y\sigma_a-\sigma_A\sigma_B+2\sigma_A\sigma_a-2\sigma_B\sigma_a$. Here, $k=-4$. Table 4 demonstrates the correctness of this solution. Note that the number of rows considered valid (4) is left unchanged from the non-augmented truth table for XOR while the number of invalid rows increases. This is key to making the system of inequalities solvable.

Table 3. Augmented truth table

Y	A	B	a
F	F	F	F
T	F	T	T
T	T	F	F
F	T	T	F

Efficiently (i.e., in polynomial time) determining the minimum number of ancilla variables that are required and how to populate the extra columns in the truth table is an unsolved problem in mathematics. The most sophisticated technique devised to date reduces the base of the exponential term to the point that functions of up to around 10 variables can be computed in reasonable time on modern hardware [6].

We have computed quadratic pseudo-Boolean functions for a number of common gates, as shown in Table 5. The particular gates chosen for inclusion in the table correspond to the set of gates considered by default by the ABC optimizer [3, 11], but other gates can easily be implemented and utilized. Doing so can reduce the required qubit count at the expense of increased compilation time. As stated above, there are infinitely many ways to map a given truth table to a quadratic pseudo-Boolean function. The specific functions listed in Table 5 were chosen to honor the hardware-imposed coefficient ranges while maximizing the gap between the $\mathcal{H}(\bar{\sigma})$ of all valid inputs and the minimal $\mathcal{H}(\bar{\sigma})$ of an invalid input. Empirically, this tends to lead to more robust output on D-Wave hardware. Also note that some of the quadratic pseudo-Boolean functions listed in the table require more than one ancilla variable.

We implement all of the functions appearing in Table 5 as QMASM macros, which we store in a "standard cell library", `stdcell.qmasm`, that can be incorporated (with QMASM's **!include** directive) into the code our compiler framework generates. Listing 2 presents an excerpt from `stdcell.qmasm` that includes the definition of a `NOT` macro and an `OR` macro. The file includes niceties such as assertions, which aid debugging, and comments ("#"), for human-readable macro descriptions.

Listing 2. Excerpt from our QMASM standard-cell library

```
1  # Y = NOT A
2  !begin_macro NOT
3  !assert Y = !A
4     A Y 1.0
5  !end_macro NOT
6
7  # Y = A OR B
8  !begin_macro OR
9  !assert Y = A|B
10    A 0.5
11    B 0.5
12    Y -1
13
14    A B 0.5
15    A Y -1
16    B Y -1
17 !end_macro OR
```

Listing 3. Verilog code employing sequential logic

```
1  module count (clk, inc, reset, out);
2     input clk;
3     input inc;
4     input reset;
5     output [5:0] out;
6     reg [5:0] var;
7
8     always @(posedge clk)
9        if (reset)
10          var <= 0;
11       else
12          if (inc)
13             var <= var + 1;
14    assign out = var;
15 endmodule
```

4.3.3 Sequential logic

We have thus far described the building blocks needed to compile combinational logic to a quadratic pseudo-Boolean function. However, it is possible, with some caveats, also to compile sequential logic. Consider the Verilog code shown in Listing 3, which resets an internal 6-bit variable `var` on any clock cycle in which `reset` is 1 and increments it on any cycle in which `inc` is 1. The challenge is that Listing 3 is stateful while our compilation target, Equation (2), is a pure function.

The solution we employ in our compiler framework is to statically unroll the code, replicating the entire program for each time step—entire program at time 0, entire program at time 1, and so forth up to a user-specified final time—with the outputs of one time step serving as the inputs to the

Table 4. System of inequalities corresponding to an XOR gate ($Y = A \oplus B$)

σ_Y	σ_A	σ_B	σ_a	$\mathcal{H}_\oplus(\sigma_Y,\sigma_A,\sigma_B,\sigma_a)$	Constraint	*Example*
−1	−1	−1	−1	$-h_Y-h_A-h_B-h_a+J_{Y,A}+J_{Y,B}+J_{Y,a}+J_{A,B}+J_{A,a}+J_{B,a}$	$=k$	−4
−1	−1	−1	+1	$-h_Y-h_A-h_B+h_a+J_{Y,A}+J_{Y,B}-J_{Y,a}+J_{A,B}-J_{A,a}-J_{B,a}$	$>k$	4
−1	−1	+1	−1	$-h_Y-h_A+h_B-h_a+J_{Y,A}-J_{Y,B}+J_{Y,a}-J_{A,B}+J_{A,a}-J_{B,a}$	$>k$	−2
−1	−1	+1	+1	$-h_Y-h_A+h_B+h_a+J_{Y,A}-J_{Y,B}-J_{Y,a}-J_{A,B}-J_{A,a}+J_{B,a}$	$>k$	−2
−1	+1	−1	−1	$-h_Y+h_A-h_B-h_a-J_{Y,A}+J_{Y,B}+J_{Y,a}-J_{A,B}-J_{A,a}+J_{B,a}$	$>k$	−2
−1	+1	−1	+1	$-h_Y+h_A-h_B+h_a-J_{Y,A}+J_{Y,B}-J_{Y,a}-J_{A,B}+J_{A,a}-J_{B,a}$	$>k$	14
−1	+1	+1	−1	$-h_Y+h_A+h_B-h_a-J_{Y,A}-J_{Y,B}+J_{Y,a}+J_{A,B}-J_{A,a}-J_{B,a}$	$=k$	−4
−1	+1	+1	+1	$-h_Y+h_A+h_B+h_a-J_{Y,A}-J_{Y,B}-J_{Y,a}+J_{A,B}+J_{A,a}+J_{B,a}$	$>k$	4
+1	−1	−1	−1	$+h_Y-h_A-h_B-h_a-J_{Y,A}-J_{Y,B}-J_{Y,a}+J_{A,B}+J_{A,a}+J_{B,a}$	$>k$	−2
+1	−1	−1	+1	$+h_Y-h_A-h_B+h_a-J_{Y,A}-J_{Y,B}+J_{Y,a}+J_{A,B}-J_{A,a}-J_{B,a}$	$>k$	−2
+1	−1	+1	−1	$+h_Y-h_A+h_B-h_a-J_{Y,A}+J_{Y,B}-J_{Y,a}-J_{A,B}+J_{A,a}-J_{B,a}$	$>k$	4
+1	−1	+1	+1	$+h_Y-h_A+h_B+h_a-J_{Y,A}+J_{Y,B}+J_{Y,a}-J_{A,B}-J_{A,a}+J_{B,a}$	$=k$	−4
+1	+1	−1	−1	$+h_Y+h_A-h_B-h_a+J_{Y,A}-J_{Y,B}-J_{Y,a}-J_{A,B}-J_{A,a}+J_{B,a}$	$=k$	−4
+1	+1	−1	+1	$+h_Y+h_A-h_B+h_a+J_{Y,A}-J_{Y,B}+J_{Y,a}-J_{A,B}+J_{A,a}-J_{B,a}$	$>k$	4
+1	+1	+1	−1	$+h_Y+h_A+h_B-h_a+J_{Y,A}+J_{Y,B}-J_{Y,a}+J_{A,B}-J_{A,a}-J_{B,a}$	$>k$	−2
+1	+1	+1	+1	$+h_Y+h_A+h_B+h_a+J_{Y,A}+J_{Y,B}+J_{Y,a}+J_{A,B}+J_{A,a}+J_{B,a}$	$>k$	−2

subsequent time step. This is implemented by having a D flip-flop (single-bit state) that is instantiated at time t forward its Q output to the D input of the same flip-flop instantiated at time $t+1$. That is, $\mathcal{H}_{\text{DFF}}(\sigma_Q,\sigma_D) = -\sigma_Q\sigma_D$, where σ_Q and σ_D are associated with different time steps. Because our notion of time is discrete, clock edges are ignored, and a D is *always* propagated to the subsequent time step's Q.

In essence, we are trading the program's time dimension for a second spatial dimension. Doing so exacts a heavy toll in qubit count, making stateful programs of even modest size impractical for current, qubit-limited quantum annealers. It does demonstrate, however, that given a suitable upper bound on execution time (number of steps or clock cycles) and a sufficient number of qubits, *any* classical Verilog program can be compiled and run using our compilation framework.

4.3.4 Ground and power

Connecting a variable to ground or power is implemented using a pair of trivial quadratic pseudo-Boolean functions. Ground is represented by $\mathcal{H}_{\text{GND}}(\sigma_A) = \sigma_A$, which is minimized when σ_A is FALSE (−1), and power is represented by $\mathcal{H}_{\text{Vcc}}(\sigma_A) = -\sigma_A$, which is minimized when σ_A is TRUE (+1). As with nets (Section 4.3.1), the coefficient can have any magnitude; only the sign matters.

4.3.5 Putting the pieces together

If quadratic pseudo-Boolean function $\mathcal{H}_P$ is minimized by input vectors $\bar{\sigma_x}$, $\bar{\sigma_y}$, and $\bar{\sigma_z}$, and quadratic pseudo-Boolean function $\mathcal{H}_Q$ is minimized by input vectors $\bar{\sigma_w}$, $\bar{\sigma_x}$, and $\bar{\sigma_y}$, then quadratic pseudo-Boolean function $\mathcal{H}_P+\mathcal{H}_Q$ is minimized by their intersection, $\bar{\sigma_x}$ and $\bar{\sigma_y}$. If the intersection is empty, it can be hard to predict what input vectors will minimize $\mathcal{H}_P+\mathcal{H}_Q$. However, by construction, this case does not arise in the quadratic pseudo-Boolean functions produced by our compiler.

To make this assertion more concrete, consider building a 3-input AND gate out of two 2-input AND gates with a wire connecting the output of the first gate to one of the inputs of the second gate. Taking the definition of a 2-input AND from Table 5 ($\mathcal{H}_\wedge$) and the definition of a wire from Section 4.3.1, we can easily produce $\mathcal{H}_{\text{AND}_3}$:

$$\begin{array}{lll} \mathcal{H}_\wedge(\sigma_Y,\sigma_m,\sigma_C) & \rightarrow & Y = m \wedge C \\ \mathcal{H}_\wedge(\sigma_n,\sigma_A,\sigma_B) & \rightarrow & n = A \wedge B \\ + \quad -\sigma_m\sigma_n & \rightarrow & m = n \\ \hline \mathcal{H}_{\text{AND}_3}(\sigma_Y,\sigma_A,\sigma_B,\sigma_C) & \rightarrow & Y = A \wedge B \wedge C \end{array}$$

That summation can be expressed in QMASM as shown in Listing 4. In QMASM, "`A = B`" is shortcut notation for $J_{A,B}\sigma_A\sigma_B$, where $J_{A,B}$ is a negative number defined externally to the code. (It can be specified on the `qmasm` command line and defaults to a magnitude of twice the largest-in-magnitude $J_{i,j}$ value that appears literally in the code.) A "`$`" appearing anywhere in a variable name indicates an "internal" or "uninteresting" variable. `qmasm` does not by default output the value of such variables. After instantiating `AND3` with, e.g., `!use_macro AND3 my_and`, one can refer to `my_and.A`, `my_and.B`, `my_and.C`, and `my_and.Y` in larger expressions.

Listing 4. A 3-input AND expressed in QMASM

```
1  !include <stdcell>
2  !begin_macro AND3
3    !use_macro AND $and1
4    !use_macro AND $and2
5    $and1.Y = Y
6    $and1.A = $and2.Y
7    $and1.B = C
8    $and2.A = A
9    $and2.B = B
10 !end_macro AND3
```

Table 5. A standard-cell library, with cells expressed as quadratic pseudo-Boolean functions

Cell	Logic	Quadratic pseudo-Boolean function representation
NOT (inverter)	$Y = \neg A$	$\mathcal{H}_{\neg}(\sigma_Y,\sigma_A) = \sigma_A\sigma_Y$
AND	$Y = A \wedge B$	$\mathcal{H}_{\wedge}(\sigma_Y,\sigma_A,\sigma_B) = -\frac{1}{2}\sigma_A - \frac{1}{2}\sigma_B + \sigma_Y + \frac{1}{2}\sigma_A\sigma_B - \sigma_A\sigma_Y - \sigma_B\sigma_Y$
OR	$Y = A \vee B$	$\mathcal{H}_{\vee}(\sigma_Y,\sigma_A,\sigma_B) = \frac{1}{2}\sigma_A + \frac{1}{2}\sigma_B - \sigma_Y + \frac{1}{2}\sigma_A\sigma_B - \sigma_A\sigma_Y - \sigma_B\sigma_Y$
NAND	$Y = A \uparrow B$	$\mathcal{H}_{\uparrow}(\sigma_Y,\sigma_A,\sigma_B) = -\frac{1}{2}\sigma_A - \frac{1}{2}\sigma_B - \sigma_Y + \frac{1}{2}\sigma_A\sigma_B + \sigma_A\sigma_Y + \sigma_B\sigma_Y$
NOR	$Y = A \downarrow B$	$\mathcal{H}_{\downarrow}(\sigma_Y,\sigma_A,\sigma_B) = \frac{1}{2}\sigma_A + \frac{1}{2}\sigma_B + \sigma_Y + \frac{1}{2}\sigma_A\sigma_B + \sigma_A\sigma_Y + \sigma_B\sigma_Y$
XOR	$Y = A \oplus B$	$\mathcal{H}_{\oplus}(\sigma_Y,\sigma_A,\sigma_B,\sigma_a) = \frac{1}{2}\sigma_A - \frac{1}{2}\sigma_B - \frac{1}{2}\sigma_Y + \sigma_a - \frac{1}{2}\sigma_A\sigma_B - \frac{1}{2}\sigma_A\sigma_Y + \sigma_A\sigma_a + \frac{1}{2}\sigma_B\sigma_Y - \sigma_B\sigma_a - \sigma_Y\sigma_a$
XNOR	$Y = A \leftrightarrow B$	$\mathcal{H}_{\leftrightarrow}(\sigma_Y,\sigma_A,\sigma_B,\sigma_a) = \frac{1}{2}\sigma_A - \frac{1}{2}\sigma_B + \frac{1}{2}\sigma_Y + \sigma_a - \frac{1}{2}\sigma_A\sigma_B + \frac{1}{2}\sigma_A\sigma_Y + \sigma_A\sigma_a - \frac{1}{2}\sigma_B\sigma_Y - \sigma_B\sigma_a + \sigma_Y\sigma_a$
2:1 MUX (multiplexer)	$Y = (S \wedge B) \vee (\neg S \wedge A)$	$\mathcal{H}_{\text{MUX}}(\sigma_Y,\sigma_S,\sigma_A,\sigma_B,\sigma_a) = \frac{1}{2}\sigma_S + \frac{1}{4}\sigma_A - \frac{1}{4}\sigma_B + \frac{1}{2}\sigma_Y + \sigma_a + \frac{1}{4}\sigma_S\sigma_A - \frac{1}{4}\sigma_S\sigma_B + \frac{1}{2}\sigma_S\sigma_Y + \sigma_S\sigma_a + \frac{1}{2}\sigma_A\sigma_B - \frac{1}{2}\sigma_A\sigma_Y + \frac{1}{2}\sigma_A\sigma_a - \sigma_B\sigma_Y - \frac{1}{2}\sigma_B\sigma_a + \sigma_Y\sigma_a$
AOI3 (3-bit AND-OR-inverter)	$Y = \neg((A \wedge B) \vee C)$	$\mathcal{H}_{\text{AOI3}}(\sigma_Y,\sigma_A,\sigma_B,\sigma_C,\sigma_a) = -\frac{1}{3}\sigma_B + \frac{1}{3}\sigma_C + \frac{2}{3}\sigma_Y - \frac{2}{3}\sigma_a + \frac{1}{3}\sigma_A\sigma_B + \frac{1}{3}\sigma_A\sigma_C + \frac{1}{3}\sigma_A\sigma_Y + \frac{1}{3}\sigma_A\sigma_a - \frac{1}{3}\sigma_B\sigma_Y + \sigma_B\sigma_a + \sigma_C\sigma_Y - \frac{1}{3}\sigma_C\sigma_a - \sigma_Y\sigma_a$
OAI3 (3-bit OR-AND-inverter)	$Y = \neg((A \vee B) \wedge C)$	$\mathcal{H}_{\text{OAI3}}(\sigma_Y,\sigma_A,\sigma_B,\sigma_C,\sigma_a) = -\frac{1}{4}\sigma_A - \frac{3}{4}\sigma_C - \frac{1}{2}\sigma_Y - \frac{1}{2}\sigma_a + \frac{3}{4}\sigma_A\sigma_C + \frac{1}{2}\sigma_A\sigma_Y + \frac{1}{2}\sigma_A\sigma_a + \frac{1}{4}\sigma_B\sigma_Y - \frac{1}{4}\sigma_B\sigma_a + \sigma_C\sigma_Y + \sigma_C\sigma_a + \frac{1}{4}\sigma_Y\sigma_a$
AOI4 (4-bit AND-OR-inverter)	$Y = \neg((A \wedge B) \vee (C \wedge D))$	$\mathcal{H}_{\text{AOI4}}(\sigma_Y,\sigma_A,\sigma_B,\sigma_C,\sigma_a,\sigma_b) = -\frac{1}{6}\sigma_A - \frac{1}{6}\sigma_B - \frac{5}{12}\sigma_C + \frac{1}{4}\sigma_D - \frac{5}{12}\sigma_Y - \frac{7}{12}\sigma_a + \frac{1}{6}\sigma_b + \frac{1}{6}\sigma_A\sigma_B + \frac{1}{3}\sigma_A\sigma_C - \frac{1}{12}\sigma_A\sigma_D + \frac{1}{2}\sigma_A\sigma_Y + \frac{1}{3}\sigma_A\sigma_a - \frac{1}{4}\sigma_A\sigma_b + \frac{1}{3}\sigma_B\sigma_C - \frac{1}{12}\sigma_B\sigma_D + \frac{1}{2}\sigma_B\sigma_Y + \frac{1}{3}\sigma_B\sigma_a - \frac{1}{4}\sigma_B\sigma_b - \frac{1}{3}\sigma_C\sigma_D + \frac{11}{12}\sigma_C\sigma_Y + \frac{11}{12}\sigma_C\sigma_a - \frac{5}{12}\sigma_C\sigma_b - \frac{1}{3}\sigma_D\sigma_Y - \frac{7}{12}\sigma_D\sigma_a + \frac{1}{3}\sigma_D\sigma_b + \sigma_Y\sigma_a - \frac{2}{3}\sigma_Y\sigma_b - \frac{7}{12}\sigma_a\sigma_b$
OAI4 (4-bit OR-AND-inverter)	$Y = \neg((A \vee B) \wedge (C \vee D))$	$\mathcal{H}_{\text{OAI4}}(\sigma_Y,\sigma_A,\sigma_B,\sigma_C,\sigma_a,\sigma_b) = \frac{2}{3}\sigma_A - \frac{1}{3}\sigma_B - \frac{1}{3}\sigma_C - \frac{1}{3}\sigma_D - \frac{1}{3}\sigma_Y - \sigma_a - \sigma_b - \frac{1}{3}\sigma_A\sigma_B + \frac{1}{3}\sigma_A\sigma_Y - \frac{1}{3}\sigma_A\sigma_a - \sigma_A\sigma_b + \frac{2}{3}\sigma_B\sigma_b + \frac{1}{3}\sigma_C\sigma_D + \frac{2}{3}\sigma_C\sigma_Y + \frac{2}{3}\sigma_C\sigma_a + \frac{2}{3}\sigma_D\sigma_Y + \frac{2}{3}\sigma_D\sigma_a + \sigma_Y\sigma_a - \frac{1}{3}\sigma_Y\sigma_b + \frac{1}{3}\sigma_a\sigma_b$
DFF_p (positive edge-triggered D flip-flop)	$Q = D$	$\mathcal{H}_{\text{DFF}_\text{p}}(\sigma_Q,\sigma_D) = -\sigma_Q\sigma_D$ *(See Section 4.3.3.)*
DFF_n (negative edge-triggered D flip-flop)	$Q = D$	$\mathcal{H}_{\text{DFF}_\text{n}}(\sigma_Q,\sigma_D) = -\sigma_Q\sigma_D$ *(See Section 4.3.3.)*

4.3.6 Passing arguments

Recall that all of the quadratic pseudo-Boolean functions in our standard-cell library (Table 5) are objective functions for the quantum annealer to minimize. Hence, $\mathcal{H}_{\wedge}$, for example, does not return Y as the AND of A and B but rather specifies a *relation* among Y, A, and B that is maintained exactly where $\mathcal{H}_{\wedge}$ is minimized. As one normally wants to compute a function of specific, rather than random, inputs, we need a way to specify function inputs.

Sections 4.3.4 and 4.3.5 provide all of the information needed for this task. Supposing we want to compute AND(TRUE, FALSE, TRUE), we can use $\mathcal{H}_{\text{VCC}}$ and $\mathcal{H}_{\text{GND}}$ to "pin" values to TRUE and FALSE, respectively:

$$\begin{array}{lcl} \mathcal{H}_{\text{AND}_3}(\sigma_Y,\sigma_A,\sigma_B,\sigma_C) & \rightarrow & Y = A \wedge B \wedge C \\ \mathcal{H}_{\text{VCC}}(\sigma_A) & \rightarrow & A = \text{TRUE} \\ \mathcal{H}_{\text{GND}}(\sigma_B) & \rightarrow & B = \text{FALSE} \\ + \ \mathcal{H}_{\text{VCC}}(\sigma_C) & \rightarrow & C = \text{TRUE} \\ \hline \mathcal{H}_{\text{AND}_3}(\sigma_Y,+1,-1,+1) & \rightarrow & Y = \text{TRUE} \wedge \text{FALSE} \wedge \text{TRUE} \end{array}$$

Because σ_Y is left unbound, the quantum annealer will assign it whatever value minimizes the function, viz. −1. Assigning values to variables is such a common operation, `qmasm` even

provides a `--pin` command-line option both to simplify the task and to separate program code from program inputs.

We just as easily could have bound σ_Y and left σ_A, σ_B, and σ_C unbound:

$$\begin{array}{ll} \mathcal{H}_{\text{AND}_3}(\sigma_Y,\sigma_A,\sigma_B,\sigma_C) & \rightarrow \quad Y = A \wedge B \wedge C \\ \mathcal{H}_{\text{VCC}}(\sigma_Y) & \rightarrow \quad Y = \text{TRUE} \\ \hline \mathcal{H}_{\text{AND}_3}(+1,\sigma_A,\sigma_B,\sigma_C) & \rightarrow \quad \text{TRUE} = A \wedge B \wedge C \end{array}$$

In this case, the quantum annealer will assign $\sigma_A = \sigma_B = \sigma_C = +1$, as doing so minimizes the function. This ability to provide outputs and solve for inputs is central to the importance of our work and will be discussed in detail throughout Section 5.

4.4 Lowering QMASM to a physical Hamiltonian function

As discussed in Section 2, there are a few discrepancies between Equation (2) and what is actually implemented by a current D-Wave quantum annealer. In particular, only a small subset of all possible $J_{i,j}$ values are allowed to be non-zero due to the sparsity of the Chimera-graph hardware topology. A Chimera graph contains no odd-length cycles, which implies that of all the quadratic pseudo-Boolean functions listed in Table 5, only $\mathcal{H}_{\neg}$, $\mathcal{H}_{\text{DFF}_\text{p}}$, and $\mathcal{H}_{\text{DFF}_\text{n}}$—the three simplest functions in the table—can be implemented directly. As an example, the terms $\frac{1}{3}\sigma_A\sigma_B$, $\frac{1}{3}\sigma_A\sigma_Y$, and $-\frac{1}{3}\sigma_B\sigma_Y$ from the definition of $\mathcal{H}_{\text{AOI3}}$ form the 3-cycle $\sigma_A - \sigma_B - \sigma_Y$, but there is no 3-cycle in a graph like Figure 1 where those variables can all be adjacent to each other.

The solution is to perform a *minor embedding* [14] of the problem graph onto the hardware graph. This is analogous to statically routing packets in a network on chip. Minor embedding works by replacing certain individual variables with two or more variables that are made equal to each other using negative-valued $J_{i,j}$ coefficients. Doing so generally requires adjusting other coefficients. For example, the logical quadratic pseudo-Boolean function

$$\mathcal{H}_{\text{log}}(\sigma_A,\sigma_B,\sigma_C) = \tfrac{1}{2}\sigma_A + \tfrac{1}{2}\sigma_B + \tfrac{1}{2}\sigma_C + \sigma_A\sigma_B + \sigma_B\sigma_C + \sigma_C\sigma_A$$

can be converted to the physical quadratic pseudo-Boolean function

$$\begin{aligned} \mathcal{H}_{\text{phys}}(\sigma_0,\sigma_2,\sigma_4,\sigma_5) &= \tfrac{1}{4}\sigma_0 + \tfrac{1}{8}\sigma_2 + \tfrac{1}{8}\sigma_4 + \tfrac{1}{4}\sigma_5 \\ &\quad + \tfrac{1}{2}\sigma_0\sigma_4 + \tfrac{1}{2}\sigma_0\sigma_5 - \sigma_2\sigma_4 + \tfrac{1}{2}\sigma_2\sigma_5 \end{aligned}$$

by mapping variable σ_A to physical qubit σ_0, σ_C to physical qubit σ_5, and σ_B to *both* physical qubits σ_2 and σ_4. σ_2 and σ_4 are equated by introducing a $-\sigma_2\sigma_4$ term. As can be seen from Figure 1, all of the quadratic terms in $\mathcal{H}_{\text{phys}}$ are backed by edges in the D-Wave's Chimera graph.

As the final step of the compilation process, QMASM code is lowered to this physical form using the minor-embedding algorithm provided by D-Wave's SAPI library [13, 25]. `qmasm` scales coefficients to honor the hardware-supported ranges and optionally performs a few optimization steps:

- Explicit `A = B` constraints in the code result in merging σ_A and σ_B into a single variable.
- `qmasm` uses SAPI's implementation of roof duality [33] to elide qubits whose final value can be determined a priori.

Once the hardware-specific Hamiltonian function is produced, `qmasm` runs it on a D-Wave system and reports $\text{argmin}_{\bar{\sigma}}\,\mathcal{H}(\bar{\sigma})$ back to the user, but in terms of logical variable names rather than physical qubit numbers.

5 Examples

In Section 4 we demonstrated that it is indeed possible to compile classical code, written in Verilog, to a hardware-specific $\mathcal{H}(\bar{\sigma})$ for execution on a quantum annealer. In this section we motivate the productivity benefits of a Verilog-to-D-Wave compiler framework and examine some uses of this technology.

5.1 Conceptual model

With at most 2048 qubits for code plus data, it is clearly infeasible to compile large Verilog programs to a current-generation quantum annealer, specifically a D-Wave 2000Q. Given that the minimum annealing time on a D-Wave 2000Q is 1µs, corresponding to thousands of cycles of execution on a modern CPU, it is unlikely that running a Verilog program on a quantum annealer would see shorter execution times than when running the same program on a CPU.

The real benefit of our work lies in the ability to run programs not only from inputs to outputs but also from outputs to inputs, as noted at the end of Section 4.3.6. Consider problems in the complexity class NP [39]. Loosely speaking, this is a class of "difficult" computational problems in that they cannot be solved efficiently—meaning, in polynomial time—on a deterministic Turing machine. In practice, one commonly sidesteps the superpolynomial cost of solving an NP problem by settling for a polynomial-time approximate solution: fast, but not guaranteed to be correct in all cases.

In addition to the lack of guaranteed correctness, the polynomial-time-approximation approach exhibits another, more qualitative, shortcoming: it can be difficult to apply in practice. Typically, one *reduces* (maps) an NP problem of interest to a problem of equal or greater complexity for which an approximate solver already exists, approximately solves the reduced problem, then maps the result back to an approximate solution to the original problem. For example, one might reduce an arbitrary NP problem to the NP-complete problem of determining if a Boolean expression in conjunctive normal form is satisfiable [19] then using a satisfiability solver [30] to propose a possible solution. (NP-complete problems are at least as difficult as NP problems.) However, it is not always straightforward to construct such a reduction, even though a reduction is known to exist.

By the definition of NP, a proposed solution to an NP problem can be verified in polynomial time. This suggests that our compilation framework can be used as follows. Rather than write a program that directly solves an NP problem, one can write a program that *verifies* a proposed solution then run the program *backward* on the quantum annealer,

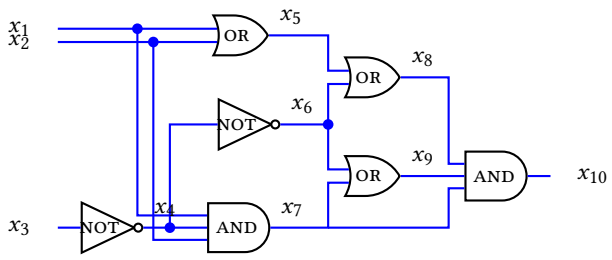


Figure 4. Sample logic circuit to satisfy

Listing 5. Verilog verification code corresponding to Figure 4

```
module circsat (a, b, c, y);
    input a, b, c;
    output y;
    wire [1:10] x;

    assign x[1] = a;
    assign x[2] = b;
    assign x[3] = c;
    assign x[4] = ~x[3];
    assign x[5] = x[1] | x[2];
    assign x[6] = ~x[4];
    assign x[7] = x[1] & x[2] & x[4];
    assign x[8] = x[5] | x[6];
    assign x[9] = x[6] | x[7];
    assign x[10] = x[8] & x[9] & x[7];
    assign y = x[10];
endmodule
```

from the output "yes, the solution is valid" to the inputs that correspond to a valid solution. The result is a (generally) simple-to-write program that rapidly finds approximate solutions to NP problems.

5.2 Circuit satisfiability

Given how fundamental circuit satisfiability is to complexity theory [39] and given that Verilog, as a hardware-description language, is ideally suited to describing circuits, circuit satisfiability is a natural first problem to showcase our approach. Circuit satisfiability can be expressed as the decision problem, *Given a combinational-logic circuit, does there exist a set of inputs that cause the circuit to output* True*?*, but in practice, we often want to know in the True case what those inputs are.

We arbitrarily select a circuit appearing in a popular algorithms textbook [20], reproduced in Figure 4, to demonstrate circuit satisfiability. Our task is to find values of x_1, x_2, and x_3 that lead to x_{10} being True, if any such values exist.

Listing 5 presents Verilog code that performs the opposite task: given values of x_1, x_2, and x_3, return the value of x_{10}. (For readers not familiar with Verilog, one can think of a `wire` as an internal variable. Listing 5 defines ten such internal variables, `x[1]` through `x[10]`.) When we compile this code and run it on a quantum annealer with y (= `x[10]`) pinned to True, the hardware returns a (= `x[1]`) and b (= `x[2]`) True and c (= `x[3]`) False, which is a correct solution.

If the circuit were not satisfiable, the quantum annealer would return an invalid solution, as Equation (1) has no ability to represent "no solution". However, because a proposed solution can, by the definition of NP, be verified in polynomial time, we can easily check a result by running the code in Listing 5 forward from inputs to outputs, on either a quantum annealer or classical hardware, and discard any results found to be incorrect.

The reader is invited to compare Listing 5 with published QMASM code that solves the same problem [49]. It is readily apparent that the Verilog version is shorter and clearer due to the higher level of abstraction it provides over a raw, quadratic pseudo-Boolean function, even one that takes advantage of QMASM macros and other features.

5.3 Factoring integers

The best known algorithms for factoring a semiprime into its two prime factors, including the quadratic sieve and the number field sieve, run in exponential time [51]. On a gate-model quantum computer, Shor's algorithm [55] delivers superpolynomial speedup over these. However, all of these algorithms rely on sophisticated number theory and would therefore be nearly impossible for a casual programmer to devise independently.

Listing 6. Verilog code to multiply two integers

```
module mult (A, B, C);
    input [3:0] A;
    input [3:0] B;
    output[7:0] C;

    assign C = A * B;
endmodule
```

In contrast, the ability to run code backward makes factoring trivial to program with our approach. One need only express a simple $C = A \times B$ multiplication (Listing 6), provide a value for C, and let the quantum annealer solve for A and B. For example, executing the generated QMASM code with `--pin="C[7:0] := 10001111"` (143 decimal) returns two unique solutions: $\{A = 11, B = 13\}$ and $\{A = 13, B = 11\}$. The same code can be used to multiply two numbers, e.g., with `--pin="A[3:0] := 1101" --pin="B[3:0] := 1011"`, or even divide, e.g., with `--pin="C[7:0] := 10001111" --pin="A[3:0] := 1101"`.

5.4 Map coloring

For our final example we consider the problem of map coloring: *Given a planar map, can it be colored with no more than four colors such that no two adjacent regions share a color?* As with circuit satisfiability, we normally want to produce a valid coloring in the "yes" case.

We choose the map of Australia, shown in Figure 5, as an example of a map to four-color. Our solution, like the previous two solutions, is to write code that verifies a proposed solution then run it backward from "the coloring is valid" to the actual coloring.

Listing 7 presents the Verilog code for such a verifier. For each of the regions on the map (excluding Tasmania, which

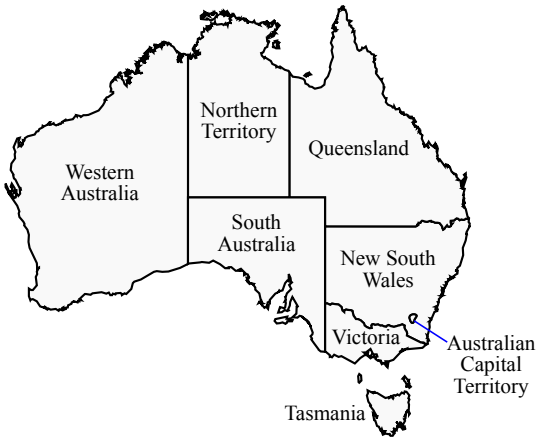


Figure 5. Map of Australia (states and territories)

Listing 7. Verilog verification code for a 4-coloring of Figure 5

```
1 module australia (NSW, QLD, SA, VIC, WA, NT, ACT, valid);
2     input [1:0] NSW, QLD, SA, VIC, WA, NT, ACT;
3     output valid;
4
5     assign valid = WA != NT && WA != SA && NT != SA && NT !=
          QLD && SA != QLD && SA != NSW && SA != VIC && QLD
          != NSW && NSW != VIC && NSW != ACT;
6 endmodule
```

is an island and therefore independent of the coloring of the mainland regions), the code inputs a 2-bit number representing the region's color. It outputs a single `valid` bit that is set to True if and only if all of the adjacency constraints hold: Western Australia has a different color from the Northern Territory; Western Australia has a different color from South Australia; the Northern Territory has a different color from South Australia; and so forth.

We run the compiled code with `--pin="valid := true"`, and it returns a valid coloring, such as $\{\text{ACT} = 2, \text{NSW} = 0, \text{NT} = 1, \text{QLD} = 3, \text{SA} = 2, \text{VIC} = 3, \text{WA} = 3\}$ or $\{\text{ACT} = 0, \text{NSW} = 2, \text{NT} = 2, \text{QLD} = 0, \text{SA} = 1, \text{VIC} = 0, \text{WA} = 3\}$. In practice, it is common to perform a large number of anneals (say, thousands) per run, both to amortize startup overhead and to increase the likelihood of encountering a correct solution. Remember, all quantum computers are fundamentally stochastic devices.

6 Analysis

Focusing solely on the map-coloring example from Section 5.4, we examine some properties of the compilation process and subsequent code execution.

6.1 Static properties

Listing 7 compiles from 6 lines of Verilog to 123 lines of EDIF to 736 lines of QMASM (excluding the 232 lines in the standard-cell library). The resulting logical quadratic pseudo-Boolean function comprises 74 variables. Because we use a randomized, heuristic minor embedder [13], the number of physical qubits varies from compilation to compilation. Over 25 compilations this number averaged 369 ± 26. The rather substantial increase from variable count to qubit count is directly attributable to the sparseness of the hardware topology. The total size of the quadratic pseudo-Boolean function increased from 312 terms in $\mathcal{H}_{\text{log}}$ to 963 ± 53 terms in $\mathcal{H}_{\text{phys}}$.

To put those logical and physical variable counts in perspective we consider the tallies that one might see when hand-coding a quadratic pseudo-Boolean function corresponding to the map-coloring problem. Following the approach laid out by Dahl [24], Lucas [42], and Rieffel et al. [53], we employ a unary rather than a binary encoding of each region's color. That is, we use one variable for each of a region's four colors. This requires a logical problem size of 4 variables per region $\times$ 7 regions = 28 variables. Representing the required all-to-all connectivity within a single unit cell of a Chimera graph requires 8 qubits, 2 per color. Mapping Australia's states and territories to the Chimera graph's 2-D grid requires that some regions be replicated, for example to enable South Australia to border five other states. A pencil-and-paper analysis indicates that four replicas should suffice for this problem. This leads to a total of $8 \times (7+4) = 88$ qubits.

In short, the automated approach presented in this paper enables a straightforward 6-line textual expression of the map-coloring problem to replace some amount of cleverness in problem expression (viz., realizing that a unary encoding simplifies the implementation of map-coloring constraints) and the tedium of mapping variables onto the underlying hardware topology using what can practically be considered a manual place-and-route effort. However, the convenience of expressing a quadratic pseudo-Boolean function in Verilog comes at a cost. When solving the problem of four-coloring the map of Australia, the Verilog version consumes 2.6x the number of logical variables as the hand-coded version (74 vs. 28). After targeting the D-Wave's sparse topology, the difference in physical qubits is even greater: a 4x increase in qubit count from the hand-coded solution to the mechanically generated one (from 88 to 369 ± 26).

6.2 Execution time

It is extremely difficult in the general case to determine the asymptotic complexity of a quantum-annealing algorithm, although analytical results do exist for a few specific algorithms [27, 52]. We therefore employ an empirical performance comparison with a classical solver.

After compiling Listing 7 to QMASM, we measured the time to perform 1,000,000 anneals of 20µs apiece. The time includes HTTPS communication with a remote D-Wave 2000Q, queuing delays, and numerous overheads on both sides of the network but not the time for minor embedding. We compared this to 100,000 runs of the MiniZinc [46] code shown in Listing 8, using Chuffed [15] as the solver. We exclude the time to lower MiniZinc to FlatZinc.

Listing 8. MiniZinc code to four-color a map of Australia

```
var 1..4: NSW;
var 1..4: QLD;
var 1..4: SA;
var 1..4: VIC;
var 1..4: WA;
var 1..4: NT;
var 1..4: ACT;

constraint WA != NT;
constraint WA != SA;
constraint NT != SA;
constraint NT != QLD;
constraint SA != QLD;
constraint SA != NSW;
constraint SA != VIC;
constraint QLD != NSW;
constraint NSW != VIC;
constraint NSW != ACT;

solve satisfy;
```

Both executions were performed on a 3.0 GHz Xeon E5-2687W v4. The D-Wave execution averaged 734µs per solution while Chuffed averaged 1798µs per solution. This is a far from scientific study, as the Chuffed version guarantees correctness and optimality of its output while the D-Wave version does not. However, the Chuffed version returns the same solution every time while the D-Wave version samples from the space of solutions. The point is that the performance of our approach is not necessarily worse than that of a classical solver, even given the early generation quantum hardware on which we ran.

7 Conclusions

Quantum annealing is a substantially different computational paradigm from what most programmers—and compilers—are accustomed to target. Rather than providing the hardware with sequences of operations that read and mutate state, one instead provides the hardware with the coefficients to a quadratic pseudo-Boolean function, and the hardware suggests variable assignments that are likely to minimize that function but with no guarantee that these assignments indeed lead to the global minimum.

Current programming models for quantum annealers, specifically those produced by D-Wave Systems, present only a thin level of abstraction above the hardware interface. We claim that to increase programmer productivity there need to exist higher-level programming abstractions. Rather than express code in terms of a list of real numbers and receive results as a list of Booleans, productivity could benefit from access to named, multi-bit variables; arithmetic and relational operations; conditionals; loops; and other programming conveniences to which classically trained programmers are accustomed. The challenge lies in mapping such software constructs to the somewhat limited form of quadratic pseudo-Boolean function the hardware accepts.

The first conclusion we draw from the work presented in this paper is that it is indeed possible to bridge the enormous semantic gap between classical programming languages and the native interface to a quantum annealer. The evidence that supports this conclusion is provided in Section 4, where we describe a compilation framework that successively lowers classical code through a series of transformations until it takes on the form of a quadratic pseudo-Boolean function. Specifically, we have shown that one can begin with classical code expressed in a hardware-description language (Section 4.1); use a hardware-synthesis tool to compile that to a netlist representation of a digital circuit (Section 4.2); translate the cells and nets appearing in that netlist to an accumulation of quadratic pseudo-Boolean function building blocks defined in a standard-cell library (Section 4.3); and finally map the resulting logical function to a physical variant that considers the underlying hardware's specific limitations (Section 4.4). As the result of our efforts, programmers can write programs using an existing, albeit somewhat domain-specific, classical programming language and compile these programs for execution on a quantum annealer.

Our second conclusion is that classically programming a quantum annealer facilitates the approximate solution of problems that lie in the NP complexity class (computationally difficult problems) but outside of P (computationally easy problems). Exactly solving such problems tends to be easy to express with brute-force techniques but requires superpolynomial time to execute. Finding an approximate solution is achievable in polynomial time but doing so tends to be an intellectually challenging endeavor as it requires figuring out how to reduce the problem to another problem for which an approximate solver exists. As we demonstrate in Section 5, our approach provides the best of both worlds: simple coding and at least the possibility of a polynomial-time approximate solution.

We are still in the very early days of the availability of quantum hardware. As future generations of quantum annealers introduce increased qubit counts, greater connectivity, improved coefficient range/precision, longer maximum annealing times, reduced noise, and other engineering improvements, such systems may become a common form of computational accelerator. If that happens, the compilation techniques presented in this paper will make it possible for even relatively novice programmers to take advantage of the computational power of quantum annealers.

Acknowledgments

Research presented in this article was supported by the Laboratory Directed Research and Development program of Los Alamos National Laboratory under project numbers 20160069DR and 20190065DR. This work was also supported by the U.S. Department of Energy through Los Alamos National Laboratory. Los Alamos National Laboratory is operated by Triad National Security, LLC for the National Nuclear Security Administration of the U.S. Department of Energy (contract no. 89233218CNA000001).

The edif2qmasm and QMASM tools (see Appendix A) were developed using Ising, Los Alamos National Laboratory's D-Wave quantum annealer. Ising is supported by the National Nuclear Security Administration's Advanced Simulation and Computing program.

References

[1] Dorit Aharonov, Wilm van Dam, Julia Kempe, Zeph Landau, Seth Lloyd, and Oded Regev. 2008. Adiabatic quantum computation is equivalent to standard quantum computation. *SIAM Review*, 50, 4, 755–787. ISSN: 1095-7200. DOI: 10.1137/080734479.

[2] Francisco Barahona. 1982. On the computational complexity of Ising spin glass models. *Journal of Physics A: Mathematical and General*, 15, 10, (October 1982), 3241–3253. ISSN: 1361-6447. DOI: 10.1088/0305-4470/15/10/028.

[3] Berkeley Logic Synthesis and Verification Group. 2016. ABC: a system for sequential synthesis and verification. (July 17, 2016). Retrieved 07/31/2018 from http://www.eecs.berkeley.edu/~alanmi/abc/.

[4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: a fresh approach to numerical computing. *SIAM Review*, 59, 1, (March 2017), 65–98. Desmond Higham, editor. ISSN: 1095-7200. DOI: 10.1137/141000671.

[5] Jacob D. Biamonte and Peter J. Love. 2008. Realizable Hamiltonians for universal adiabatic quantum computers. *Physical Review A*, 78, (July 28, 2008), 012352-1–7, 1, (July 28, 2008). DOI: 10.1103/PhysRevA.78.012352.

[6] Zhengbing Bian, Fabian Chudak, Robert Israel, Brad Lackey, William G. Macready, and Aidan Roy. 2014. Discrete optimization using quantum annealing on sparse Ising models. *Frontiers in Physics*, 2, (September 18, 2014). ISSN: 2296-424X. DOI: 10.3389/fphy.2014.00056.

[7] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS (Amsterdam, The Netherlands, March 20–28, 1999). Cleaveland and W. Rance, editors. Springer, Berlin, Germany and Heidelberg, Germany, 193–207. ISBN: 978-3-540-49059-3. DOI: 10.1007/3-540-49059-0_14.

[8] Michael Booth, Edward Dahl, Mark Furtney, and Steven P. Reinhardt. 2016. Abstractions considered helpful: a tools architecture for quantum annealers. In *Proceedings of the 2016 IEEE High Performance Extreme Computing Conference*. HPEC (Waltham, Massachusetts, USA, September 13–15, 2016). IEEE, (December 1, 2016). ISBN: 978-1-5090-3525-0. DOI: 10.1109/HPEC.2016.7761625.

[9] M[ax] Born and V[ladimir] Fock. 1928. Beweis des adiabatensatzes. *Zeitschrift für Physik*, 51, 3–4, (March 1, 1928), 165–180. ISSN: 0044-3328. DOI: 10.1007/BF01343193.

[10] Sergey Bravyi, David P. DiVincenzo, Roberto I. Oliveira, and Barbara M. Terhal. 2007. The complexity of stoquastic local Hamiltonian problems. (October 2, 2007). arXiv: 0606140v4 [quant-ph].

[11] Robert Brayton and Alan Mishchenko. 2010. ABC: an academic industrial-strength verification tool. In *Proceedings of the 22nd International Conference on Computer Aided Verification*. CAV (Edinburgh, United Kingdom, July 15–19, 2010). Tayssir Touili, Byron Cook, and Paul Jackson, editors. Springer, Berlin, Germany and Heidelberg, Germany, 24–40. ISBN: 978-3-642-14295-6. DOI: 10.1007/978-3-642-14295-6_5.

[12] Paul I. Bunyk, Emile M. Hoskinson, Mark W. Johnson, Elena Tolkacheva, Fabio Altomare, Andrew J. Berkley, Richard Harris, Jeremy P. Hilton, Trevor Lanting, Anthony J. Przybysz, and Jed Whittaker. 2014. Architectural considerations in the design of a superconducting quantum annealing processor. *IEEE Transactions on Applied Superconductivity*, 24, 4, (August 2014), 1–10. ISSN: 1051-8223. DOI: 10.1109/TASC.2014.2318294.

[13] Jun Cai, Bill Macready, and Aidan Roy. 2014. A practical heuristic for finding graph minors. D-Wave Systems Inc. (June 10, 2014). arXiv: 1406.2741v1 [quant-ph].

[14] Vicky Choi. 2008. Minor-embedding in adiabatic quantum computation. I. the parameter setting problem. *Quantum Information Processing*, 7, 5, (October 2008), 193–209. ISSN: 1573-1332. DOI: 10.1007/s11128-008-0082-9.

[15] Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. [n. d.] Chuffed, a lazy clause generation solver. Retrieved 12/19/2018 from https://github.com/chuffed/chuffed.

[16] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems* (Lecture Notes in Computer Science). Kurt Jensen and Andreas Podelski, editors. Volume 2988. Springer, Berlin, Germany and Heidelberg, Germany, 168–176. ISBN: 978-3-540-24730-2. DOI: 10.1007/978-3-540-24730-2_15.

[17] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. 2004. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25, 2–3, (September 2004), 105–127. ISSN: 1572-8102. DOI: 10.1023/B:FORM.0000040025.89719.f3.

[18] Emily Conover. 2018. Google moves toward quantum supremacy with 72-qubit computer. *Science News*, 193, 6, (March 5, 2018), 13. Retrieved 07/19/2018 from https://www.sciencenews.org/article/google-moves-toward-quantum-supremacy-72-qubit-computer.

[19] Stephen A. Cook. 1971. The complexity of theorem-proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing. Papers Presented at the Symposium*. STOC '71 (Shaker Heights, Ohio, USA, May 3–5, 1971). Michael A. Harrison, Ranan B. Banerji, Jeffrey D. Ullman, and Philip M. Lewis, editors. ACM Special Interest Group for Automata and Computability Theory. ACM, New York, New York, USA, 151–158. DOI: 10.1145/800157.805047.

[20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. (3rd edition). MIT Press, (July 31, 2009). 1312 pages. ISBN: 978-0-262-03384-8.

[21] D-Wave Systems, Inc. 2017. D-Wave initiates open quantum software environment. (January 11, 2017). Retrieved 08/02/2018 from https://www.dwavesys.com/press-releases/d-wave-initiates-open-quantum-software-environment.

[22] D-Wave Systems, Inc. [n. d.] D-Wave Ocean software documentation. Retrieved 12/19/2018 from https://docs.ocean.dwavesys.com/en/latest/index.html.

[23] D-Wave Systems, Inc. 2016. *qbsolv—Minimize the Objective Function Represented by a QUBO*. qbsolv(1) manual page. Burnaby, British Columbia, Canada. 3 pages. Retrieved 07/23/2018 from https://github.com/dwavesystems/qbsolv/blob/master/doc/qbsolv.pdf.

[24] E. D. Dahl. 2013. Programming with D-Wave: Map Coloring Problem. White Paper. D-Wave Systems, Burnaby, British Columbia, Canada, (November 2013). https://www.dwavesys.com/sites/default/files/Map%20Coloring%20WP2.pdf.

[25] 2018. Developer Guide for Python. User Manual 09-1024A-K. D-Wave Systems, Inc., Burnaby, British Columbia, Canada, (July 30, 2018). 69 pages.

[26] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A Quantum Approximate Optimization Algorithm. Technical report MIT-CTP/4610. Center for Theoretical Physics, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, (November 14, 2014). 16 pages. arXiv: 1411.4028v1 [quant-ph].

[27] Edward Farhi, Jeffrey Goldstone, Sam Gutmann, and Michael Sipser. 2000. Quantum computation by adiabatic evolution. (January 28, 2000). arXiv: 0001106v1 [quant-ph].

[28] A. B. Finnila, M. A. Gomez, C. Sebenik, C. Stenson, and J. D. Doll. 1994. Quantum annealing: a new method for minimizing multidimensional functions. *Chemical Physics Letters*, 219, 5, (March 18, 1994), 343–348. ISSN: 0009-2614. DOI: 10.1016/0009-2614(94)00117-0.

[29] Elizabeth Gibney. 2017. D-Wave upgrade: how scientists are using the world's most controversial quantum computer. *Nature*, 541, 7638, (January 24, 2017), 447–448. DOI: 10.1038/541447b.

[30] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. 2008. Satisfiability solvers. In *Handbook of Knowledge Representation*. Foundations of Artificial Intelligence. Volume 3. Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors. Elsevier. Chapter 2, 89–134. DOI: 10.1016/S1574-6526(07)03002-7.

[31] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: a scalable quantum

programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI (Seattle, Washington, USA, June 16–19, 2013). Hans Boehm and Cormac Flanagan, editors. Association of Computing Machinery, New York, New York, USA, 333–342. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462177.

[32] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference*. SciPy 2008 (Pasadena, California, USA, August 19–24, 2008). Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, 11–15. Retrieved 12/19/2018 from http://conference.scipy.org/proceedings/SciPy2008/paper_2/.

[33] P. L. Hammer, P. Hansen, and B. Simeone. 1984. Roof duality, complementation and persistency in quadratic 0–1 optimization. *Mathematical Programming*, 28, 2, (February 1984), 121–155. ISSN: 1436-4646. DOI: 10.1007/BF02612354.

[34] R. Harris, T. Lanting, A. J. Berkley, J. Johansson, M. W. Johnson, P. Bunyk, E. Ladizinsky, N. Ladizinsky, T. Oh, and S. Han. 2009. Compound Josephson-junction coupler for flux qubits with minimal crosstalk. *Physical Review B*, 80, (August 20, 2009), 052506-1–4, 5, (August 20, 2009). ISSN: 2469-9969. DOI: 10.1103/PhysRevB.80.052506.

[35] IEEE. 2006. IEEE Standard for Verilog Hardware Description Language. Standard IEEE Std 1364-2005. Design Automation Standard Committee of the IEEE Computer Society, New York, New York, USA, (April 7, 2006). 590 pages. DOI: 10.1109/IEEESTD.2006.99495.

[36] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. 2015. ScaffCC: scalable compilation and analysis of quantum programs. *Parallel Computing*, 45, (June 2015), 2–17. ISSN: 0167-8191. DOI: 10.1016/j.parco.2014.12.001.

[37] Tadashi Kadowaki and Hidetoshi Nishimori. 1998. Quantum annealing in the transverse Ising model. *Physical Review E*, 58, (November 1998), 5355–5363, 5, (November 1998). ISSN: 1063-651X. DOI: 10.1103/PhysRevE.58.5355.

[38] Hilary Kahn, Robin La Fontaine, and Rachel Lau. 2000. Electronic Design Interchange Format (EDIF). Part 2: Version 4 0 0. International Standard IEC 61690-2:2000. International Electrotechnical Commission, Manchester, United Kingdom, (January 31, 2000).

[39] Richard M. Karp. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations*. Symposium on the Complexity of Computer Computations (Yorktown Heights, New York, USA, March 20–22, 1972). Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger, editors. Plenum, New York, New York, USA, 85–103. ISBN: 978-1-4684-2001-2. DOI: 10.1007/978-1-4684-2001-2_9.

[40] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. 1983. Optimization by simulated annealing. *Science*, 220, 4598, (May 13, 1983), 671–680. ISSN: 0036-8075. DOI: 10.1126/science.220.4598.671.

[41] Gary Kochenberger, Jin-Kao Hao, Fred Glover, Mark Lewis, Zhipeng Lü, Haibo Wang, and Yang Wang. 2014. The unconstrained binary quadratic programming problem: a survey. *Journal of Combinatorial Optimization*, 28, 1, (July 2014), 58–81. ISSN: 1573-2886. DOI: 10.1007/s10878-014-9734-0.

[42] Andrew Lucas. 2014. Ising formulations of many NP problems. *Frontiers in Physics*, 2, (February 14, 2014), 5. ISSN: 2296-424X. DOI: 10.3389/fphy.2014.00005.

[43] Satoshi Matsubara, Hirotaka Tamura, Motomu Takatsu, Danny Yoo, Behraz Vatankhahghadim, Hironobu Yamasaki, Toshiyuki Miyazawa, Sanroku Tsukamoto, Yasuhiro Watanabe, Kazuya Takemoto, and Ali Sheikholeslami. 2018. Ising-model optimizer with parallel-trial bit-sieve engine. In *Proceedings of the 11th International Conference on Complex, Intelligent, and Software Intensive Systems* (Advances in Intelligent Systems and Computing). CISIS (Torino, Italy, July 10–12, 2017). Leonard Barolli and Olivier Terzo, editors. Volume 611. Springer International, Cham, Germany, 432–438. ISBN: 978-3-319-61566-0. DOI: 10.1007/978-3-319-61566-0_39.

[44] Microsoft Corp. 2017. The Q# progamming language. (December 11, 2017). Retrieved 07/23/2018 from https://docs.microsoft.com/en-us/quantum/quantum-qr-intro.

[45] Ari Mizel, Daniel A. Lidar, and Morgan Mitchell. 2007. Simple proof of equivalence between adiabatic quantum computation and the circuit model. *Physical Review Letters*, 99, (August 16, 2007), 070502-1–4, 7, (August 16, 2007). DOI: 10.1103/PhysRevLett.99.070502.

[46] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. 2007. MiniZinc: towards a standard CP modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*. CP 2007 (Providence, Rhode Island, USA, September 25–29, 2007). Christian Bessière, editor. Springer, Berlin, Germany and Heidelberg, Germany, 529–543. ISBN: 978-3-540-74970-7. DOI: 10.1007/978-3-540-74970-7_38.

[47] Daniel O'Malley and Velimir V. Vesselinov. 2016. ToQ.jl: a high-level programming language for D-Wave machines based on Julia. In *Proceedings of the 2016 IEEE High Performance Extreme Computing Conference*. HPEC (Waltham, Massachusetts, USA, September 13–15, 2016). IEEE, (December 1, 2016). ISBN: 978-1-5090-3525-0. DOI: 10.1109/HPEC.2016.7761616.

[48] Takuya Okuyama, Masato Hayashi, and Masanao Yamaoka. 2017. An Ising computer based on simulated quantum annealing by path integral Monte Carlo method. In *2017 IEEE International Conference on Rebooting Computing*. ICRC (Washington, DC, USA, November 8–9, 2017). IEEE, (December 1, 2017). ISBN: 978-1-5386-1553-9. DOI: 10.1109/ICRC.2017.8123652.

[49] Scott Pakin. 2016. A quantum macro assembler. In *Proceedings of the 2016 IEEE High Performance Extreme Computing Conference*. HPEC (Waltham, Massachusetts, USA, September 13–15, 2016). IEEE, (December 1, 2016). ISBN: 978-1-5090-3525-0. DOI: 10.1109/HPEC.2016.7761637.

[50] Scott Pakin. 2018. Performing fully parallel constraint logic programming on a quantum annealer. *Theory and Practice of Logic Programming*. DOI: 10.1017/S1471068418000066.

[51] Carl Pomerance. 1996. A tale of two sieves. *Notices of the AMS*, 43, 12, 1473–1485. ISSN: 0002-9920. Retrieved 08/05/2018 from https://www.ams.org/journals/notices/199612/pomerance.pdf.

[52] Ben W. Reichardt. 2004. The quantum adiabatic optimization algorithm and local minima. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*. STOC'04 (Chicago, Illinois, USA, June 13–16, 2004). ACM, New York, New York, USA, 502–510. ISBN: 1-58113-852-0. DOI: 10.1145/1007352.1007428.

[53] Eleanor G. Rieffel, Davide Venturelli, Bryan O'Gorman, Minh B. Do, Elicia M. Prystay, and Vadim N. Smelyanskiy. 2015. A case study in programming a quantum annealer for hard operational planning problems. *Quantum Information Processing*, 14, 1, (January 2015), 1–36. ISSN: 1573-1332. DOI: 10.1007/s11128-014-0892-x.

[54] Ronald L. Rivest. 1994. S-Expressions. Internet Draft draft-rivest-sexp-00.txt. Internet Engineering Task Force, Network Working Group, Cambridge, Massachusetts, USA, (May 4, 1994). Retrieved 08/01/2018 from http://people.csail.mit.edu/rivest/Sexp.txt.

[55] Peter W. Shor. 1999. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41, 2, 303–332. DOI: 10.1137/S0036144598347011.

[56] Alexander Singh, Konstantinos Giannakis, and Theodore Andronikos. 2017. Qumin, a minimalist quantum programming language. (April 14, 2017). arXiv: 1704.04460v1 [cs.PL].

[57] Robert S. Smith, Michael J. Curtis, and William J. Zeng. 2017. A practical quantum instruction set architecture. Rigetti & Co., Inc. (February 17, 2017). arXiv: 1608.03355v2 [quant-ph].

[58] Damian S. Steiger, Thomas Häner, and Matthias Troyer. 2018. ProjectQ: an open source software framework for quantum computing. *Quantum*, 2, (January 31, 2018), 49. ISSN: 2521-327X. DOI: 10.22331/q-2018-01-31-49. arXiv: 1612.08091v2 [quant-ph].

[59] The Cirq Developers. [n. d.] Cirq: a Python library for NISQ circuits. Retrieved 12/19/2018 from https://cirq.readthedocs.io/.

[60] Sanroku Tsukamoto, Motomu Takatsu, Satoshi Matsubara, and Hirotaka Tamura. 2017. An accelerator architecture for combinatorial

optimization problems. *Fujitsu Scientific & Technical Journal*, 53, 5, (November 2017), 8–13. Retrieved 07/20/2018 from http://www.fujitsu.com/global/documents/about/resources/publications/fstj/archives/vol53-5/paper02.pdf.

[61] Dave Wecker and Krysta M. Svore. 2014. LIQ$Ui|\rangle$: a software design architecture and domain-specific language for quantum computing. Microsoft Research. (February 18, 2014). arXiv: 1402.4467v1 [quant-ph].

[62] J. D. Whitfield, M. Faccin, and J. D. Biamonte. 2012. Ground-state spin logic. *Europhysics Letters*, 99, 5, (September 11, 2012), 57004-p1–p7. ISSN: 1286-4854. DOI: 10.1209/0295-5075/99/57004.

[63] Clifford Wolf and Johann Glaser. 2013. Yosys—a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics*. Austrochip 2013 (Linz, Austria). (October 10, 2013). Retrieved 08/01/2018 from http://www.clifford.at/yosys/files/yosys-austrochip2013.pdf.

[64] Masanao Yamaoka, Chihiro Yoshimura, Masato Hayashi, Takuya Okuyama, Hidetaka Aoki, and Hiroyuki Mizuno. 2016. A 20k-spin Ising chip to solve combinatorial optimization problems with CMOS annealing. *IEEE Journal of Solid-State Circuits*, 51, 1, (January 2016), 303–309. ISSN: 0018-9200. DOI: 10.1109/JSSC.2015.2498601.

A Software Availability

All the tools used in this paper for compiling Verilog code to a quadratic pseudo-Boolean function are available as free, open-source software:

- Yosys [63] (Verilog → EDIF): http://www.clifford.at/yosys/
- ABC [3, 11] (logic optimization): https://people.eecs.berkeley.edu/~alanmi/abc/
- edif2qmasm (EDIF → QMASM): https://github.com/lanl/edif2qmasm
- QMASM [49] (QMASM → quadratic pseudo-Boolean function): https://github.com/lanl/qmasm

QMASM requires D-Wave's proprietary SAPI library [25] to minimize the resulting quadratic pseudo-Boolean function using D-Wave hardware. However, QMASM can also be directed to minimize the function classically using the open-source qbsolv solver:

- qbsolv [21] (minimize a quadratic pseudo-Boolean function): https://github.com/dwavesystems/qbsolv