Performing fully parallel constraint logic programming on a quantum annealer*

SCOTT PAKIN

*Computer, Computational, and Statistical Sciences Division, Los Alamos National Laboratory, MS B287,
Los Alamos, New Mexico, USA*
(*e-mail:* `pakin@lanl.gov`)

Abstract

A *quantum annealer* exploits quantum effects to solve a particular type of optimization problem. The advantage of this specialized hardware is that it effectively considers all possible solutions in parallel, thereby potentially outperforming classical computing systems. However, despite quantum annealers having recently become commercially available, there are relatively few high-level programming models that target these devices. In this article, we show how to compile a subset of Prolog enhanced with support for constraint logic programming into a two-local Ising-model Hamiltonian suitable for execution on a quantum annealer. In particular, we describe the series of transformations one can apply to convert constraint logic programs expressed in Prolog into an executable form that bears virtually no resemblance to a classical machine model yet that evaluates the specified constraints in a fully parallel manner. We evaluate our efforts on a 1,095-qubit D-Wave 2X quantum annealer and describe the approach's associated capabilities and shortcomings.

KEYWORDS: quantum annealing, quantum computing, constraint logic programming, Prolog, D-Wave.

1 Introduction

Quantum annealing (Farhi and Gutmann 1998; Kadowaki and Nishimori 1998; Finnila *et al.* 1994) is a presumably weaker (Bravyi and Hastings 2017)[1] but more easily scalable (Kaminsky *et al.* 2004) form of quantum computing than the more traditional gate model (Feynman 1986). To quantify what we mean by "scalable," the recently introduced D-Wave 2000Q quantum annealer provides 2,000 qubits, while the state-of-the-art gate-model quantum computers are just starting to reach mid-double-digit qubit counts (Knight 2017).

[1] By "presumably weaker," we mean that quantum annealing is associated with the StoqMA complexity class (Bravyi *et al.* 2006); gate-model quantum computing is associated with the QMA complexity class; and StoqMA $\subseteq$ QMA—with a supposition that StoqMA $\subset$ QMA.

Programming a quantum annealer is nearly identical to solving an optimization problem using (classical) simulated annealing (Kirkpatrick *et al.* 1983). That is, one constructs an energy landscape via a multivariate function such that the coordinates of the landscape's ground state (i.e., its lowest value) correspond to the solution being sought. Quantum-annealing hardware then automatically relaxes to a solution—or one of the multiple equally valid solutions—with some probability. The "quantum" in "quantum annealer" refers to the use of quantum effects, most notably *quantum tunneling*: The ability to "cut through" tall energy barriers to reach ground states with a higher probability than could be expected from a classical solution (Kadowaki and Nishimori 1998).

The advantage of quantum annealing over classical code execution is its abundant inherent parallelism. A quantum annealer effectively examines all possible inputs in parallel to find solutions to a problem. For NP-complete problems (Cormen *et al.* 2001), this implies a potential exponential speedup over a brute-force approach. The catch is the "effectively." Quantum annealers are fundamentally stochastic devices. They provide no guarantee of finding an optimal (lowest energy) solution. Consequently, they can be considered more as an automatic heuristic-finding mechanism than as a formal solver.

The question we ask in this work is, *Can one express constraint logic programming (CLP) in the form accepted by quantum-annealing hardware?* Although such hardware is in its first few generations and not yet able to compete performance-wise with traditional, massively parallel systems, our belief is that the potential exists to one day be able to solve CLP problems faster on quantum annealers than on conventional hardware. The primary challenge lies in how to express CLP—or, for that matter, almost any programming model—as an energy landscape whose ground state corresponds to a satisfaction of the given constraints. The primary contribution of this work is therefore the demonstration that such problem expressions are indeed possible and the presentation of a methodology (reified in software) to accomplish that task.

The rest of this article is structured as follows. Section 2 details how problems need to be formulated for execution on a quantum annealer. Although mapping CLP onto a quantum annealer is a novel endeavor, Section 3 discusses other programming models that target quantum annealers and gate-model quantum computers. Section 4 is the core part of the article. It describes our implementation of quantum-annealing Prolog, a CLP-enhanced Prolog subset and associated compiler for exploiting the massive effective parallelism of a quantum annealer when solving CLP problems. Some examples and experiments are presented in Section 5. Finally, we draw some conclusions from our work in Section 6.

2 Background

2.1 Quantum annealing

A quantum annealer is a special-purpose device that finds a vector, $\boldsymbol{\sigma}$, of *spins* (Booleans, represented as ± 1) that minimize the energy of an Ising-model

Hamiltonian (Johnson *et al.* 2011). Quantum annealers from D-Wave systems, Inc. further restrict the Hamiltonian to being two-local, meaning that it can contain quadratic terms but not cubic or beyond. The specific problem that a D-Wave system solves can be expressed as

$$\arg\min_{\boldsymbol{\sigma}} \mathscr{H}(\boldsymbol{\sigma}), \text{ where } \mathscr{H}(\boldsymbol{\sigma}) = \sum_{i=1}^{N} h_i \sigma_i + \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} J_{i,j} \sigma_i \sigma_j \qquad (2.1)$$

In the above, $\sigma_i \in \{-1, +1\}$, $h_i \in \mathbb{R}$ and $J_{i,j} \in \mathbb{R}$. In other words, $\mathscr{H}$ is a pseudo-Boolean function of degree 2. Physically, the h_i represents the strength of the external field applied to σ_i, and the $J_{i,j}$ represents the strength of the interaction between σ_i and σ_j. Given a set of h_i and $J_{i,j}$, finding the σ_i that minimize $\mathscr{H}(\boldsymbol{\sigma})$ in equation (2.1) is an NP-hard problem (Barahona 1982). Consequently, an efficient (i.e., polynomial-time) classical algorithm for finding these σ_i in the general case is expected not to exist. The best-known classical algorithms run in exponential time, which is intractable for large N. (On a D-Wave 2000Q system, $N \approx 2,000$.) Nevertheless, contemporary D-Wave systems can propose a solution in *microseconds*, which is an impressive capability.

A program for a quantum annealer is merely a list of h_i and $J_{i,j}$ for equation (2.1). Clearly, there is a huge semantic gap between such a list and CLP. Perhaps surprisingly, we show in Section 4 that it is indeed possible to map CLP problems into Ising-model Hamiltonians.

An important point regarding equation (2.1) is that it represents a *classical* Hamiltonian. In contrast to gate-model quantum computers, in which the programmer directly controls the application of quantum-mechanical effects, these effects are almost entirely hidden from the user of a quantum annealer. Hence, the approach this paper presents is equally applicable to classical annealers such as Hitachi's CMOS annealer (Yamaoka *et al.* 2016), Fujitsu's Digital Annealer (Fujitsu Ltd. 2017) or even all-software implementations of simulated annealing (Kirkpatrick *et al.* 1983). We focus our discussion on quantum annealers, however, because such devices offer the potential of converging to an optimal solution with higher probability than can classical annealing methods (Kadowaki and Nishimori 1998).

2.2 D-Wave hardware

D-Wave systems, Inc. is a producer of commercial quantum annealers. Although their hardware performs the basic quantum-annealing task described in Section 2.1, engineering reality imposes a number of constraints on the specific Hamiltonians that can be expressed:

- As stated above, only two-local Hamiltonians are supported. Three-local Hamiltonians and beyond can be converted to two-local Hamiltonians at the cost of additional spins.
- Even though, nominally, $h_i, J_{i,j} \in \mathbb{R}$, those coefficients in fact have finite precision and are limited to relatively few distinct values.
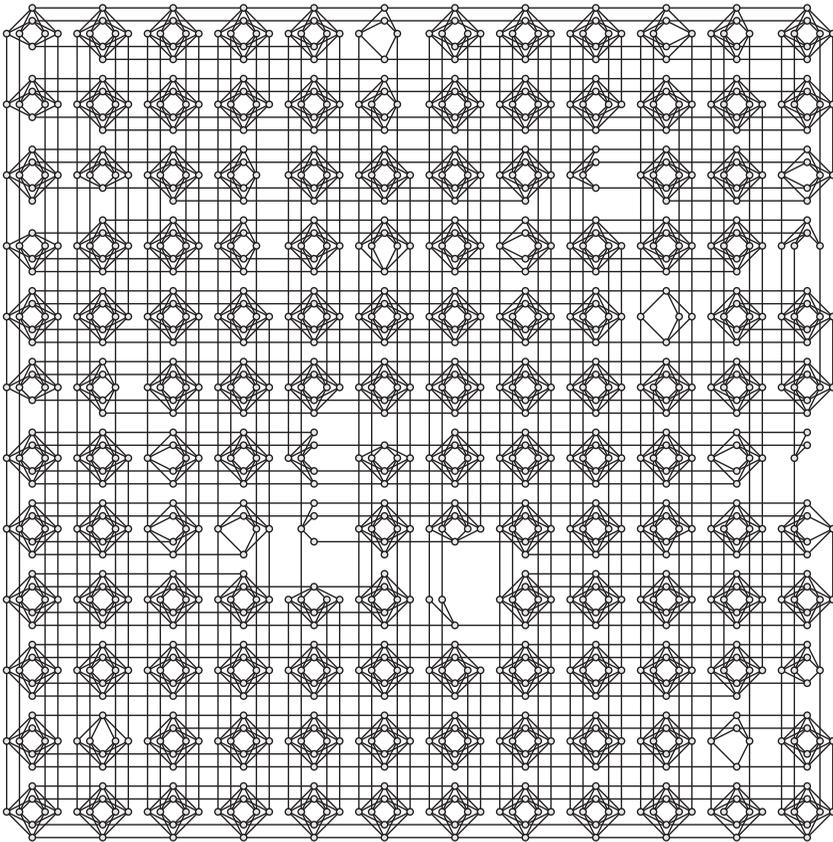
Fig. 1. Physical topology of LANL's D-Wave 2X.

- Although ideally, there should be a $J_{i,j}$ for any $1 \leqslant i < j \leqslant N$, in practice, the system's physical topology, called a *Chimera graph* (Bunyk *et al.* 2014), provides limited connectivity: at most six $J_{i,j}$ for any given i.
- On top of the preceding constraint, in any given installation, a fraction of the h_i and $J_{i,j}$ will be inoperative. (See below.)

Figure 1 illustrates the physical topology of Ising, the D-Wave 2X system installed at Los Alamos National Laboratory that was used for all of the experiments reported in this article. Edges represent the $J_{i,j}$, and nodes represent the h_i (and σ_i). Physically, graph nodes are superconducting flux qubits, implemented as niobium rings that are written and read electromagnetically (Johnson *et al.* 2011). At superconducting temperatures—Ising operates at a mere 10.45 mK (0.01℃ above absolute zero)— quantum-mechanical effects (entanglement, superposition and quantum tunneling) come into play. In Ising, 1,095 (95.1%) of the qubits and 3,061 (91.1%) of the couplers are operational.

Annealing times are also installation-dependent. Ising supports annealing times of 5–2,000 μs (user-selectable). Longer annealing times are theoretically more likely to find the input Hamiltonian's global minimum, but shorter annealing times enable more attempts per unit time.

Fig. 2. Sample gate-model program (Grover's method).

3 Related work

Very little exists in terms of programming models and programming languages for gate-model quantum computers and quantum annealers.

Programming models for gate-model quantum computers: Although gate-model quantum computers are more heavily studied than quantum annealers, virtually all programming languages and compilers developed for these systems are based on the same underlying programming model: Place a specified gate at a specified location in a circuit, which is a directed acyclic graph. To illustrate this approach, Figure 2 presents an example of Grover's search algorithm (Grover 1996) in standard gate-model circuit notation.[2] Each gate (e.g., X, H or $CNOT$ [●—⊕]) represents a small unitary matrix that transforms the current state—in this case, a complex vector with 8 (2^3) elements.

The question is how to express algorithms such as the one shown in Figure 2 as a computer code. OpenQASM (Cross *et al.* 2017) is a domain-specific language in which the "statements" (x, h, cx, etc.) represent the application of a gate to one or more qubits, and programs can define parameterized macros to simplify repeated tasks. Scaffold (JavadiAbhari *et al.* 2015) is similar but also supports C-style for loops to apply multiple gates in a fixed pattern. It further includes support for expressing classical oracles as gates. Quipper (Green *et al.* 2013) and LIQ$Ui|\rangle$ (Wecker and Svore 2014) are embedded domain-specific languages, with the former embedded in Haskell and the latter embedded in F#, implying they have access to those languages' code features. Like OpenQASM and Scaffold, Quipper and LIQ$Ui|\rangle$ are based primarily on specifying an ordered sequence of gates applied to qubits, although in a functional-language context (e.g., a Quipper circuit runs within a Haskell monad). Quil (Smith *et al.* 2017) and Q# (Microsoft Corp. 2017) are recent domain-specific languages that, while sharing the same "ordered sequence of gates" abstraction as all the other efforts, put particular emphasis on tightly interleaving classical and quantum computation.

In short, the state-of-the-art in programming models for gate-model quantum computers is at the level of an assembly language that offers various convenience features but little in the way of higher level abstractions. The closest equivalent in the context of quantum annealing is perhaps QMASM (Pakin 2016), which we introduce in Section 4.2 but then build upon to present a more expressive programming model

---

[2] The dashed box delineates the user-provided "oracle" function, which indicates if a set of inputs (2 bits, in this case) represents the item being searched for by flipping an ancilla qubit.

based on classical circuits and, on top of that, one based on predicate logic. Unlike all of the works discussed above, in this article, we establish a large semantic gap between the programming model exposed by the underlying hardware and that presented to the user.

Quantum Prolog: James *et al.* (2011) demonstrate that one can express the equivalent of a pure version of Prolog over finite relations in terms of a model of *discrete* quantum computing. (Conventionally, quantum computing is defined over Hilbert spaces, which are complex-valued.) This is one of the very few attempts to develop a high-level programming model for gate-model quantum computers, although for a form that has never been implemented. Besides targeting gate-model quantum computers rather than quantum annealers, this work differs from ours in that it focuses on the *mathematical equivalency* of relational programming and discrete quantum computing over the field of Booleans, while our work showcases the *implementation* of a Prolog compiler that generates code suitable for running on a physical quantum annealer.

Programming models for quantum annealers: Most of the literature that relates to programming quantum annealers focuses on expressing a single algorithm or single class of algorithms in terms of an Ising-model Hamiltonian (equation (2.1)). Lucas (2014) surveys a number of such algorithms. More recent efforts include traveling-salesman problems (Heim *et al.* 2017) and satisfiability problems (Hen and Young 2011). In contrast, our work is to make it possible to express problems without explicitly specifying individual h_i and $J_{i,j}$ coefficients.

D-Wave systems, Inc. provides a few tools that let one express problems in a higher level form than a list of h_i and $J_{i,j}$ coefficients. ToQ (D-Wave Systems, Inc. 2017b), formerly Deqo (Dahl 2014), is the most related tool to our quantum-annealing Prolog in that it is centered on constraint satisfaction and targets a quantum annealer. ToQ accepts a set of constraints and returns a set of values that satisfy those constraints. However, each individual constraint is evaluated classically and exhaustively; the D-Wave system is used only for combining individually satisfied constraints into a global problem and solving that. In contrast, our Prolog implementation performs *all* of its constraint satisfaction on the quantum annealer, not classically.

4 Implementation

4.1 Primitives

Tables 1 and 2 present the primitive mechanisms that one can use to represent a problem as an Ising-model Hamiltonian of the form stated in equation (2.1). Interpreting $\sigma_i = -1$ as FALSE and $\sigma_i = +1$ as TRUE, a negative h_i expresses a preference for the corresponding σ_i being TRUE (Table 1), while a positive h_i expresses a preference for the corresponding σ_i being FALSE (Table 1). The magnitude of the h_i corresponds to the strength of the preference. A negative $J_{i,j}$ expresses a preference for the corresponding σ_i and σ_j having the same value (Table 2), while a positive $J_{i,j}$ expresses a preference for the corresponding σ_i and σ_j having opposite values

Table 1. *Effect of negative and positive* h_i

(a) $h_i < 0$: Favor TRUE		
σ_i	$-1\,\sigma_i$	$\arg\min_\sigma$
-1	$+1$	
$+1$	-1	✔
(b) $h_i > 0$: Favor FALSE		
σ_i	$+1\,\sigma_i$	$\arg\min_\sigma$
-1	-1	✔
$+1$	$+1$	

Table 2. *Effect of negative and positive* $J_{i,j}$

(a) $J_{i,j} < 0$: Favor equality			
σ_i	σ_j	$-1\,\sigma_i\sigma_j$	$\arg\min_\sigma$
-1	-1	-1	✔
-1	$+1$	$+1$	
$+1$	-1	$+1$	
$+1$	$+1$	-1	✔
(b) $J_{i,j} > 0$: Favor inequality			
σ_i	σ_j	$+1\,\sigma_i\sigma_j$	$\arg\min_\sigma$
-1	-1	$+1$	
-1	$+1$	-1	✔
$+1$	-1	-1	✔
$+1$	$+1$	$+1$	

(Table 2). In other words, $J_{i,j} < 0$ can be interpreted as a wire in a digital circuit, and $J_{i,j} > 0$ can be interpreted as an inverter. As with the h_i, the magnitude of the $J_{i,j}$ corresponds to the strength of the preference.

One can set up a system of inequalities to find h_i and $J_{i,j}$ values with a desired set (or sets) of σ_i in the ground state. This system of inequalities can be solved by hand in simple cases or with a constraint solver for more complicated cases. We use MiniZinc (Nethercote *et al.* 2007) as our constraint-modeling language, but any similar system would suffice. Table 3 presents an Ising-model Hamiltonian, $\mathcal{H}_\wedge(\boldsymbol{\sigma})$, whose four-way degenerate ground state (meaning, the four-way tie for the minimum value) is exactly the set of spins for which $\sigma_k = \sigma_i \wedge \sigma_j$ (i.e., logical conjunction). The Hamiltonian was found by constraining the ground-state $\mathcal{H}_\wedge(\boldsymbol{\sigma})$ all to have the same value and all other $\mathcal{H}_\wedge(\boldsymbol{\sigma})$ to have a strictly greater value. Similarly, Table 3 presents an Ising-model Hamiltonian, $\mathcal{H}_\vee(\boldsymbol{\sigma})$, whose four-way degenerate ground state is exactly the set of spins for which $\sigma_k = \sigma_i \vee \sigma_j$ (i.e., logical disjunction).

Table 3. *Representing simple Boolean functions as Ising-model Hamiltonians*

(a) Logical conjunction (AND)

σ_i	σ_j	σ_k	$\mathscr{H}_\wedge(\boldsymbol{\sigma})$	arg min$_\sigma$
−1	−1	−1	−1.5	✔
−1	−1	+1	4.5	
−1	+1	−1	−1.5	✔
−1	+1	+1	0.5	
+1	−1	−1	−1.5	✔
+1	−1	+1	0.5	
+1	+1	−1	0.5	
+1	+1	+1	−1.5	✔

$$\mathscr{H}_\wedge(\boldsymbol{\sigma}) = -0.5\sigma_i + -0.5\sigma_j + 1.0\sigma_k + 0.5\sigma_i\sigma_j + -1.0\sigma_i\sigma_k + -1.0\sigma_j\sigma_k$$

(b) Logical disjunction (OR)

σ_i	σ_j	σ_k	$\mathscr{H}_\vee(\boldsymbol{\sigma})$	arg min$_\sigma$
−1	−1	−1	−1.5	✔
−1	−1	+1	0.5	
−1	+1	−1	0.5	
−1	+1	+1	−1.5	✔
+1	−1	−1	0.5	
+1	−1	+1	−1.5	✔
+1	+1	−1	4.5	
+1	+1	+1	−1.5	✔

$$\mathscr{H}_\vee(\boldsymbol{\sigma}) = 0.5\sigma_i + 0.5\sigma_j + -1.0\sigma_k + 0.5\sigma_i\sigma_j + -1.0\sigma_i\sigma_k + -1.0\sigma_j\sigma_k$$

It is worth noting that Hamiltonians are additive. Specifically, the ground state of $\mathscr{H}_A + \mathscr{H}_B$ is the intersection of the ground state of $\mathscr{H}_A$ and the ground state of $\mathscr{H}_B$. The implication is that primitive operations like those shown in Table 3 can be trivially combined into more complicated expressions without having to solve a (computationally expensive) constraint problem for the overall expression. Because Ising-model Hamiltonians always have a non-empty ground state—they must have at least one minimum value—summing Hamiltonians whose ground states have an empty intersection can lead to an unintuitive ground state of the combined Hamiltonian. As a trivial example, $\mathscr{H}_A = \sigma_a - \sigma_b$ has the unique ground state $\{\sigma_a = -1, \sigma_b = +1\}$; $\mathscr{H}_B = -\sigma_a\sigma_b$ has the two-fold degenerate ground state $\{\sigma_a = -1, \sigma_b = -1\}$ and $\{\sigma_a = +1, \sigma_b = +1\}$; but $\mathscr{H}_A + \mathscr{H}_B$ has the *three-fold* degenerate ground state $\{\sigma_a = -1, \sigma_b = -1\}$, $\{\sigma_a = -1, \sigma_b = +1\}$ and $\{\sigma_a = +1, \sigma_b = +1\}$.

In this work, however, we ensure by construction that we do not sum Hamiltonians whose intersection is empty. Specifically, because we sum Hamiltonians only for Boolean expressions and only by equating (cf. Table 2) outputs to inputs and never outputs to outputs or inputs to inputs (i.e., we rely on a directed acyclic graph organization), we will not normally wind up with an empty intersection. The exception is in the case in which spins are "pinned" to values that introduce

!begin_macro and	**!begin_macro** or
A −0.5	A 0.5
B −0.5	B 0.5
Y 1.0	Y −1.0
A B 0.5	A B 0.5
A Y −1.0	A Y −1.0
B Y −1.0	B Y −1.0
!end_macro and	**!end_macro** or
(a) QMASM version of Table 3(a)	(b) QMASM version of Table 3(b)

Fig. 3. QMASM macro definitions for AND and OR. (a) QMASM version of Table 3,
(b) QMASM version of Table 3.

unsatisfiable output–output or input–input couplings, as discussed on Section 4.2 in the following section.

Armed with Hamiltonians for AND (Table 3), OR (Table 3) and NOT (Table 2) and the knowledge that Hamiltonians can be added together to produce new, more constrained Hamiltonians, we can implement a complete zeroth-order logic.

4.2 QA prolog

We have implemented a Prolog compiler that compiles Prolog programs to a list of h_i and $J_{i,j}$ coefficients for use with equation (2.1). We call our implementation *quantum-annealing Prolog* or *QA Prolog* for short. Although the underlying concept is no more sophisticated than what was described in Section 4.1, a large software-engineering effort was needed to bridge the gap between what was presented there and a usable Prolog compiler.

Internally, QA Prolog comprises a number of software layers. The lowest layer is a "quantum macro assembler" we developed called QMASM (Pakin 2016).[3] QMASM provides a thin but convenient layer of abstraction atop equation (2.1). It lets programs refer to spins symbolically rather than numerically, shields the program from having to consider the specific underlying physical topology and provides modularization through the use of macros that can be defined once and instantiated repeatedly.

As an example, Figure 3 shows how one can define and and or macros corresponding to the Hamiltonians portrayed by Table 3. Lines containing a single symbol and a value correspond to an h_i, and lines containing two symbols and a value correspond to a $J_{i,j}$. Macros can be instantiated using the **!use_macro** directive.

The QMASM code in Figures 3(a) and (b) is compiled to a physical Hamiltonian, i.e., one that uses only the h_i and $J_{i,j}$ that exist on the specific underlying hardware, with all coefficients scaled to the supported range. The D-Wave's physical,

---

[3] QMASM is freely available from `https://github.com/lanl/qmasm`.

```
module silly (this, that, the_other);
    input [3:0] this, that;
    output [4:0] the_other;
    wire [4:0] this2, that2;

    assign this2 = (this[0] == 1'b1) ? this*5'd2 : this;
    assign that2 = (that[0] == 1'b1) ? that*5'd2 : that;
    assign the_other = this2 + that2;
endmodule
```

Fig. 4. Sample Verilog program.

Chimera-graph topology is not only fairly sparse but also contains no odd-length cycles—needed for the two A–B–Y cycles in Figure 3, for example—implying that a typical Hamiltonian must be *embedded* (Choi 2008) into the physical topology. Doing so requires that additional spins and additional terms be added to the Hamiltonian and slightly alters the coefficients. For example, Figure 4.2 may compile to $\mathcal{H}_\wedge(\boldsymbol{\sigma}) = -0.125\sigma_8 - 0.25\sigma_9 + 0.5\sigma_{14} - 0.125\sigma_{15} - 0.5\sigma_8\sigma_{14} - 0.5\sigma_9\sigma_{14} - 1.0\sigma_8\sigma_{15} + 0.25\sigma_9\sigma_{15}$. A benefit of QMASM is that it lets programs work with arbitrary two-local Ising Hamiltonians—support for higher order interaction terms may be added in the future—while it automatically maps those Hamiltonians onto the available hardware.

Given that we can implement Boolean functions as Hamiltonians, we can take a large leap in abstraction and programmability and map Verilog programs (Thomas and Moorby 2002) into the form of equation (2.1). Verilog is a popular hardware-description language. Unlike QMASM, which looks foreign to a conventional-language programmer, the Verilog language supports variables, arithmetic operators, conditionals, loops and other common programming-language constructs.

Verilog is a good match for current quantum annealers because, unlike most programming languages, it provides precise control over the number of bits used by each variable. With a total of only a few thousand bits (a few hundred bytes) available for both code and data *combined* on contemporary quantum annealers, there is no room for waste. Figure 4 presents an example of a Verilog module that inputs two 4-bit variables, *this* and *that*, and outputs a 5-bit variable, *the_other*, which is a function of the two inputs. Although this program does not perform a useful function, it works well for pedagogical purposes because it employs a variety of language features including internal variables (defined with **wire**), a relational operator (==), a C-style ternary conditional (?:), multiplication, addition and assignment.

We use Yosys (Wolf and Glaser 2013), an open-source hardware-synthesis tool, to compile Verilog code to an Electronic Design Interchange Format netlist (Kahn *et al.* 2000), a precise, machine-parseable description of a digital circuit. In addition to compiling, Yosys performs a number of optimizations on the design and, with the help of ABC (Brayton and Mishchenko 2010; Berkeley Logic Synthesis and Verification Group 2016), transforms the design to use only a relatively small set of

basic gates. A program we developed, edif2qmasm,[4] translates the Electronic Design Interchange Format netlist to a QMASM Hamiltonian. The generated code employs a standard-cell library of precomputed Hamiltonians for AND, OR, NOT, XOR and other such primitives.

In the case of Figure 4, the generated Hamiltonian comprises 263 σ_i and h_i and 375 $J_{i,j}$. We can "pin" values to the *this* and *that* inputs by specifying the h_i as in Table 1, run the Hamiltonian on a quantum annealer, and read out the *the_other* output. (QMASM even supports pinning directly on the command line (Pakin 2016).) We can alternatively pin the *the_other* output, run the Hamiltonian and read out the two inputs; pin the output and one of the inputs and read out the other input or pin nothing and read out valid mappings of inputs to outputs. (As could be expected, though, if the function were non-surjective, specifying an output that is not the image of any element in the function's domain will not result in valid inputs.) In essence, we support a relational semantics in a language that does not normally offer such a capability.

The final piece of QA Prolog is the Prolog compiler itself. QA Prolog supports only a subset of Prolog but enough to handle basic CLP. For instance, QA Prolog supports atoms and positive integers but not floating-point numbers, strings, lists, first-class compound terms or any other data type. It supports arithmetic and relational operations but not ! (cut), `fail` or any impure predicates. Clauses can reference other clauses but not recursively. Polymorphic clauses are not supported, but `integer/1` and `atom/1` can be used to disambiguate otherwise polymorphic clauses. QA Prolog does support unification (Robinson 1965), backtracking and predicates comprising multiple clauses.

Importantly, QA Prolog allows the goals in a rule's body to be specified in any order without impacting their ability to be proven. In particular, operations can be performed on variables even before they are ground. This is in contrast to basic Prolog—as opposed to Prolog with the `library(clpfd)` predicates (Triska 2012)—which is limited in its ability to manipulate free variables.

After the usual lexing and parsing steps, the compiler performs type inference on the abstract syntax tree. Because so few spins are available in current hardware, distinguishing between the two supported data types lets the compiler represent each of them using a different numbers of bits: $\lceil \log_2(a) \rceil$ bits for atoms, assuming a distinct atoms are named in the program, and $\lceil \log_2(n) \rceil$ bits for integers, assuming the largest integer appearing literally in the program is n. A QA Prolog command-line argument lets the user increase the number of bits per integer in case larger values are needed for intermediate results.

Once all types are inferred, QA Prolog's code generator generates Verilog code. Although compiling Prolog to Verilog appears, on the surface, to be a peculiar strategy, recall that

(1) our quantum-annealing implementation of Verilog already has a relational semantics,

---

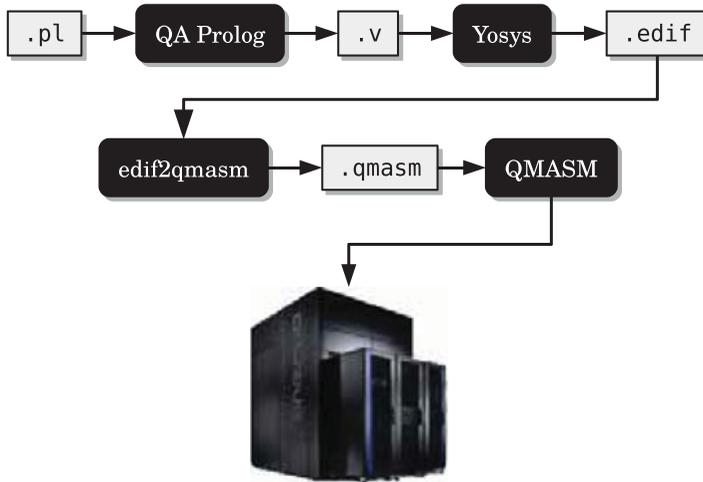[4] edif2qmasm is freely available from `https://github.com/lanl/edif2qmasm`.

Fig. 5. Overall QA Prolog processing flow.

(2) Verilog provides arithmetic and relational operations, saving QA Prolog from having to implement those itself,

(3) hardware-synthesis tools such as Yosys perform a number of logic optimizations and simplifications on QA Prolog's behalf, and

(4) QA Prolog gets unification support "for free" because the same spins are used to represent every instance of the same variable appearing in the Verilog code.

In the compilation from Prolog to Verilog, each predicate (including all clauses) is converted to a single Verilog module. Because the names of Verilog arguments must be unique, arguments are renamed. In Prolog terms, this is like replacing the fact "same(X, X)." with the rule "same(A, B) :- A = B.." In addition, an extra, single-bit argument called Valid is included in the argument list. This is an output value that is set to 1 if and only if all goals are proved. Queries that include variables are implemented by pinning Valid to 1 and letting the annealing process solve for all the variables so as not to violate that condition. Queries that do not include variables leave Valid unpinned and let the annealing process find a value for it that does not violate any other conditions.

Once QA Prolog has compiled the Prolog source code—including the query, which can be specified on the command line—to Verilog, QA Prolog invokes Yosys, edif2qmasm and QMASM, as illustrated in Figure 5. With the help of D-Wave's SAPI library (D-Wave Systems, Inc. 2017a), QMASM remotely executes the user's program on a D-Wave system and reports the (Boolean) value of each symbol appearing in the QMASM source file. QA Prolog maps these lists of Booleans back to integers and named atoms, associates those values with variables named in the user's query, and reports all variables and their values to the user just like a typical Prolog environment would.

A quantum annealer *always* returns a vector of spins (σ), there is no notion of "no result found." Although QMASM can detect certain obviously incorrect results and discard them, in the general case, QA Prolog can return incorrect results. Consider

---

and(false, false, false).
and(false, true, false).
and(true, false, false).
and(true, true, true).

and(A, B, C, Y) :—
        and(A, B, AB),
        and(AB, C, Y).

Fig. 6. Three-input AND.

the CLP, "`impossible(X) :- X < 4, X > 4..`" Although `impossible(X)` should never succeed, QA Prolog proposes all eight 3-bit integers as possible solutions, as all eight are equally bad choices. We do not consider this odd behavior a showstopper because many important problems in P and NP have solutions that can be (classically) verified quickly. A user can run QA Prolog to quickly produce a set of candidate solutions then filter out invalid ones as part of a post-processing step.

5 Evaluation

In this section, we examine what QA Prolog is and is not capable of expressing. Section A expands upon this section by presenting all of the transformations a particular program undergoes from Prolog source code to a two-local Ising Hamiltonian. Our main metric in this section is the cost in spins for various programs. Figure 6 presents a Prolog program that defines an `and/3` predicate in terms of an `and/2` predicate, which itself is defined by four facts. Note that this example does not involve any CLP.

Compiling Figure 6 with QA Prolog results in a logical Hamiltonian with 26 spins, and QMASM assembles that into a physical Hamiltonian with 42 spins. For comparison, a hand-constructed logical Hamiltonian for a three-input AND requires only five spins,[5] and QMASM assembles that into a physical Hamiltonian containing 10 spins.

Next, consider a Prolog program that finds two positive integers whose sum and product are each 4 (Figure 7). When run with the query "`fours(A, B)`," the program correctly produces "`A = 2, B = 2`." The program in fact additionally produces "`A = 6, B = 6`." Because 3 is the minimum number of bits needed to represent all integers appearing literally in Figure 7, QA Prolog represents all numbers with that many bits. Because $6 + 6 \equiv 4 \pmod{2^3}$ and $6 \cdot 6 \equiv 4 \pmod{2^3}$, the second solution is valid, albeit a bit surprising.

---

[5] $\mathcal{H}_{\text{and}/3}(\boldsymbol{\sigma}) = -\frac{1}{2}\sigma_A - \frac{1}{2}\sigma_B - \frac{1}{2}\sigma_C + \sigma_Y + \frac{1}{2}\sigma_x + \frac{1}{2}\sigma_A\sigma_B - \sigma_A\sigma_x - \sigma_B\sigma_x - \sigma_C\sigma_Y + \frac{1}{2}\sigma_C\sigma_x - \sigma_Y\sigma_x$, where σ_x is an ancilla spin needed to produce the correct ground state.

$$\text{fours(A, B)} :-$$
$$\text{A+B} = 4,$$
$$\text{A*B} = 4.$$

Fig. 7. Find two numbers that both add and multiply to 4.

Even a program this small consumes a large number of spins, primarily because of the 3-bit multiplication. Specifically, it maps to 24 logical spins at the QMASM level, which in turn get mapped to 39 physical spins or 3.6% of the total available on Los Alamos National Laboratory's D-Wave 2X system. The number of logical spins is a function of both the compilation tools (Yosys and ABC, in this case) and the set of precomputed Hamiltonians included in edif2qmasm's standard-cell library. The number of physical spins is a function of the both the physical topology and the minor-embedding algorithm—and the random-number generator's state in the case of a stochastic embedder such as the one we use (Cai *et al.* 2014).

The total end-to-end execution time, including all of the compilation steps and the network connection to the D-Wave 2X, averages 3.0 ± 1.0 s over 100 trials. For each trial, we specified an annealing time of 20 µs and had the hardware perform 1,000 anneals, enabling a maximum of 1,000 unique solutions to be returned. Hence, a fixed 20 ms of the end-to-end time is the computation proper—the actual annealing time. Although the end-to-end time is high, our belief is that this time will grow slower with program complexity than would a classical implementation. Remember, all goals in the entire program are effectively evaluated in parallel. Not only are CLP semantics honored, but there is no additional performance cost for going beyond plain Prolog's unification capabilities.

What are the limits on the Prolog programs that can be run on a D-Wave 2X using QA Prolog? We found that the "light meal" example from Dutra's CLP tutorial (Dutra 2010) is too large to fit, but if we skip dessert, as in Figure 8, the code runs, with 170 logical spins dilating to 602 physical spins. As a more controlled experiment, consider a `mult/3` predicate defined as "`mult(A, B, C) :- C = A*B.`" Because of QA Prolog's support for CLP, we can provide a value only for `C` in a query to factor `C` into `A` and `B`. The reader should note that this one-line integer-factoring program for a quantum annealer is conceptually far simpler than Shor's famous integer-factoring algorithm for gate-model quantum computers (Shor 1999), which requires knowledge of both quantum mechanics and number theory to understand. By factoring a relatively small number, say 6, we can steadily increase the bit width and measure the number of logical and physical spins required to represent the program.

Figure 9 presents the results of this study. The number of logical spins for a given bit width does not change from compilation to compilation, but the number of physical spins does because it relies on a stochastic minor-embedding algorithm (Cai *et al.* 2014). Consequently, the figure includes error bars for physical spin counts. Points represent measurements, lines represent regressions. The regression curves

---

```
light_meal(S, M) :−
        Skc + Mkc =< 8,
        starter(S, Skc),
        main_course(M, Mkc).

meat(steak, 5).
meat(pork, 7).
fish(sole, 2).
fish(tuna, 4).

main_course(M, Mkc) :− meat(M, Mkc).
main_course(M, Mkc) :− fish(M, Mkc).

starter(salad, 1).
starter(soup, 6).
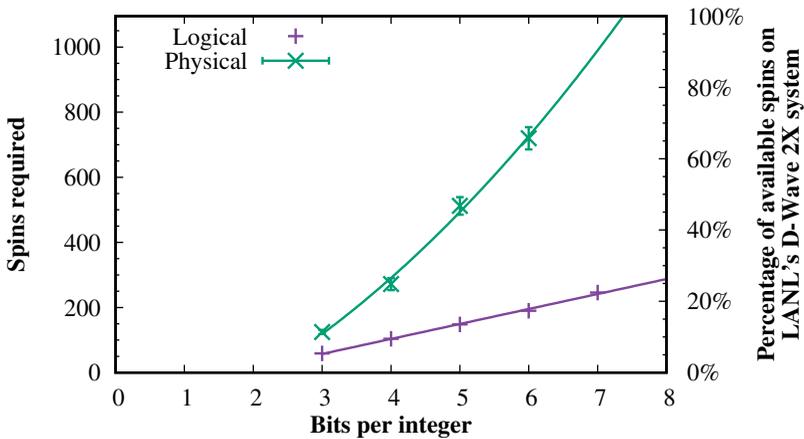```

---

Fig. 8. Planning a meal of no more than 8 kcal.



Fig. 9. Cost in spins for "`mult(P, Q, 6)`" with different integer bit widths.

used are $f_{\text{Log}}(x) = 46x − 80.6$ and $f_{\text{Phys}}(x) = 15.2x^2 + 65.72x − 215.44$. Both have a coefficient of determination $R^2 > 0.996$. As Figure 9 indicates, the number of physical spins grows faster than the number of logical spins. For integer factorization, we run out of physical spins at 7 bits per integer.

## 6 Conclusions

Quantum annealers represent a radical departure from conventional computer architectures. Rather than perform a sequence of operations that modify state (registers and memory), a quantum annealer performs in hardware a particular type of NP-hard optimization problem. Specifically, it finds a set of Boolean values (spins) that minimize a real-valued, fixed-form function of a potentially large number of

variables. In effect, a quantum annealer evaluates the function in parallel for all $2^N$ possible inputs and reports in a fixed length of time (microseconds) the best instance found. The catch is that the solution is not guaranteed to be optimal.

Although quantum annealers offer the potential for huge performance gains through massive effective parallelism, programming them can be a challenge. At the lowest level, a program for a quantum annealer is merely a list of coefficients for the aforementioned fixed-form pseudo-Boolean function. The question we sought to answer in this work is, *Can one express CLP in the form accepted by quantum-annealing hardware?* The key insight we make in answering that question is identifying an analogy between expression minimization in an Ising-model Hamiltonian and unification in CLP.

Based on that insight we implemented QA Prolog, a compiler that, through a sequence of transformations, converts Prolog programs into two-local Ising-model Hamiltonians, runs these on a D-Wave quantum annealer, and reports the results in terms of program variables. We draw the following conclusions from our study:

(1) Despite the enormous semantic gap, it is indeed possible to automatically convert CLP, expressed in a subset of Prolog, to the solution to an optimization problem, expressed as coefficients to a two-local Ising-model Hamiltonian.
(2) An important limiting factor is the number of spins needed to express even trivial constraint logic problems. Our experimental platform, a D-Wave 2X quantum annealer installed at Los Alamos National Laboratory, has 1,095 spins. One can think of those 1,095 spins as corresponding to roughly 136 bytes of memory, which need to hold all program inputs, outputs and logic.
(3) End-to-end performance (i.e., including compilation time) is poor: A few seconds for even trivial CLP problems. Although we expect these times to rise slowly with problem size and complexity, we cannot confirm that hypothesis or compare it to classical implementations until we can execute programs sufficiently large so as to challenge classical CLP systems.

Even considering the preceding shortcomings, we remain optimistic about the potential of exploiting the massive effective parallelism provided by quantum annealers to accelerate the execution of CLP. Although the hardware is in its early generations, when increased scale and other engineering improvements are put into place, QA Prolog will be ready to take advantage of these advances.

## References

BARAHONA, F. 1982. On the computational complexity of Ising spin glass models. *Journal of Physics A: Mathematical and General 15,* 10, 3241.

BERKELEY LOGIC SYNTHESIS AND VERIFICATION GROUP. 2016. ABC: A system for sequential synthesis and verification. URL: `http://www.eecs.berkeley.edu/~alanmi/abc/`. [Accessed on April 24, 2018].

BRAVYI, S., BESSEN, A. J. AND TERHAL, B. M. 2006. Merlin-Arthur games and stoquastic complexity. *arXiv:quant-ph/0611021v2*.

BRAVYI, S. AND HASTINGS, M. 2017. On complexity of the quantum Ising model. *Communications in Mathematical Physics 349,* 1, 1–45.

BRAYTON, R. AND MISHCHENKO, A. 2010. ABC: An academic industrial-strength verification tool. In *Proc. 22nd International Conference on Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds. Springer, Berlin, Heidelberg, Edinburgh, UK, 24–40.

BUNYK, P. I., HOSKINSON, E. M., JOHNSON, M. W., TOLKACHEVA, E., ALTOMARE, F., BERKLEY, A. J., HARRIS, R., HILTON, J. P., LANTING, T., PRZYBYSZ, A. J. AND WHITTAKER, J. 2014. Architectural considerations in the design of a superconducting quantum annealing processor. *IEEE Transactions on Applied Superconductivity 24,* 4, 1–10.

CAI, J., MACREADY, B. AND ROY, A. 2014. A practical heuristic for finding graph minors. *arXiv:1406.2741 [quant-ph]*.

CHOI, V. 2008. Minor-embedding in adiabatic quantum computation: I. The parameter setting problem. *Quantum Information Processing 7,* 5, 193–209.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L. AND STEIN, C. 2001. *Introduction to Algorithms*, 2nd ed. MIT Press.

CROSS, A. W., BISHOP, L. S., SMOLIN, J. A. AND GAMBETTA, J. M. 2017. Open quantum assembly language. *arXiv:1707.03429 [quant-ph]*.

D-Wave Systems, Inc. 2017a. *Developer Guide for Python*. D-Wave Systems, Inc., Burnaby, British Columbia, Canada.

D-Wave Systems, Inc. 2017b. *ToQ Overview*. D-Wave Systems, Inc., Burnaby, British Columbia, Canada. ToQ documentation, qOp version 2.3.1.

DAHL, E. D. 2014. *Deqo: A Direct Embedding Quantum Optimizer*. D-Wave Systems, Inc.

DUTRA, I. 2010. Constraint logic programming: A short tutorial. URL: https://www.dcc.fc.up.pt/~ines/talks/clp-v1.pdf. [Accessed on April 24, 2018].

FARHI, E. AND GUTMANN, S. 1998. Analog analogue of a digital quantum computation. *Physical Review A 57*, 2403–2406.

FEYNMAN, R. P. 1986. Quantum mechanical computers. *Foundations of Physics 16,* 6, 507–531.

FINNILA, A. B., GOMEZ, M. A., SEBENIK, C., STENSON, C. AND DOLL, J. D. 1994. Quantum annealing: A new method for minimizing multidimensional functions. *Chemical Physics Letters 219,* 5, 343–348.

FUJITSU LABORATORIES LTD. 2017. Fujitsu Laboratories Develops New Architecture that Rivals Quantum Computers in Utility. Press release, 20 October 2016, Kawasaki, Japan. URL: http://www.fujitsu.com/global/about/resources/news/press-releases/2016/1020-02.html. [Accessed on April 24, 2018].

GANSNER, E. R. AND NORTH, S. C. 2000. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience 30,* 11, 1203–1233.

GREEN, A. S., LUMSDAINE, P. L., ROSS, N. J., SELINGER, P. AND VALIRON, B. 2013. Quipper: A scalable quantum programming language. In *Proc. of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, USA, 333–342.

GROVER, L. K. 1996. A fast quantum mechanical algorithm for database search. In *Proc. of the 28th Annual ACM Symposium on Theory of Computing*. ACM, New York, NY, USA, 212–219.

HEIM, B., BROWN, E. W., WECKER, D. AND TROYER, M. 2017. Designing adiabatic quantum optimization: A case study for the traveling salesman problem. *arXiv:1702.06248v1 [quant-ph]*.

HEN, I. AND YOUNG, A. P. 2011. Exponential complexity of the quantum adiabatic algorithm for certain satisfiability problems. *Physical Review E 84*, 061152.

JAMES, R. P., ORTIZ, G. AND SABRY, A. 2011. Quantum computing over finite fields. *arXiv:1101.3764 [quant-ph]*.

JavadiAbhari, A., Patil, S., Kudrow, D., Heckey, J., Lvov, A., Chong, F. T. and Martonosi, M. 2015. ScaffCC: Scalable compilation and analysis of quantum programs. *Parallel Computing 45*, 2–17.

Johnson, M. W., Amin, M. H. S., Gildert, S., Lanting, T., Hamze, F., Dickson, N., Harris, R., Berkley, A. J., Johansson, J., Bunyk, P., Chapple, E. M., Enderud, C., Hilton, J. P., Karimi, K., Ladizinsky, E., Ladizinsky, N., Oh, T., Perminov, I., Rich, C., Thom, M. C., Tolkacheva, E., Truncik, C. J. S., Uchaikin, S., Wang, J., Wilson, B. and Rose, G. 2011. Quantum annealing with manufactured spins. *Nature 473,* 7346, 194–198.

Kadowaki, T. and Nishimori, H. 1998. Quantum annealing in the transverse Ising model. *Physical Review E 58*, 5355–5363.

Kahn, H., La Fontaine, R. and Lau, R. 2000. Electronic design interchange format (EDIF): Part 2: Version 4 0 0. Standard IEC 61690-2:2000, International Electrotechnical Commission, Manchester, UK.

Kaminsky, W. M., Lloyd, S. and Orlando, T. P. 2004. Scalable superconducting architecture for adiabatic quantum computation. *arXiv:quant-ph/0403090v2*.

Kirkpatrick, S., Gelatt, C. D. and Vecchi, M. P. 1983. Optimization by simulated annealing. *Science 220,* 4598, 671–680.

Knight, W. 2017. IBM Raises the Bar with a 50-Qubit Quantum Computer. *MIT Technology Review*. 10 November 2017. ISSN: 0040-1692. URL: `https://www.technologyreview.com/s/609451/ibm-raises-the-bar-with-a-50-qubit-quantum-computer` [Accessed on April 24, 2018].

Lucas, A. 2014. Ising formulations of many NP problems. *Frontiers in Physics 2*, 5, 1–15.

Microsoft Corp. 2017. The Q# progamming language. URL: `https://docs.microsoft.com/en-us/quantum/quantum-qr-intro` [Accessed on April 24, 2018].

Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J. and Tack, G. 2007. MiniZinc: Towards a standard CP modelling language. In *Proc. of the 13th International Conference on Principles and Practice of Constraint Programming*, C. Bessière, Ed. Springer, Berlin, Heidelberg, 529–543.

Pakin, S. 2016. A quantum macro assembler. In *Proc. of the 2016 IEEE High Performance Extreme Computing Conference (HPEC)*.

Robinson, J. A. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM 12,* 1, 23–41.

Shor, P. W. 1999. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review 41,* 2, 303–332.

Smith, R. S., Curtis, M. J. and Zeng, W. J. 2017. A practical quantum instruction set architecture. *arXiv:1608.03355 [quant-ph]*.

Thomas, D. and Moorby, P. 2002. *The Verilog Hardware Description Language*, 5th ed. Springer.

Triska, M. 2012. The finite domain constraint solver of SWI-Prolog. In *Proc. of the 11th International Symposium on Functional and Logic Programming (FLOPS 2012)*. Lecture Notes in Computer Science, vol. 7294. Kobe, Japan, 307–316.

Wecker, D. and Svore, K. M. 2014. LIQ$Ui|\rangle$: A software design architecture and domain-specific language for quantum computing. *arXiv:1402.4467 [quant-ph]*.

Wielemaker, J., Schrijvers, T., Triska, M. and Lager, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming 12,* 1–2, 67–96.

Wolf, C. and Glaser, J. 2013. Yosys—A free Verilog synthesis suite. In *Proc. of the 21st Austrian Workshop on Microelectronics (Austrochip 2013)*. Linz, Austria.

Yamaoka, M., Yoshimura, C., Hayashi, M., Okuyama, T., Aoki, H. and Mizuno, H. 2016. A 20k-spin Ising chip to solve combinatorial optimization problems with CMOS annealing. *IEEE Journal of Solid-State Circuits 51,* 1, 303–309.

---

bigger(X, Y) :−
    X > Y,
    X > 0,
    X < 10,
    Y > 6,
    Y < 15.

---

Fig. 10. Sample Prolog source file.

---

```
module \bigger/2 (A, B, Valid);
   input [3:0] A;
   input [3:0] B;
   output Valid;
   wire [4:0] $v1;
   assign $v1[0] = A > B;
   assign $v1[1] = A > 4'd0;
   assign $v1[2] = A < 4'd10;
   assign $v1[3] = B > 4'd6;
   assign $v1[4] = B < 4'd15;
   assign Valid = &$v1;
endmodule

module Query (Big, Little, Valid);
   input [3:0] Big;
   input [3:0] Little;
   output Valid;
   wire $v1;
   \bigger/2 \bigger_xvLbZ/2 (Big, Little, $v1);
   assign Valid = &$v1;
endmodule
```

Fig. 11. Verilog code generated from Figure 10 by QA Prolog.

Appendix A. An end-to-end example

In this section, we detail the complete QA Prolog compilation and execution process illustrated in Figure 5. Figure 10 presents a Prolog source file that defines a `bigger/2` predicate, which we query with "`bigger(Big, Little).`" Because the constraint logic works on free variables, this example does not work with ordinary Prolog environments. For instance, SWI-Prolog (Wielemaker *et al.* 2012) returns an "`Arguments are not sufficiently instantiated`" error.

QA Prolog compiles `bigger/2` and the associated query into the Verilog code shown in Figure 11. Because the largest integer literal appearing in the Prolog source code is 15, the Verilog code uses 4-bit integers throughout.
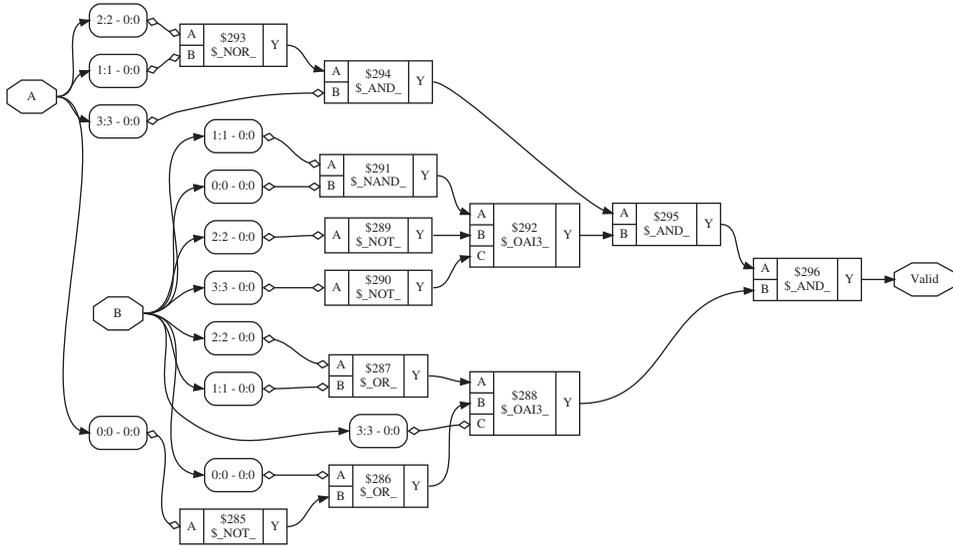
Fig. 12. Visualization of the optimized netlist Yosys produced from Figure 11.

The EDIF netlist that Yosys produces from Figure 11 is a rather verbose s-expression. Rather than present it here, Figure 12 shows a visualization of the circuit that that Yosys renders using Graphviz (Gansner and North 2000). The notation that Yosys uses indicates how bits are renumbered as they flow from one component to the next. "OAI3" represents a 3-input OR-AND-invert gate: $Y = \neg((A \vee B) \wedge C)$.

edif2qmasm translates the circuit illustrated in Figure 12 to the QMASM code listed in Figure 13. In QMASM syntax, "=" specifies a *chain* (a strongly negative $J_{i,j}$) and "<->" specifies an *alias* (multiple names for the same spin). The translation from EDIF constructs to QMASM constructs is nearly one-to-one, which makes the process fairly straightforward.

Finally, QMASM assigns each variable to one or more physical spins that are available on the target hardware. Because the process contains some stochastic elements, Figure 14 presents only one instance of a mapping to the hardware. In the case shown, the Hamiltonian uses 76 h_i and 101 $J_{i,j}$. The Big variable in the Prolog query is mapped to the bit string $\sigma_{198}\,\sigma_{13}\,\sigma_{193}\,\sigma_{212}$, and the Little variable in the query is mapped to the bit string $\sigma_9\,\sigma_{197}\,\sigma_8\,\sigma_{215}$ (both big-endian). The Valid bit (Figure 11) is mapped to σ_{109}.

We ran the Figure 14 Hamiltonian on Los Alamos National Laboratory's D-Wave 2X system, specifying that we wanted 1,000 samples of the $\boldsymbol{\sigma}$ vector and an annealing time of 20 μs. The D-Wave 2X found exactly the three valid solutions: "Big = 8, Little = 7" (139 instances), "Big = 9, Little = 7" (137 instances) and "Big = 9, Little = 8" (226 instances). QMASM automatically rejected the remaining instances. (It is not uncommon for a quantum annealer to return an invalid solution, which is analogous to a classical optimization algorithm getting stuck in a local minimum.) Although only 20 ms were spent performing quantum

```
!include <stdcell>

# bigger/2
!begin_macro id00007
  !use_macro NOT $id00008
  !use_macro OR $id00009
  !use_macro OR $id00010
  !use_macro OAI3 $id00011
  !use_macro NOT $id00012
  !use_macro NOT $id00013
  !use_macro NAND $id00014
  !use_macro OAI3 $id00015
  !use_macro NOR $id00016
  !use_macro AND $id00017
  !use_macro AND $id00018
  !use_macro AND $id00019
  $id00009.B = $id00008.Y
  $id00011.A = $id00010.Y
  $id00019.B = $id00011.Y
  $id00011.B = $id00009.Y
  Valid = $id00019.Y
  B[0] = $id00009.A
  B[0] = $id00014.B
  $id00009.A = $id00014.B
  B[0] <-> $id00009.A
  B[1] = $id00014.A
  B[1] = $id00010.B
  $id00014.A = $id00010.B
  B[1] <-> $id00014.A
  B[2] = $id00010.A
  B[2] = $id00012.A
```

```
  $id00010.A = $id00012.A
  B[2] <-> $id00010.A
  B[3] = $id00013.A
  B[3] = $id00011.C
  $id00013.A = $id00011.C
  B[3] <-> $id00013.A
  A[1] = $id00016.B
  A[0] = $id00008.A
  A[3] = $id00017.B
  A[2] = $id00016.A
  $id00017.A = $id00016.Y
  $id00015.C = $id00013.Y
  $id00018.B = $id00015.Y
  $id00019.A = $id00018.Y
  $id00018.A = $id00017.Y
  $id00015.B = $id00012.Y
  $id00015.A = $id00014.Y
!end_macro id00007

!begin_macro Query
  !use_macro id00007 $id00039
  $id00039.Valid = Valid
  $id00039.B[1] = Little[1]
  $id00039.B[0] = Little[0]
  $id00039.B[3] = Little[3]
  $id00039.B[2] = Little[2]
  $id00039.A[2] = Big[2]
  $id00039.A[3] = Big[3]
  $id00039.A[0] = Big[0]
  $id00039.A[1] = Big[1]
!end_macro Query

!use_macro Query Query
```

Fig. 13. QMASM code generated from the EDIF version of Figure 12 by edif2qmasm.

$$\mathcal{H}_{\text{Query}}(\boldsymbol{\sigma}) = \tfrac{5}{80}\sigma_0 + \tfrac{1}{12}\sigma_1 + \tfrac{1}{8}\sigma_3 + \tfrac{1}{8}\sigma_5 + \tfrac{5}{80}\sigma_6 + \tfrac{5}{80}\sigma_{10} - \tfrac{1}{2}\sigma_{14} - \tfrac{1}{8}\sigma_{15} - \tfrac{1}{4}\sigma_{16} - \tfrac{1}{8}\sigma_{17} - \tfrac{1}{8}\sigma_{19} -$$
$$\tfrac{1}{8}\sigma_{20} - \tfrac{1}{8}\sigma_{22} - \tfrac{1}{8}\sigma_{23} + \tfrac{5}{80}\sigma_{96} + \tfrac{1}{12}\sigma_{97} + \tfrac{1}{2}\sigma_{99} + \tfrac{5}{80}\sigma_{100} + \sigma_{101} + \tfrac{1}{12}\sigma_{102} - \tfrac{5}{80}\sigma_{104} + \tfrac{5}{80}\sigma_{106} - \tfrac{5}{80}\sigma_{107} +$$
$$\tfrac{5}{80}\sigma_{108} - \tfrac{5}{80}\sigma_{110} - \tfrac{1}{8}\sigma_{113} - \tfrac{1}{8}\sigma_{115} + \tfrac{1}{2}\sigma_{117} - \tfrac{5}{80}\sigma_{118} - \tfrac{1}{8}\sigma_{119} + \tfrac{5}{80}\sigma_{192} - \tfrac{1}{8}\sigma_{194} - \tfrac{1}{12}\sigma_{195} - \tfrac{1}{2}\sigma_{196} -$$
$$\tfrac{1}{8}\sigma_{200} - \tfrac{1}{2}\sigma_{203} - \tfrac{1}{8}\sigma_{204} + \tfrac{1}{12}\sigma_{205} - \tfrac{1}{2}\sigma_{208} - \tfrac{1}{8}\sigma_{209} + \tfrac{1}{12}\sigma_{210} + \tfrac{1}{12}\sigma_{213} + \tfrac{1}{4}\sigma_{214} - \tfrac{1}{8}\sigma_{290} - \tfrac{1}{12}\sigma_{291} +$$
$$\tfrac{1}{2}\sigma_{294} - \tfrac{1}{4}\sigma_{296} - \tfrac{3}{8}\sigma_{297} - \tfrac{5}{80}\sigma_{299} - \tfrac{5}{80}\sigma_{300} - \tfrac{1}{4}\sigma_{301} - \tfrac{1}{8}\sigma_{302} - \tfrac{1}{4}\sigma_{303} + \tfrac{1}{4}\sigma_{305} + \tfrac{1}{4}\sigma_{309} - \tfrac{1}{8}\sigma_{310} +$$
$$\tfrac{1}{4}\sigma_1\sigma_5 - \sigma_3\sigma_5 - \sigma_0\sigma_6 - \sigma_5\sigma_{13} - \tfrac{1}{2}\sigma_6\sigma_{14} - \tfrac{1}{2}\sigma_{10}\sigma_{14} - \sigma_9\sigma_{15} + \tfrac{1}{8}\sigma_{16}\sigma_{20} + \tfrac{1}{4}\sigma_{17}\sigma_{20} - \sigma_{19}\sigma_{20} -$$
$$\sigma_{14}\sigma_{22} + \tfrac{1}{4}\sigma_{16}\sigma_{22} + \tfrac{3}{8}\sigma_{17}\sigma_{22} + \tfrac{1}{4}\sigma_{19}\sigma_{22} - \sigma_{15}\sigma_{23} + \tfrac{1}{2}\sigma_{16}\sigma_{23} - \sigma_{17}\sigma_{23} + \tfrac{1}{4}\sigma_{19}\sigma_{23} - \sigma_0\sigma_{96} - \sigma_1\sigma_{97} +$$
$$\tfrac{1}{2}\sigma_3\sigma_{99} + \tfrac{1}{4}\sigma_{96}\sigma_{100} - \sigma_{97}\sigma_{102} + \tfrac{1}{2}\sigma_{99}\sigma_{102} - \sigma_8\sigma_{104} + \tfrac{1}{2}\sigma_9\sigma_{105} - \sigma_{10}\sigma_{106} - \sigma_{100}\sigma_{108} - \tfrac{1}{3}\sigma_{104}\sigma_{108} -$$
$$\sigma_{106}\sigma_{108} - \tfrac{1}{3}\sigma_{107}\sigma_{108} + \sigma_{101}\sigma_{109} - \sigma_{104}\sigma_{110} - \tfrac{1}{3}\sigma_{106}\sigma_{110} - \sigma_{107}\sigma_{110} - \tfrac{1}{8}\sigma_{16}\sigma_{112} - \sigma_{19}\sigma_{115} -$$
$$\sigma_{108}\sigma_{116} - \sigma_{114}\sigma_{116} - \sigma_{109}\sigma_{117} - \tfrac{1}{2}\sigma_{113}\sigma_{117} - \tfrac{1}{2}\sigma_{115}\sigma_{117} - \sigma_{110}\sigma_{118} - \sigma_{114}\sigma_{118} + \tfrac{1}{8}\sigma_{112}\sigma_{119} +$$
$$\tfrac{1}{4}\sigma_{113}\sigma_{119} - \sigma_{115}\sigma_{119} - \sigma_{96}\sigma_{192} - \sigma_{97}\sigma_{193} - \sigma_{99}\sigma_{195} + \tfrac{1}{4}\sigma_{194}\sigma_{196} - \sigma_{195}\sigma_{196} - \sigma_{192}\sigma_{197} -$$
$$\sigma_{194}\sigma_{198} + \tfrac{1}{4}\sigma_{104}\sigma_{200} - \sigma_{105}\sigma_{201} + \tfrac{1}{2}\sigma_{107}\sigma_{203} - \sigma_{200}\sigma_{204} - \sigma_{202}\sigma_{204} + \tfrac{1}{2}\sigma_{203}\sigma_{204} - \sigma_{200}\sigma_{205} -$$
$$\sigma_{202}\sigma_{205} - \sigma_{112}\sigma_{208} - \sigma_{113}\sigma_{209} + \tfrac{1}{2}\sigma_{211}\sigma_{212} - \sigma_{205}\sigma_{213} - \tfrac{1}{2}\sigma_{208}\sigma_{213} - \sigma_{210}\sigma_{213} - \tfrac{1}{2}\sigma_{208}\sigma_{214} +$$
$$\tfrac{1}{4}\sigma_{210}\sigma_{214} - \sigma_{211}\sigma_{214} - \sigma_{210}\sigma_{215} - \sigma_{192}\sigma_{288} - \sigma_{194}\sigma_{290} - \sigma_{195}\sigma_{291} + \tfrac{1}{4}\sigma_{288}\sigma_{293} - \sigma_{289}\sigma_{293} -$$
$$\tfrac{1}{2}\sigma_{290}\sigma_{294} - \tfrac{1}{2}\sigma_{291}\sigma_{294} + \tfrac{1}{4}\sigma_{288}\sigma_{295} - \sigma_{289}\sigma_{295} - \sigma_{201}\sigma_{297} - \sigma_{203}\sigma_{299} + \tfrac{1}{8}\sigma_{296}\sigma_{300} + \tfrac{3}{8}\sigma_{297}\sigma_{300} -$$
$$\sigma_{299}\sigma_{300} + \tfrac{1}{8}\sigma_{293}\sigma_{301} - \sigma_{296}\sigma_{301} + \tfrac{1}{2}\sigma_{297}\sigma_{301} + \tfrac{1}{8}\sigma_{299}\sigma_{301} - \sigma_{294}\sigma_{302} + \tfrac{1}{4}\sigma_{296}\sigma_{302} - \tfrac{1}{8}\sigma_{295}\sigma_{303} +$$
$$\tfrac{1}{8}\sigma_{296}\sigma_{303} + \tfrac{1}{2}\sigma_{297}\sigma_{303} + \tfrac{1}{4}\sigma_{299}\sigma_{303} - \sigma_{209}\sigma_{305} - \tfrac{1}{2}\sigma_{301}\sigma_{309} - \sigma_{305}\sigma_{309} - \sigma_{302}\sigma_{310} - \tfrac{1}{2}\sigma_{305}\sigma_{310}$$

Fig. 14. Final Hamiltonian executed on LANL's D-Wave 2X.

annealing, a total of 200 ms of real time was spent on the D-Wave 2X. This includes various requisite pre- and post-processing tasks.

The total end-to-end time, including not only the time spent on the D-Wave 2X but also the time spent in QA Prolog, Yosys, edif2qmasm, QMASM, the local filesystem and various networks between the development workstation and the D-Wave 2X, averages 3.2 ± 1.1 s over 100 trials. Clearly, a problem like the one used in this exercise is insufficiently complex to overcome the numerous overheads and observe an increase in the performance over what can readily be achieved classically.