C to D-Wave: A High-level C Compilation Framework for Quantum Annealers

Mohamed W. Hassan*, Scott Pakin†, and Wu-chun Feng*‡
*Electrical & Computer Engineering, Virginia Tech, Blacksburg, VA, USA
†Computer, Computational, and Statistical Sciences Division, Los Alamos National Laboratory, Los Alamos, NM 87545
‡Computer Science, Virginia Tech, Blacksburg, VA, USA
Email: {mwasfy@vt.edu, pakin@lanl.gov, wfeng@vt.edu}

***Abstract*—A quantum annealer solves optimization problems by exploiting quantum effects. Problems are represented as Hamiltonian functions that define an energy landscape. The quantum-annealing hardware relaxes to a solution corresponding to the ground state of the energy landscape. Expressing arbitrary programming problems in terms of real-valued Hamiltonian-function coefficients is unintuitive and challenging. This paper addresses the difficulty of programming quantum annealers by presenting a compilation framework that compiles a subset of C code to a quantum machine instruction (QMI) to be executed on a quantum annealer. Our work is based on a modular software stack that facilitates programming D-Wave quantum annealers by successively lowering code from C to Verilog to a symbolic "quantum macro assembly language" and finally to a device-specific Hamiltonian function. We demonstrate the capabilities of our software stack on a set of problems written in C and executed on a D-Wave 2000Q quantum annealer.**

***Index Terms*—Compiler, D-Wave, high-level language, mapping, quantum annealing, quantum assembler, macro, quantum computing, QMASM, Verilog, EDIF**

I. Introduction

One can currently classify a quantum computer as either a quantum annealer (QA) or a gate-model (or circuit-model) quantum computer. The latter supports general-purpose quantum computation [1] but with a limited number of qubits.[1] On the other hand, a QA is a special-purpose quantum computer that scales well in terms of qubits but is oriented to a specific type of application, namely *optimization problems*. The largest available QA is D-Wave's with 2048 qubits [3]; D-Wave projects a new platform with more than 5000 qubits, along with a next-generation Pegasus architecture, for mid-2020 [4].

A classical analogue to quantum annealing is simulated annealing [5], a well-established technique for finding an optimal value in a large search space. Annealing in hardware offers a potential gain over classical solutions in the quality of the solution attainable in a given length of time. A QA further increases this potential by exploiting quantum effects, most notably, *quantum tunneling*. Quantum tunneling supports cutting through tall energy barriers to transition from one state to a superior state, leading to a greater probability than simulated annealing of finding the ground state, given the same annealing schedule [6].

Because a QA fundamentally differs from the von Neumann architecture used in classical computing, classical programming techniques are *not* directly applicable to quantum annealing. Programming a QA involves defining an energy landscape in terms of a 2-local Ising-model Hamiltonian function such that the coordinates of the ground state (i.e., minimum energy value) correspond to the solution sought. More precisely, a QA program consists of a list of real numbers that correspond to linear and quadratic coefficients in the Hamiltonian function. Programming at this level is tedious and error-prone. Thus, the question that we seek to answer in this paper is as follows: *Can one compile from a high-level language to an optimization problem accepted by quantum-annealing hardware?*

Our approach centers around the realization of a software stack that abstracts programmability from a very low-level quantum language, consisting of a single instruction called a quantum machine instruction (QMI), to a much higher level of abstraction, such as C. It encompasses a sequence of tools that successively compiles from a more user-friendly abstraction level, namely C, to a more hardware-friendly one, namely a QMI, as outlined in Figure 1.

Specifically, this paper, as encompassed by Figure 1, introduces C-to-D-Wave, a translator from a stylized subset of C [7] to Verilog [8]. We then compile the resulting Verilog code to EDIF [9] using Yosys [10]. Next, we use edifqmasm [11] to compile EDIF to QMASM, a symbolic, hardware-independent representation of a Hamiltonian function. This QMASM code is then compiled to a hardware-dependent 2-local Ising-model Hamiltonian function using the QMASM tool [12]. Finally, D-Wave's solver API (SAPI) libraries [13] generate the corresponding QMI to run on a D-Wave quantum annealer (QA).

The following are the main contributions of this work:

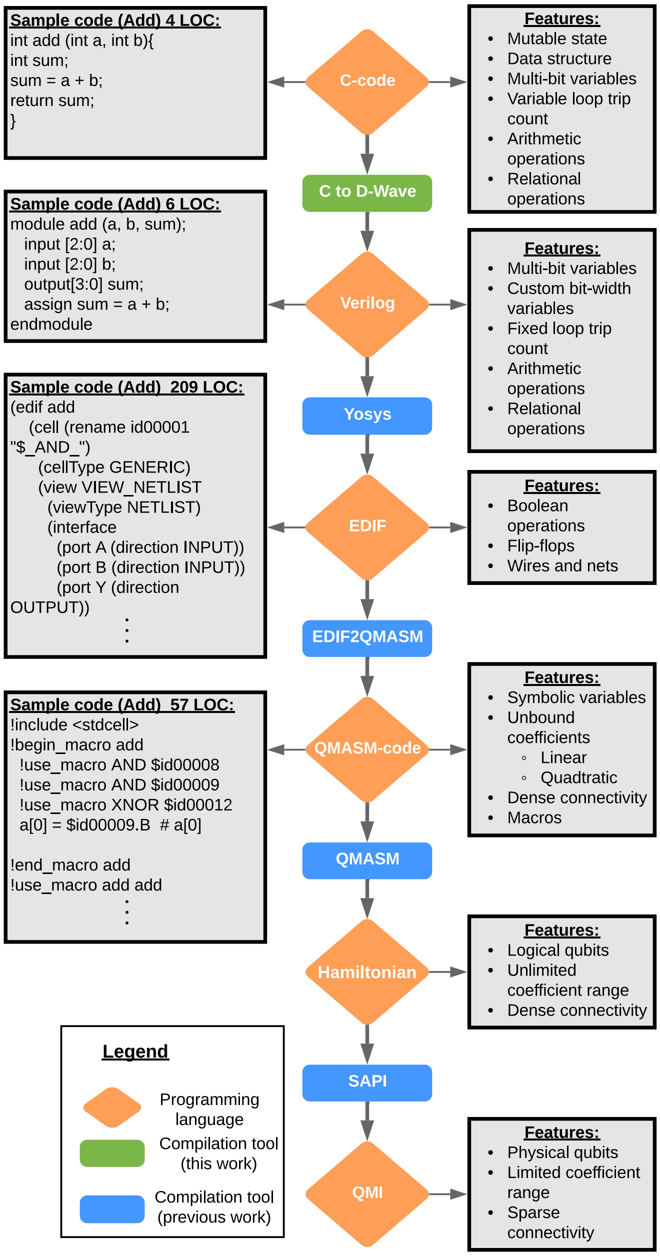
- A novel framework that compiles high-level classical code to a quantum machine instruction for quantum annealers like D-Wave.
- An evaluation of the efficacy of our high-level programming abstraction on the D-Wave 2000Q QA.

To that end, we first present background material on quantum annealing in general and D-Wave systems in particular in Section II. Section III describes our compilation framework

This work was supported by a VT ICTAS seed grant, by LANL LDRD projects 20190065DR and 20160069DR, and by the U.S. Department of Energy through Los Alamos National Laboratory, which is operated by Triad National Security, LLC for the National Nuclear Security Administration of the U.S. Department of Energy (contract no. 89233218CNA000001).

[1] Currently, Google has the largest reported gate-model quantum computer with 72 qubits [2].

978-1-7281-5020-8/19/$31.00 ©2019 IEEE



Sample code (Add) 4 LOC:

```
int add (int a, int b){
int sum;
sum = a + b;
return sum;
}
```

Sample code (Add) 6 LOC:

```
module add (a, b, sum);
   input [2:0] a;
   input [2:0] b;
   output[3:0] sum;
   assign sum = a + b;
endmodule
```

Sample code (Add) 209 LOC:

```
(edif add
    (cell (rename id00001
"$_AND_")
      (cellType GENERIC)
      (view VIEW_NETLIST
        (viewType NETLIST)
        (interface
          (port A (direction INPUT))
          (port B (direction INPUT))
          (port Y (direction
OUTPUT))
                ⋮
```

Sample code (Add) 57 LOC:

```
!include <stdcell>
!begin_macro add
  !use_macro AND $id00008
  !use_macro AND $id00009
  !use_macro XNOR $id00012
  a[0] = $id00009.B  # a[0]

!end_macro add
!use_macro add add
                ⋮
```

Fig. 1: Our C-to-D-Wave quantum annealing software stack

that maps C code to a QMI for execution on a D-Wave QA. Section IV evaluates our framework's productivity and efficiency. Related work, including previous efforts to develop programming models for quantum annealing, is described in Section V. Finally, Section VI draws conclusions from our findings and lays out opportunities for future work.

II. Background

This section presents the basics of quantum annealing and discusses the underlying hardware infrastructure of a D-Wave QA. We explain how to program a QA at the lowest level and articulate the semantic and conceptual gaps that must be overcome to program a QA in a high-level language.

A. *Quantum Annealing Basics*

A QA is a special-purpose device that specializes in finding the set of spins that minimize the energy of an Ising-model Hamiltonian. A D-Wave QA imposes the added restriction of operating on 2-local Ising-model Hamiltonians. "2-local" means that the function can contain up to quadratic terms, which limits interactions to at most two (2) spins. D-Wave's problem Hamiltonian can be expressed as follows:

$$\mathcal{H}(\sigma) = \sum_{i=1}^{N} h_i \sigma_i + \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} J_{ij} \sigma_i \sigma_j \qquad (1)$$

where $\sigma_i \in \{-1, +1\}$, $h_i \in \mathbb{R}$, and $J_{ij} \in \mathbb{R}$. In other words, $\mathcal{H}$ is a quadratic pseudo-Boolean function. Physically, a spin σ_i is implemented with a qubit (i.e., quantum bit); an h_i coefficient represents the strength of the external field applied to σ_i; and a J_{ij} coefficient represents the strength of the interaction between σ_i and σ_j.

A QA program is merely a list of h_i and J_{ij} coefficients for Equation (1). This list is referred to as a *quantum machine instruction* (QMI). Although a QMI is semantically simple, expressing non-trivial programs in terms of Equation (1)'s h_i and J_{ij} coefficients requires substantial effort.

B. *D-Wave System Architecture and Low-Level Interface*

1) Hardware architecture: The fundamental component of a D-Wave system is a flux qubit, implemented with a superconducting loop. A D-Wave quantum processing unit (QPU) is a sparsely connected graph of connected qubits. The topology, called a *Chimera graph* [14], is a 2-D grid of *unit cells*, where each unit cell consists of eight qubits linked with a bipartite connectivity structure, as shown in Figure 2. In addition, each qubit in a unit cell is connected to its peers in two adjacent unit cells (either "north and south" or "east and west" but not both). A Chimera graph is therefore of degree six: each qubit has four (4) internal connections within a unit cell and two connections to neighboring unit cells.

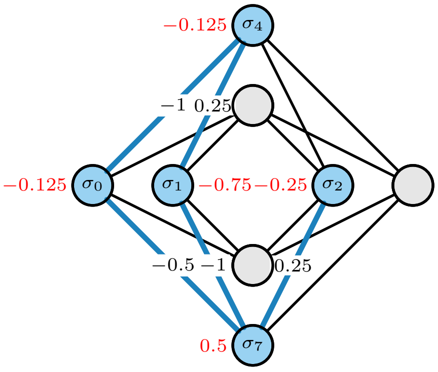


Fig. 2: AND gate embedded on D-Wave's quantum annealer (QA) unit cell

2) Chains and embedding: A Hamiltonian function is logically represented as a graph, with the h_i corresponding to node weights and the J_{ij} corresponding to edge weights. A

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

problem's logical representation does not necessarily follow a Chimera graph's structure. Therefore, one must map the logical representation of a problem onto the hardware's physical topology in a process known as *minor embedding* [14], in which high-degree logical qubits are implemented in terms of multiple degree-6 physical qubits. The efficiency of this mapping process is critical because it determines the number of utilized physical qubits. An example of an AND gate embedded on the physical hardware is shown in Figure 2. Using fewer qubits is preferred in order to fit larger problems on the limited hardware resources.

3) Solver API (SAPI): SAPI is the lowest-level software interface to D-Wave's QPU [15]. SAPI includes multiple features needed to interact with the QPU such as querying and connecting to a solver (an actual D-Wave or a local simulator), minor-embedding a logical problem onto the physical hardware, constructing a QMI from a list of Hamiltonian function coefficients, submitting QMIs to the solver, and post-processing the solutions returned.

C. D-Wave Compatible High-Level Programming Style

Here we outline the conceptual differences between programming a conventional processor and programming a QA.

- *A QA has no notion of a clock.* QA code cannot include sequential dependencies. This can be faked by replicating the entire program per discrete time step—entire program at time 0, entire program at time 1, and so forth—and linking each time step to its successor [11]. However, this trade-off of time for space is often impractical in terms of the required qubit count.
- *A QA has no notion of registers.* Mutable state is not supported by a QA.
- *A QA has no notion of explicit inputs.* All a QA does is minimize a Hamiltonian function. Inputs must be encoded as function coefficients (h_i and J_{ij}).

In Section III, we illustrate how to compile C code in a way that satisfies the above constraints and runs on a D-Wave QA.

III. C-to-D-Wave

This section discusses the C code compilation framework that transforms a subset of C code to a QMI. We chose C as the programming language because it is low level enough to provide precise control over the hardware yet high level enough to be accessible to a wide range of developers. We leverage the large engineering effort spent developing a software stack [10]–[13] that abstracts quantum annealing programming to the level of Verilog, a hardware description language. Figure 1 illustrates the levels of the software stack, including both programming languages and tools. It also articulates the basic features of each abstraction level in this toolchain, which highlights the challenges in compiling high-level code to lower-level code—generally, lower levels offer a more restricted feature set. Finally, Figure 1 presents code samples at all but the bottom two levels, where code size is prohibitively large.

The work presented in this paper extends the previously implemented toolchain (blue boxes) from Verilog to a C compilation environment (green box). However, the challenge is that the C code and the resulting Verilog code must be compatible with the underlying toolchain and in turn the limitations of QA hardware (discussed in Section II-C).

We used Clang [16] release 6.0 for constructing an abstract syntax tree (AST) from a C program. Then we used Clang's Libtool to create a standalone tool that uses the rewriter class along with Clang's AST recursive visitor for the conversion from C to Verilog. This approach provides us the flexibility we need to generate Verilog code compatible with the coding style of D-Wave's software stack. The tool is open-source and available at https://github.com/lanl/c2dwave.

A. Verilog to a QMI

The previously existing portion of the toolchain generates D-Wave QMIs from Verilog code through a sequence of translation and compilation steps. The first step of the compilation process compiles a hardware specification expressed in Verilog [8] to a digital circuit in EDIF (Electronic Design Interchange Format) format [17]. EDIF is a vendor-neutral format for expressing the netlist (description of the connectivity of an electronic circuit) of a synthesized design. For this step we use the Yosys hardware synthesizer [10] to synthesize Verilog to hardware and the open-source ABC circuit-optimization tool [18] to optimize the synthesized circuit.

The second step uses edif2qmasm [11] to compile EDIF code to a logical Hamiltonian function expressed in QMASM format [12].

The third step uses the QMASM tool [12], a quantum macro assembler that allows qubits to be referenced symbolically (string variable names) as opposed to numerically (physical qubit number). QMASM codes can be run immediately by the tool on any D-Wave solver, either real hardware or a simulator, using SAPI [13] to map them to a QMI. QMASM can also generate input for D-Wave's qbsolv [19] tool, which can split a large problem into hardware-sized QMIs, execute these, and combine the results into a solution to the full problem. Regardless of how QMASM is run, execution results are automatically expressed in terms of their program-specified symbolic names, which greatly facilitates program development and debugging.

B. C-to-D-Wave Supported Features

Verilog is a hardware-centric language that is not readily accessible to the mainstream software community. This shortcoming is overcome by the work presented in this paper (C to D-Wave), which abstracts the programming environment to the level of C code.

C-to-D-Wave supports a feature set that has enabled the successful generation of Verilog modules for multiple problems, on which we will elaborate in Section IV. Supported constructs include arithmetic operations (Add, Sub, Mul, Div, Mod), logic operations (NOT, AND, OR, XOR), branching operations (**for** loops, **while** loops, **if** conditions, and C's ternary operator,

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

"? :"), integer and Boolean variables, array declarations, and **return** statements. These constructs can be used as building blocks to implement more complex problems.

However, some of these constructs have limited support. Integer variables default to being only five bits wide, and branch conditions (such as the **if** statement condition and the **for** loop exit condition) are assumed to be constant to avoid synthesizing a sequential circuit.

C. Generating Verilog Code

Here we describe the basic operation of C-to-D-Wave that maps a form of C code to a form of Verilog that is compatible with the programming style discussed earlier. A function's inputs and outputs are converted to their Verilog equivalent in the generated Verilog module, while the function body is transformed into a logic circuit to be implemented in Verilog. We begin by describing C-to-D-Wave's conversion of a C function signature to the signature of a Verilog module:

- Function parameters are considered inputs.
- The **return** statement defines the output.
- Integers are converted to 5-bit variables (arbitrary; can be changed) in Verilog.
- **bool** variables are converted to 1-bit variables in Verilog.
- The **register** keyword is used before the variable declaration to force the generation of a register in Verilog.

Registers are required in Verilog in two cases: when a variable is going to be used as an induction variable of a loop, and when a variable is going to be reassigned mid-code, as in "temp = temp + val;". In this case, the variable "temp" should be a register.

Now we shift our focus to the C code function body, which we convert to a logic circuit expressed in Verilog. In Verilog, combinatorial **assign** statements assign a value or the result of an operation to a given variable. Loops in Verilog require a **genvar** directive, which is linked to the loop's induction variable. The last aspect we consider is "**always** @ (clk)" blocks, which are commonly used to specify actions to take on a clock signal. This implies sequential logic controlled by the specified clock signal in the sensitivity list targeting the logic inside that block.

Because a QA has no clock (Section II-C), we convert all code to a combinational circuit. Consequently, we encapsulate the entire function body in an "**always** @*" block, where using "*" allows the hardware synthesizer to infer combinational logic. Inside an "**always** @*" block, we do not need to generate **assign** statements. Also, loops inside "**always** @*" blocks do not need a **genvar** directive. Instead, the induction variable can be a register. In this case, the use of registers does not necessarily imply that the circuit becomes sequential because the loop bounds are assumed to be static, and the loop will be completely unrolled to be synthesized as a combinational circuit.

IV. Testing and evaluation

In this section, we explore what C-to-D-Wave is capable of expressing. We step in detail through the compilation of a maximum-cut problem from C code to Verilog. We have compiled more complex codes but omit the details of their compilation from this paper due to space purposes. However, we do present graphs of qubit utilization for the logical and physical representations of these problems. We also quantify the productivity of our software stack in terms of SLOC (source lines of code). C-to-D-Wave successfully generates D-Wave-compatible Verilog code for multiple problems including the traveling salesman problem (TSP), subset sum, map coloring, max-cut, sorting, and multiplication/factoring.

Listing 1: C-code example of max-cut problem

```
1  bool max_cut (bool a,bool b,bool c,bool d,bool e) {
2    register bool valid = 0; // output
3    int cut = 4; // variable to set the number of cuts
4    register int temp = 0; // calculate number of cuts
5    register int x; // induction variable
6    register int y; // induction variable
7    bool in_val[5]; // array to hold input
8    bool arr[5][5]; // 2−D array for graph connectivity
9    for (x = 0 ; x < 5 ; x = x + 1)
10       for (y = 0 ; y < 5 ; y = y + 1)
11           if((arr[x][y]==1)&&(in_val[x]!=in_val[y]))
12               temp=temp+1;
13   if(temp>=2∗cut)
14       valid = 1;
15   else
16       valid = 0;
17   return valid; }
```

A. Example of Lowering C Code to Verilog

Here we provide an example of a max-cut problem expressed in C code and the generated Verilog code. Code listing 1 shows a shrunk-down portion of the C code used to represent a max-cut problem (finding a subset S of a graph's nodes that maximizes the number of cut edges between S and the complementary subset). We explain the approach of constructing the max-cut problem as a validation program for specific inputs.

The problem constructs a graph of five nodes, where the graph connectivity structure is defined in the 2-D matrix "arr" (line 8 of Listing 1). However, defining the graph connectivity has been elided from Listing 1 for space purposes. The function inputs five **bool** (Boolean) variables that are set to either 0 or 1, which represent the two sets of the cut graph. The nested loop that starts on line 9 is responsible for iterating on the connectivity matrix of the graph. This nested loop performs a check if two nodes are connected (according to the connectivity matrix) and are located in different sets (according to the input). If so, the "temp" variable is incremented for a valid cut. Then, on line 13, "temp" is compared to the number of cuts set to a value of 4 to decide whether the max-cut constraint is satisfied for a valid solution. The output "valid" is of type **bool**, which if true means that the input did partition the graph into two sets with a number of cuts more than that specified in the code.

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

Listing 2: Verilog example of max-cut problem

```
1   module max_cut (input a,input b,input c,input d,
        input e,output result);
2   wire arr [4:0][4:0];
3   wire in_val [4:0];
4   reg [4:0] y;
5   reg [4:0] x;
6   reg [4:0] temp;
7   wire [4:0] cut;
8   reg valid;
9   always@*
10  begin
11  for (x = 0 ; x < 5 ; x = x + 1)
12  begin
13      for (y = 0 ; y < 5 ; y = y + 1)
14      begin
15          if((arr[x][y]==1)&&(in_val[x]!=in_val[y]))
16          begin
17              temp=temp+1;
18          end
19      end
20  end
21  if(temp>=2*cut)
22  begin
23      valid = 1;
24  end
25  else
26  begin
27      valid = 0;
28  end
29  result = valid;
30  end
31  endmodule
```

The generated Verilog code is illustrated in Listing 2. The return type of the C-function being **bool** specifies that the output of the Verilog module "result" is a one-bit variable. The Boolean arguments of the C code "a", "b", "c", "d", and "e" are translated to one-bit inputs to the Verilog module. **int** variables are translated to 5-bit wires, while **register int** variables are translated to 5-bit **reg** variables in Verilog. The **return** statement on line 17 of the C code specifies "valid" as the internal variable to set the output, which is generated in the Listing 2 Verilog code on line 29.

B. *Quantifying Productivity*

We use a count of source lines of code (SLOC) as a metric for quantifying the productivity of our software stack. We are particularly interested in C-to-D-Wave's SLOC counts relative to those of the lower-level languages in the stack. As shown in Figure 3, the two higher-level programming languages (C and Verilog) require fewer SLOC, which is to be expected. In fact, the C and Verilog versions have similar SLOC, with Verilog slightly higher in all cases. SLOC is substantially higher at the QMASM level (and higher still at the EDIF level, but EDIF is not intended to be written explicitly by programmers). For the logical-Hamiltonian abstraction level, we consider the SLOC to be the number of coefficients of qubit weights and coupler strengths. SLOC is higher here than in any of the other implementations.

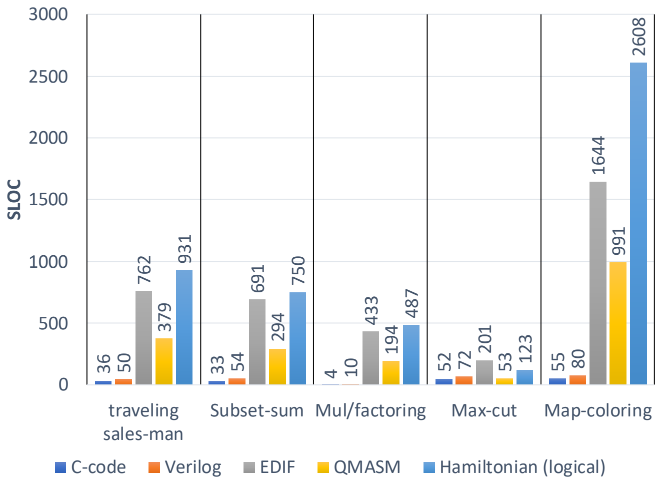


Fig. 3: SLOC count for the tested application set across the abstraction levels of the software stack

The structure of a problem has a large effect on the utilization of scarce hardware resources (qubits). The max-cut problem embeds well onto the D-Wave's Chimera-graph topology, while map-coloring requires a more qubit-hungry minor embedding. Figure 3 shows that in spite of both applications having almost the same C code SLOC count (max-cut: 52, map-coloring: 55), the logical-Hamiltonian SLOC count is different by an order of magnitude (max-cut: 123, map-coloring: 2608). As a matter of fact, the max-cut problem shows the most efficient hardware embedding out of the tested application set. Although the efficiency of this application set may be improved by hand-crafting and manually optimizing Hamiltonian functions, it is infeasible to attempt doing so by hand.

C. *Qubit Utilization Report*

Although QAs have a large number of qubits compared to gate model quantum computers, they are still a scarce resource. Hence, the number of logical and physical qubits for a problem is a very important metric in this context. We show in Figure 4 the reported qubit utilization for the set of problems implemented by C-to-D-Wave. The number of logical qubits is constant for a given problem from compilation to compilation. However, the number of physical qubits changes from one compilation to the other because it depends on a stochastic minor-embedding algorithm [20]. We report the average of 25 runs for each problem, with 100,000 samples and 1μs anneal time per run.

The logical-to-physical qubit mapping depicted in Figure 4 shows nonlinear growth of the physical qubit count as the logical qubit count increases. The max-cut problem shows the lowest qubit utilization which proves that this problem

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

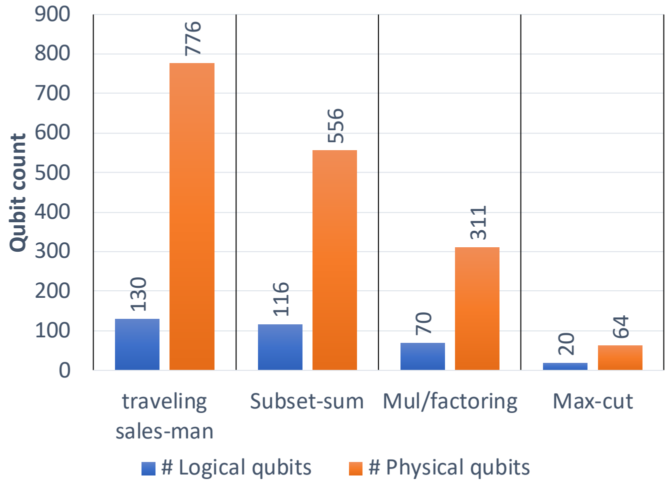


Fig. 4: Logical vs. physical qubit count for the tested application set

is well suited for the target architecture. The 20 logical qubits used to represent this problem are expanded to 64 physical qubits, which is a 3.2x expansion. On the other hand, the traveling salesman problem is represented using 130 logical qubits which expanded to 776 physical qubits—an almost 6x expansion. The map-coloring problem's qubit utilization was not reported, as the design failed to embed on the D-Wave 2000Q. This emphasizes the importance of efficient logical representation of the problem in addition to efficient embedding heuristic.

V. Related work

There are multiple tools available to convert high-level codes such as C and OpenCL to Verilog, including commercial high-level synthesis tools (e.g., Vivado HLS [21], Altera Offline Compiler (aoc) [22]) and open-source tools that use LLVM's intermediate representation [16] to do the conversion (e.g., LegUp [23]). However, the above tools target field-programmable gate array (FPGA) compilation, which imposes two major problems when targeting a QA:

- Converting the code to a finite state machine (FSM), while appropriate for FPGA, is *not* compatible with quantum combinational-style circuits.
- Inserting FPGA board-specific code for the memory interface, registers, and clock control, while required for FPGA, is unnecessary overhead for QA.

Hence, programming a QA using the software stack presented in Figure 1 required a Verilog generation tool that is flexible enough to comply with the restriction imposed by the underlying hardware (Section II-C) and avoids generating the FPGA-specific overhead of HLS tools.

A. General-Purpose Quantum Annealing Programming

QA Prolog [24] converts constraint logic programs expressed in Prolog into a QMI for D-Wave systems. QA Prolog implements fully parallel, constraint-based logic programming on the QA. It leverages the lower portion of the software stack discussed in this paper. Because Prolog is less commonly used than C, this paper presents our C-to-D-Wave compilation framework.

The D-Wave Ocean software suite [25] incorporates higher-level programming abstractions but not expressed as standalone general-purpose programming languages. An example is D-Wave NetworkX, which is a Python library based on the NetworkX graph library [26]. This library focuses on solving NP-complete and NP-hard graph problems on a D-Wave system. D-Wave NetworkX programming model simply transforms a constraint satisfaction problem to a Hamiltonian objective function.

Most other programming tools targeting the D-Wave QA, such as qbsolv [19], are lower level than discussed in this work. Qbsolv takes as input a logical QUBO (a 2-local Ising-model Hamiltonian with $\{0, 1\}$ variables instead of $\{-1, +1\}$ variables), which could be naturally dense and not conformant with the Chimera graph structure. It maps the QUBO to the physical hardware of the D-Wave QA.

B. Domain-Specific Quantum Annealing Programming

Other work targeting QAs generally focuses on domain-specific conversion of an algorithm (or a class of algorithms) to an Ising-model Hamiltonian, such as set-partitioning [27], max-cut [28], clique partitioning [29], and the vertex-coloring problem [30]. In contrast, our work is a general-purpose programming paradigm that subsumes these applications (and others) in one programming framework.

VI. Conclusion

Programming a quantum annealer (QA) is fundamentally different and significantly more difficult from programming a classical computer. Therefore, to better facilitate the productive programming of a QA, we present a compilation framework that compiles a subset of C code to a quantum machine instruction (QMI) to be executed on a QA. To demonstrate the efficacy of our framework, we compile a diverse set of problems, ranging from arithmetic problems (such as multiplication and factoring) to optimization problems (such as traveling salesman and max-cut problems).

We quantify the productivity of our framework by reporting source lines of code (SLOC) as a metric. Based on this metric, our C to D-Wave compilation framework significantly reduces the burden of programming and compiling programs for a D-Wave QA. Specifically, our framework reduces the SLOC of the selected problems by *three orders of magnitude* when comparing the high-level C code to the low-level Hamiltonian used to generate the QMI.

With the next-generation D-Wave system slated to increase the number of qubits by 2.5× and the connectivity by 2.5×, our compilation framework provides a general abstraction that allows quantum developers to seamlessly compile and run their C code across both current and future QAs. Furthermore, the modular design of the software stack makes it easy to add support for new generations of QA hardware.

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

References

[1] E. G. Rieffel and W. H. Polak, *Quantum computing: A gentle introduction*. MIT Press, 2011.

[2] E. Conover. (2018). Google moves toward quantum supremacy with 72-qubit computer, [Online]. Available: https://www.sciencenews.org/article/google-moves-toward-quantum-supremacy-72-qubit-computer.

[3] E. Gibney, "D-Wave upgrade: How scientists are using the world's most controversial quantum computer," *Nature*, pages 447–448, January 2017. DOI: 10.1038/541447b.

[4] J. Russell, "D-Wave previews next-gen platform; debuts Pegasus topology; targets 5000 qubits," *HPCwire*, February 27, 2019. [Online]. Available: https://www.hpcwire.com/2019/02/27/d-wave-previews-next-gen-platform-debuts-pegasus-topology-targets-5000-qubits/.

[5] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, volume 220, number 4598, pages 671–680, 1983.

[6] T. Kadowaki and H. Nishimori, "Quantum annealing in the transverse Ising model," *Physical Review E*, volume 58, pages 5355–5363, 5 November 1998, ISSN: 1063-651X. DOI: 10.1103/PhysRevE.58.5355.

[7] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd edition. Prentice Hall, April 1, 1988, 262 pages, ISBN: 978-0131103627.

[8] IEEE, "IEEE standard for Verilog hardware description language," Design Automation Standard Committee of the IEEE Computer Society, New York, New York, USA, Standard IEEE Std 1364-2005, April 7, 2006, 590 pages. DOI: 10.1109/IEEESTD.2006.99495.

[9] H. Kahn, R. La Fontaine, and R. Lau, "Electronic design interchange format (EDIF), Part 2: Version 4 0 0," International Electrotechnical Commission, Manchester, United Kingdom, International Standard IEC 61690-2:2000, January 31, 2000.

[10] C. Wolf and J. Glaser, "Yosys—a free Verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics*, (Linz, Austria, October 10, 2013). [Online]. Available: http://www.clifford.at/yosys/files/yosys-austrochip2013.pdf (visited on 05/23/2019).

[11] S. Pakin, "Targeting classical code to a quantum annealer," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, (Providence, RI, USA), series ASPLOS '19, New York, NY, USA: ACM, 2019, pages 529–543, ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304071. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304071.

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

[13] "Developer guide for Python," D-Wave Systems, Inc., Burnaby, British Columbia, Canada, User Manual 09-1024A-K, Jul. 30, 2018, 69 pages.

[14] D-Wave Systems, Inc. (2019). Getting started with the d-wave system, [Online]. Available: https://docs.dwavesys.com/docs/latest/doc_getting_started.html (visited on 05/24/2019).

[15] ——, (2018). Solver API REST Web services guide, [Online]. Available: https://docs.dwavesys.com/docs/latest/doc_rest_api.html (visited on 05/23/2019).

[16] C. Lattner, "LLVM and Clang: Advancing compiler technology," in *Free and Open Source Developers' European Meeting*, (Brussels, Belgium), February 5–6, 2011. [Online]. Available: http://llvm.org/pubs/2011-02-FOSDEM-LLVMAndClang.pdf.

[17] H. J. Kahn and R. F. Goldman, "The electronic design interchange format EDIF: Present and future," in *[1992] Proceedings 29th ACM/IEEE Design Automation Conference*, IEEE, 1992, pages 666–671.

[18] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *International Conference on Computer Aided Verification*, Springer, 2010, pages 24–40.

[19] M. Booth, S. P. Reinhardt, and A. Roy, "Partitioning optimization problems for hybrid classical/quantum execution," Tech. Rep. 14-1006A-A, 2017. [Online]. Available: https://github.com/dwavesystems/qbsolv/blob/master/qbsolv_techReport.pdf.

[20] J. Cai, W. G. Macready, and A. Roy, "A practical heuristic for finding graph minors," *arXiv preprint arXiv:1406.2741*, 2014.

[21] Xilinx. (2014). Vivado high-level synthesis, [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf.

[22] Altera. (2015). Altera SDK for OpenCL: Best practices guide, [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_optimization_guide.pdf.

[23] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, ACM, 2011, pages 33–36.

[24] S. Pakin, "Performing fully parallel constraint logic programming on a quantum annealer," *Theory and Practice of Logic Programming*, volume 18, number 5-6, 928–949, 2018. DOI: 10.1017/S1471068418000066.

[25] D-Wave Systems, Inc. (2018). D-wave ocean software documentation, [Online]. Available: https://docs.ocean.dwavesys.com/en/latest/index.html.

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

[26] A. A. Hagberg, D. A. Schult, and P. J. Swart, “Exploring network structure, dynamics, and function using NetworkX,” in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pages 11–15.

[27] M. Lewis, G. Kochenberger, and B. Alidaee, “A new modeling and solution approach for the set-partitioning problem,” *Computers & OR*, volume 35, pages 807–813, March 2008. DOI: 10.1016/j.cor.2006.04.002.

[28] G. A. Kochenberger, J.-K. Hao, Z. Lü, H. Wang, and F. Glover, “Solving large scale max cut problems via tabu search,” *Journal of Heuristics*, volume 19, number 4, pages 565–571, August 2013, ISSN: 1572-9397. DOI: 10.1007/s10732-011-9189-8. [Online]. Available: https://doi.org/10.1007/s10732-011-9189-8.

[29] G. Kochenberger, F. Glover, B. Alidaee, and H. Wang, “Clustering of microarray data via clique partitioning,” *Journal of Combinatorial Optimization*, volume 10, number 1, pages 77–92, August 2005, ISSN: 1573-2886. DOI: 10.1007/s10878-005-1861-1. [Online]. Available: https://doi.org/10.1007/s10878-005-1861-1.

[30] G. A. Kochenberger, F. Glover, B. Alidaee, and C. Rego, “An unconstrained quadratic binary programming approach to the vertex coloring problem,” *Annals of Operations Research*, volume 139, number 1, pages 229–241, October 2005, ISSN: 1572-9338. DOI: 10.1007/s10479-005-3449-7. [Online]. Available: https://doi.org/10.1007/s10479-005-3449-7.

978-1-7281-5020-8/19/$31.00 ©2019 IEEE