Modular Architecture for Classical Simulation of Quantum Circuits

Aniket S. Dalvi*†, Filip Mazurek*, Leon Riesebos*, Jacob Whitlow*,
Swarnadeep Majumder* and Kenneth R. Brown*
*Department of Electrical and Computer Engineering, Duke University, NC 27708, USA
†Email: aniketsudeep.dalvi@duke.edu

***Abstract*—Duke ARTIQ Extensions (DAX) provides a framework for writing modular control software for ion-trap quantum systems. DAX allows users to interface with the system at the gate level using an intermediate representation called DAX.program. However, DAX does not have the tools needed to simulate these programs to see quantum state evolutions, as DAX only allows for simulations at the device level. We propose a modular simulation framework, DAX.program-sim (DPS), that can simulate quantum programs at the level of quantum operations. This addition to DAX for classical simulation of quantum systems is vital for testing, benchmarking, and verification of quantum hardware. The DPS pipeline is designed to have input identical to the one that runs on the hardware. Our architecture allows flexible backends for simulation, enabling both gate-level simulations and pulse-level simulations with and without noise. We demonstrate this unified workflow, executing the same DAX.program file in simulation as well as on hardware. As a specific example, we run benchmarking circuits using simulators targeting ion trap quantum computers and compare them to hardware results of the same circuits.**

***Index Terms*—classical simulation, modular software, quantum circuits, software testing, benchmarking**

Poster Relevance

The work described by this poster primarily falls under the conference topic of quantum software engineering. We describe a modular software architecture used to build a framework for simulating quantum circuits. The project demonstrates a workflow of running a program on a physical system and then using its noise model to run a noisy simulation of the same program on the framework. This would provide a pipeline that runs in parallel to the compilation stack of the physical system and can be utilized to test, validate, and verify experiments and calibration routines before they are run on the hardware.

I. Introduction

Real-time control of quantum systems requires nanosecond precision and efficient execution. Advanced Real-Time Infrastructure for Quantum physics (ARTIQ) [1] provides such a real-time control system. However, ARTIQ requires the users to interface directly at the device driver level. As experimental setups get more elaborate, the portability of control systems is vital. Duke ARTIQ extensions (DAX) [2], [3] builds upon ARTIQ to provide a modular software architecture for the control of experimental setups. It achieves this modularity by adding abstraction layers such as modules (which interface with devices), services (which interface with modules), and experiments (which are written using the underlying modules and services). This modularity allows for portability of code between systems, and ease of use for users as they can now write code at the experiment level. Apart from the experiment level, DAX also allows high-level access to the system through clients, which control the system through standardized interfaces and can run portable experiments. Two such clients that are of relevance here are DAX.program and the pyGSTi [4] client. DAX.program allows users to write explicitly-timed gate-level programs which will be compiled through the layers of DAX to eventually run on the physical system. Similarly, the pyGSTi client allows users to run protocols such as randomized benchmarking (RB) [5], [6] and gate-set tomography (GST) [7] on the physical system. However, DAX does not have tools to simulate these clients at the operation level to see quantum state evolutions, as it can only do device-level simulations. The ability to do these operation level simulations is important for testing and iterating over experiments before running them on the physical system.

Our work, DAX.program-sim (DPS), adds a modular simulation framework to DAX for simulating quantum programs at the level of quantum operations. The modularity of our simulation framework is demonstrated by the ability to have the input as either a DAX.program file or a pyGSTi client with the provision to add additional inputs. The modularity of the framework is further established by allowing multiple simulation backends. Currently, DPS supports noisy statevector and density matrix simulations using Qiskit [8] and Cirq [9], but our architecture is built to interface with any user-provided backend. DPS aims to create a tool to simulate realistic hardware by providing the simulator with noise models extracted from the system.

II. Software Architecture

A. Frontend

The frontend of DPS is the user's entry point to the simulation pipeline. There were two primary considerations in designing this frontend. First, a DAX.program file or pyGSTi client written to be executed on the hardware needs to run identically on DPS, and the output after execution on the simulation framework should have the same format as that from the hardware. Secondly, although DPS is closely tied with DAX, we wanted it to be a standalone component.

Keeping the above considerations in mind, a program can be executed on DPS through the command-line interface (CLI). The CLI instruction takes minimal arguments, like the file to be run. Other more backend-dependent arguments like the number of qubits, type of backend to use, and the noise model are passed in through a configuration file. The DPS frontend architecture independently constructs the required DAX dependencies, parses the inputs and arguments, and passes on all the appropriate information to the backend. The frontend has been built to allow for different CLI commands depending on the type of input.

B. *Backend*

The backend of the DPS architecture handles the actual execution of the simulation. The backend has been divided into two layers - the abstract backend and the specific backend. The abstract backend is the piece that hides away all the DAX and ARTIQ details that are required for execution. It receives the backend-dependent input from the frontend and passes it on to the specific backend. It also contains performance optimizations on the operation and measurement buffers which are backend-independent. A key piece that the abstract backend hosts is the debugger. A vital use-case of simulators is to analyze and verify code, and DPS provides a debugging tool to this end. The debugger is built to give the user access to intermediate outputs from the simulation backend, and the control to flush operations and take a peek at the buffers at any point in the program.

The specific backend is designed to be a minimal piece that is required to handle three tasks. First, mapping the gates exposed by DAX.program to their counterparts on the simulation library being used. Second, handling the application programming interface (API) calls required to execute the circuit on the simulation library. And, lastly, exposing the debugger object after appropriately linking it with the simulation backend. The specific backend is designed to be minimal to allow users to add backends based on their use case. Currently, DPS supports noisy statevector and density matrix simulations using Qiskit, a Qiskit backend optimized for circuits that have no intermediate measurements and are run over multiple shots of execution, and more recently a Cirq simulation backend.

III. Results

As mentioned earlier, we want to use DPS as a tool to simulate realistic hardware by using the noise models extracted from these systems in simulation. We were able to demonstrate this functionality by comparing the results of the Direct RB protocol between the hardware and simulator. First, the Direct RB protocol was run on the physical system. These results were used to extract the noise model of the system, which was then used to run a noisy simulation of the same Direct RB protocol on DPS. The comparative results can be seen in Figure 1. The error rate of the hardware was found to be 6.98×10^{-5}, while that of the simulated result was found to be 3.87×10^{-5}, which is within a factor of 2 of the hardware

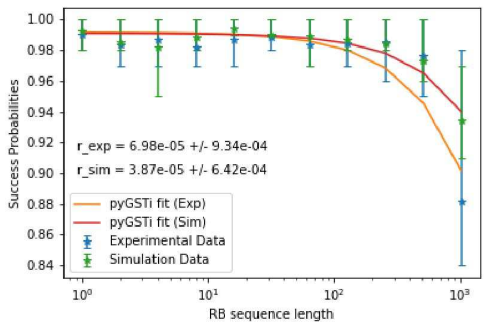

Fig. 1. Results from running Direct RB on hardware and in simulation.

results. The error bars used in the plot are calculated using the 10th and 90th percentile of the data as boundaries. The difference in the comparative results between the simulator and hardware can be explained by the constraints of defining a noise model at the gate-level.

IV. Conclusion and Future Work

With our modular simulation architecture - DAX.program-sim, we have created a flexible and portable pipeline for users to simulate their programs on multiple backends before executing them on the physical system. We demonstrated the closed loop infrastructure of running a program on hardware, getting its noise model, and simulating the same program with very comparable results.

However, to make this tool more powerful, it is vital to add support for finer levels of input and more sophisticated error models. We plan to do this by adding support for inputs at the pulse-level and an analog simulation backend that will simulate Hamiltonian evolutions. Adding these will give users finer control over the operations they want to simulate, and will also allow for more complex noise models that are a better representation of the hardware. The addition of these tools will help make DPS a test-bed for simulating and iterating over experiments and calibration routines before running them on hardware.

REFERENCES

[1] S. Bourdeauducq, R. Jördens, P. Zotov, J. Britton, D. Slichter, D. Leibrandt, D. Allcock, A. Hankin, F. Kermarrec, Y. Sionneau, R. Srinivas, T. R. Tan, and J. Bohnet, “Artiq 1.0,” May 2016.

[2] L. Riesebos, B. Bondurant, J. Whitlow, J. Kim, M. Kuzyk, T. Chen, S. Phiri, Y. Wang, C. Fang, A. V. Horn, J. Kim, and K. R. Brown, “Modular software for real-time quantum control systems,” in *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2022.

[3] L. Riesebos, B. Bondurant, and K. R. Brown, “Duke artiq extensions (dax),” 2021. https://gitlab.com/duke-artiq/dax.

[4] Erik, L. Saldyt, Rob, J. Gross, tjproct, kmrudin, T. L. Scholten, colibri coruscans, msarovar, kevincyoung, D. Nadlinger, pyIonControl, and R. Blume-Kohout, “pygstio/pygsti: Version 0.9.9.3,” Sept. 2020.

[5] E. Magesan, J. M. Gambetta, and J. Emerson, “Scalable and robust randomized benchmarking of quantum processes,” *Physical review letters*, vol. 106, no. 18, p. 180504, 2011.

[6] T. J. Proctor, A. Carignan-Dugas, K. Rudinger, E. Nielsen, R. Blume-Kohout, and K. Young, “Direct randomized benchmarking for multiqubit devices,” *Phys. Rev. Lett.*, vol. 123, p. 030503, Jul 2019.

[7] R. Blume-Kohout, J. K. Gamble, E. Nielsen, J. Mizrahi, J. D. Sterk, and P. Maunz, “Robust, self-consistent, closed-form tomography of quantum logic gates on a trapped ion qubit,” 2013.

[8] M. S. ANIS, H. Abraham, AduOffei, *et al.*, “Qiskit: An open-source framework for quantum computing,” 2021.

[9] C. Developers, “Cirq,” Mar. 2021. See full list of authors on Github: https://github .com/quantumlib/Cirq/graphs/contributors.