Qibosoq: an open-source framework for quantum circuit RFSoC programming

Rodolfo Carobene,[1,2,3] Alessandro Candido,[4,5] Javier Serrano,[3,6] Alvaro Orgaz-Fuertes,[3] Andrea Giachero,[1,2,7] and Stefano Carrazza[5,4,3]

[1] *Dipartimento di Fisica, Università di Milano-Bicocca, I-20126 Milano, Italy.*
[2] *INFN - Sezione di Milano Bicocca, I-20126 Milano, Italy.*
[3] *Quantum Research Center, Technology Innovation Institute, Abu Dhabi, UAE.*
[4] *TIF Lab, Dipartimento di Fisica, Università degli Studi di Milano and INFN Sezione di Milano, Milan, Italy.*
[5] *CERN, Theoretical Physics Department, CH-1211 Geneva 23, Switzerland.*
[6] *Departament de Telecomunicació i Enginyeria de Sistemes, Universitat Autonoma de Barcelona, ES-08193 Barcelona, Spain*
[7] *Bicocca Quantum Technologies (BiQuTe) Centre, I-20126 Milano, Italy.*

We present Qibosoq, an open-source server-side software package designed for radio frequency system on chip (RFSoC) for executing arbitrary pulse sequences on self-hosted quantum processing units. Qibosoq bridges the RFSoC firmware provided by Qick, a Quantum Instrumentation Control Kit, with Qibo, a quantum computing middleware framework. It enables experimentalists and developers to delegate all complex aspects of client-server communication protocols to the library, implementing tests and validation protocols. The client-side integration is achieved with dedicated drivers implemented in Qibolab, the specialized software module of Qibo for quantum hardware control. Therefore, this setup provides a seamless mechanism to deploy circuit-based algorithms on custom self-hosted quantum hardware platforms controlled by RFSoC electronics. We first describe the status of all components of the software package, then we show examples of control setup for superconducting qubits platforms. Finally, we present successful application results related to RFSoC performance and circuit-based algorithms.

CONTENTS

I. INTRODUCTION

The control of superconducting qubits and other quantum technologies requires instruments and software drivers for the generation and modulation of arbitrary pulses in the microwave radio frequency range [1]. With qubits being extremely delicate systems, reliable and efficient control electronics are mandatory for the successful operation of quantum hardware [2].

Nowadays, one of the major challenges of research institutions is to identify the proper set of instruments with the desired specifications and performance for quantum technologies. In the last decade, new proprietary commercial products dedicated to quantum computing have been released, some examples include Qblox [3], Quantum Machines [4], Zurich Instruments [5] among others. Despite the interest from manufacturers in commercializing expensive ready-to-run solutions, experimental laboratories still have to face the issue of acquiring instruments which are in continuous improvement in terms of firmware and software, therefore customers might participate indirectly as co-developer by providing feedback, testing and waiting for improvements. This situation is not ideal because it increases the required manpower and time of a research team. Therefore, with commercial instruments that did not yet reach fully stability, there is still an advantage in looking into lower-level electronics and open firmwares/softwares for qubit control, in order to provide more accessible tools to the research community.

Radio Frequency System on Chip [6–9] (RFSoC) FPGA (Field Programmable Gate Arrays) is a low-cost hardware alternative which provides flexible development of firmware and software related to quantum technologies. The research community has already achieved open-firmware for quantum applications through the Qick (Quantum Instrument Control Kit) project [10], that enables the use of

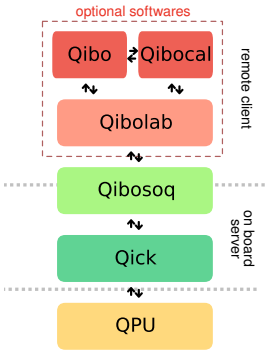

FIG. 1. Deployment pipeline from `Qibo` to the quantum processing unit (QPU). `Qibosoq` can also be used as a standalone server library, the objects in the dashed area are optional and only required to connect `Qick` and `Qibosoq` with `Qibo`.

RFSoCs to generate the sequences of pulses required for controlling and reading out qubits. Moreover, `Qick` also provides a higher-level Python library that eases the execution of pulses.

In this manuscript, we present for the first time `Qibosoq` [11], an open-source software package which unlocks `Qick`'s potential to execute quantum algorithms on self-hosted quantum hardware platforms through `Qibo`, a quantum computing framework [12–15] and its hardware module `Qibolab` [16]. `Qibosoq` provides a dedicated application programming interface (API) for sending arbitrary pulses sequences to the `Qick` firmware. `Qibosoq` has both a server component, running on the RFSoC along with `Qick`, and a remote client that can be attached to `Qibolab` or potentially other higher-level softwares. In Figure 1 we show a schematic view of the software stack from the quantum processing unit (QPU) to `Qibo`. The full pipeline is required for the execution of quantum circuit algorithms. The user can also use directly `Qibolab` for setting up experiments related to pulse generation. The stack is designed in a modular way, therefore `Qibosoq` can also be used as a standalone server application which runs on RFSoC FPGA boards opening the possibility to interface `Qick` with multiple experimental configurations.

The paper is organized as follows. In Section II we describe a detailed overview of the `Qibosoq` library for version `0.1.0`. In Section III we show performance benchmark results and example results of calibration experiments. Finally, in Section IV we draw our conclusion and discuss about future development directions.

II. METHODOLOGY

In this section, we will provide a brief overview of the current status of the `Qick` project. We will then delve into the specifics of `Qibosoq`, including its internal layout and the comprehensive set of supported features.

A. The QICK project

The `Qick` project was introduced in 2022, providing an open source qubit controller based on the Zynq UltraScale+ Xilinx ZCU111 [17] evaluation board, supporting direct radio frequency (RF) synthesis up to 6 GHz via the XCZU28DR RFSoC. It consists of a custom FPGA firmware and a Python library for coding experiments.

The firmware enables the use of the RFSoC's digital-to-analog converters (DACs) as arbitrary waveform generators for RF pulses and facilitates fast and precise acquisitions through the analog-to-digital converters (ADCs). Additionally, it features a custom timed-processor (tProcessor) that allows users to program sequences of timed pulses and loops using an assembly-like language.

Since its initial presentation, the `Qick` firmware has undergone several updates and now supports the evaluation boards RFSoC4x2 [18] and ZCU216 [19] as well, mounting respectively the XCZU48DR and XCZU49DR chip. New firmware versions have been developed with support for multiplexing, enabling readouts on different qubits coupled to the same line using the same DAC and ADC. Moreover, the team is still exploring new functionalities, including resonator simulators on the FPGA.

The role of the software side of `Qick` is to eliminate the need for writing experiments directly in the tProcessor language. Instead, it provides a more user-friendly approach by allowing users to write experiments in the Python programming language It accomplishes this by offering helper functions and program templates that streamline the coding experience for users, while still leveraging the capabilities of the tProcessor.

`Qick` is currently used in different labs around the world, with application in different technologies such as superconducting qubit control, entangled photon pair generation, and SNSPD (Superconducting Nanowire Single-Photon Detectors) readout. Several experiments have been already published using the `Qick` to control superconducting quantum hardware [16, 20–24].

The introduction of `Qick` has already proven to be beneficial for researchers in developing custom solutions for qubit control. However, from a usability perspective, there are still limitations. For example, to fully utilize the tProcessor's functionalities, users are required to write low-level code, even when using the Python API. This entails managing memory, registers, and other low-level aspects, which can be cumbersome and error-prone for users. Moreover, `Qick` operates at a pulse-level abstraction, and while this makes it a powerful tool for controlling quantum systems and executing pulse-based experiments, it is not directly suitable for quantum computing applications that rely on

gate-based circuits.

In this context, Qibosoq plays an important role by leveraging the existing Qick software and firmware to simplify the coding of quantum computing experiments in the tProcessor language. It offers a more user-friendly interface, allowing researchers to focus on their experiments, abstracting away the low-level details of the tProcessor. The experiments defined are still just based on pulses, but since Qibosoq is natively integrated with the Qibo framework, performing circuit-based experiments become straightforward through Qibolab. Moreover, Qick becomes also connected with Qibocal [25] that enables simplified and automated qubit calibration. Lastly, it is possible to use directly Qibo to execute algorithms, that are first converted into pulses by Qibolab and then executed on the RFSoC using Qibosoq. Overall, the connection between Qibosoq and Qibo extends the capabilities of Qick, enabling users to work at both the pulse-level with additional tools and gate-level abstractions.

B. Networking

The boards supported by Qick share a common architecture, that includes an on-board CPU, beyond the actual FPGA device. This design makes them independent, not requiring a further processor to execute applications. Qick itself is organized to run on this embedded processors, based on *Pynq* [26] structure.

However, the typical QPU user will not have direct access to the board, writing or loading its software there, but the access will be granted through a network. For this reason, since Qibosoq is already operating at a higher level than Qick, it is worth to take into account the networking part, controlling the communication between the board and the end user, making it as simple as possible, while flexible enough to support many different kinds of applications.

The communication layer can happen at different levels, e.g. it could transfer entire executables and run them on-board or transmit just minimal set of instructions. This might be advantageous to allow lower level access to the user, thus being free to access primitives from the programming language and Qick library. But this is not the typical use case, since most of the QPU operations share a common layout.

Recognizing the common elements allows us to factorize them out from the multiple applications into a single library, reducing duplication and simplifying the applications development. Furthermore, restricting the applications' degrees of freedom reduces the amount of information that has to be transferred on the network. These considerations led the design of Qibosoq and its own internal *language* for experiments description.

In this matter, there are other two considerations that is worth taking into account. One is related to the available on-board CPUs, the other to a partial current Qick limitation.

When taking into account where to run the application bulk, the nature of the application itself has to be examined. Some applications require to run just QPU experiments and collect their data, for further elaboration later on. This kind of application is QPU-bound, and it has very limited requirements in terms of classical processing. However, more hybrid applications are also possible: a typical example being a Quantum Machine Learning (QML) with classical optimization, where the QPU usage is interleaved with classical computation. When coming to hybrid applications there are two competing factors: latency and performances. Limiting possible applications to run on the on-board CPUs it would not scale, i.e. it would not allow to keep good performances for increasingly larger problems (as compared to the time required for execution on other commonly available hardware), becoming soon the main bottleneck. On the other hand, off-loading the computation to a separate machine introduces

communication latency during the application execution. The chosen trade-off in Qibosoq values more the potential application scaling, giving it the flexibility to support arbitrary kinds of hybrid applications, at the expense of time performances. This takes into account the current limited performances of the on-board CPUs. For instance, the Xilinx [27] boards feature ARM Cortex-A53 processors, which are considerably slower even than processors found in most modern laptops. In future, different type of boards might become available, and they could be developed having in mind quantum hybrid applications. Or the communication with an external processors might improve, not relying on a LAN but making use of local buses, like Peripheral Component Interconnect (PCI). Qibosoq approach is easily extendable to this improved scenarios, in which the same (or similar) internal language could be maintained, just acting on the communication layer implementation.

Instead, the second consideration is purely technical: each experiment in Qick requires a certain common initialization. Repeating this initialization has two main drawbacks, since on the one side it requires to pay every time the performance cost, and on the other it resets the ADCs and DACs clocks, thereby introducing a random phase between them, and consequently loosing phase coherence between experiments. This phase coherence is critical in quantum computing applications as it allows for qubit calibration to be maintained between executions, eliminating the need of a partial recalibrating every run. Because of this, it is necessary to keep a Qick instance alive between multiple user connections. This limitation requires to have at least a minimal server running on board. While Qick already provides a way of solving this problem, leveraging

the *Pyro4* library that is used to send the required objects through a network, it is still designed as an on-board software and the integration is not straightforward. `Qibosoq` is built as an alternative to this solution to make it easier to control `Qick` remotely, while also simplifying its control and integrating it with `Qibo`. In this sense, `Qibosoq` is just an extension of `Qick`, offering a higher level interface, preferable for the end user applications development.

It is relevant to note that, despite having discussed of *user applications* until here, a `Qibosoq` user might also be a higher level library, dealing with some kind of task involving QPU execution. `Qibolab` and `Qibocal` are two examples of this layout, and they showcase the flexibility of `Qibosoq`, to support arbitrary execution through an RFSoC controller. In particular, `Qibolab` also allows the execution of arbitrary circuits, described in the `Qibo` language, and shows how it is possible to deal with the transpilation and experiment preparation on the `Qibosoq` client side, making use of arbitrary classical resources. Instead, `Qibocal` is a perfect example of how multiple clients can share the same server, since its calibration routines are all potentially independent, and being executed as fully separate programs. The only requirement to interface with the `Qibosoq` server is to describe the final QPU execution using `Qibosoq` primitives, and finally establish a connection to the server.

C. Qibosoq layout

`Qibosoq` exposes various tools and abstractions to the user: the **programs**, representing `Qick` programs, and eventually taking care of the experiment compilation into the tProcessor language, the **components**, high-level structures used in the *programs* construction, the **server** implementation, and **client** utilities, to manage the communication.

The **programs** bridge the gap between the high-level interfaces (components) and the low-level execution on quantum hardware. Eventually, only two distinct programs are directly used, but the full hierarchy (presented in Figure 2) also includes intermediate abstractions. Considering all layers, the defined **programs** are:

- the abstract **base** program, that contains functions shared among all possible experiments and executions. It serves as the foundation for all the other `Qibosoq` programs;
- the abstract **flux** program, that collects the additional elements required for controlling flux-tunable qubits. In addition to the functionalities defined in **base**, it includes support for bias voltages and fast DC (direct current) pulses;
- the **sequences** and **sweepers** programs that contain the different elements used, respectively, in the execution of fixed parameters pulse sequences and real-time sweeps. They inherit all the functionalities defined in **base** and **flux**, while also being specific implementations of `Qick` classes: the `AveragerProgram` and the `NDAveragerProgram`.

With this set of **programs** it is possible to define a large variety of experiments. Indeed, the number of drive pulses, of readouts and flux pulses, is only limited by the on-board available memory. And various acquisition modes are available for all the combinations of pulses. Moreover, using **sweepers** it is possible to speed up real experiments, taking the best out of the tProcessor speed.

The **components** play the crucial role of establishing a common language for communication, easing the implementation of a `Qibosoq` client in `Qibolab` or by other parties. The main elements defined within the **components** submodule include:

- the **Config** object that contains essential general information required for executions. This includes the number of software and hardware repetitions, delay between repetitions (to ensure qubit relaxation), and whether to average the results among repetitions or not;
- the **Pulse** base object that serves as the foundation for different implemented pulse shapes. Rectangular, Gaussian and DRAG [28] pulses are natively supported, as well as custom waveform shapes defined by their "in-phase" and "quadrature" (IQ) [29, 30] values;
- the **Qubit** object that holds information about any necessary bias required for operating it;
- the **Sweeper** and **Parameter** objects that are used to describe real-time on-hardware scans.

The last two fundamental elements are the **client** and the **server**. The **client** is composed of a set of tools used to connect to the server, convert components into a serialized form, and send them following the `Qibosoq` communication protocol. The **server** implements the on-board server, continuously listening for connections, and executing received instructions by initializing and running the required programs on the quantum hardware.

More details on the communication protocol will be given in the next section.

D. Serialization and communication protocol

To run an experiment, users in the client define instructions using `Qibosoq` components, describing the experiment to be executed. These instructions are then sent to

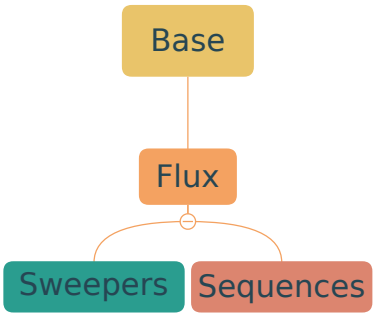

FIG. 2. Hierarchy in the **programs** submodule

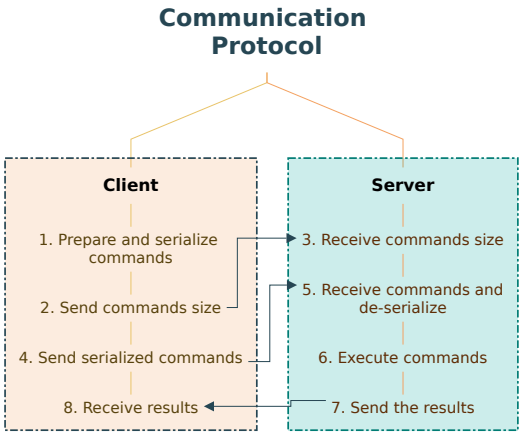

FIG. 3. Schematic of the communication protocol.

the server using the Transmission Control Protocol (TCP) for communication. TCP was chosen for its reliability [31], since it ensures data integrity, preventing any loss during transmission. Additionally, its session-based protocol suits `Qibosoq`'s requirements by disallowing multiple concurrent executions, ensuring smooth and well-controlled communication between the server and potentially multiple clients.

While the server is composed only of `Qibosoq` code, the client can be part of very different frameworks or even be just a standalone script, as long as it sends to the server the expected commands in the expected format, `Qibosoq` will work properly. Through the **client** module, `Qibosoq` offers some helpers to make following the communication protocol easier.

The `Qibosoq` communication protocol is schematically presented in Figure 3.

The first step in the process is defining the experiment to be executed, utilizing the `Qibosoq` components:

operation: `Qibosoq` supports different execution modes that can be selected through a specific component. These modes enable the execution of different type of experiments:

- fixed-parameters experiments defined as a sequence of pulses.
- varying-parameters experiments defined using a combination of sequence of pulses and sweepers.
- raw acquisition experiments, where the raw signal (after demodulation) is acquired without integration.

configuration: a **Config** object with general experiment parameters;

sequence: a list of pulses that describe the experiment, including control, flux, and readout pulses for performing measurements. Each pulse contains information about its shape, frequency, and start time.

qubits: a list of qubit objects.

sweepers: a list of sweepers, only required for real-time parameter scans on the FPGA logic. The sweepers can act on multiple parameters simultaneously or update them sequentially, enabling exploration of all combinations of chosen parameters during the experiment.

This set of instruction provides a comprehensive description of the experiment to be executed and offer a flexibility that can accommodate various qubit-related experiments.

The first step ends with the serialization, where the instructions are first dumped in the JSON format and then encoded into bytes using UTF-8.

Then, a connection is established between the client and server. Following the TCP three way handshake, a first packet containing a 32-bit integer (four bytes) is sent. This integer represents the byte-size of the instructions to be transmitted immediately afterward.

Upon receiving and de-serializing the commands, the server initializes the required program based on the specified **operation** parameter. After executing the experiment, the server sends back the results, comprising acquired "i" and "q" values, through the same TCP connection to the client.

Once the client receives all the data, the connection is closed, and the server returns to waiting for new commands.

Note that, while `Qibosoq` already provides a client that takes care of implementing this communication protocol, it does not strictly require its use, and potentially other clients could implement the same protocol and interact with the `Qibosoq` server part.

Feature	Qick	Qibolab	Qibosoq
Arbitrary pulse sequences	✓	✓	✓
Arbitrary waveforms	✓	✓	✓
Multiplex readout	✓[a]	✓	✓
Feedback	✓	✓	⚙
RTS frequency drive	✓	✓	✓
RTS frequency readout	✗[a]	✓	✗
RTS amplitude	✓	✓	✓
RTS duration	✗[b]	✓	✗
RTS start	✓	✓	✓
RTS relative phase	✓	✓	✓
RTS N-Dimensional	✓	✓	✓
Hardware averaging	✓	✓	✓
Singleshot (No Averaging)	✓	✓	✓
Integrated acquisition	✓	✓	✓
Classified acquisition	✓	✓	✓
Raw waveform acquisition	✓	✓	✓

[a] Special firmware available from `Qick` under request
[b] Supported for specific pulse shapes

TABLE I. Main features and limitations of `Qick`, `Qibosoq` and `Qibolab` compared. The features denoted by " ✓ " are supported, " ✗ " means not supported and " ⚙ " under development.

E. Features and limitations

`Qibosoq` abstracts a higher-level interface over the `Qick` primitives, and its current main purpose has been to serve `Qibolab`, despite not being restricted to it. Therefore, it is relevant to be aware of the present limitations of these related libraries when approaching `Qibosoq`, and how `Qibosoq` itself is affected by them.

In Table I a small comparison of the main features supported by `Qick`, `Qibosoq` and `Qibolab` are presented.

The following is a description of the features presented in Table I.

Arbitrary pulse sequences: the capability of executing arbitrary pulse sequences.

Arbitrary waveforms: the capability of executing pulse waveforms of arbitrary shape. All the three libraries have a set of pre-defined waveforms, but for applications of optimal control it is critical to define custom shapes.

Multiplexed readout: allows playing and acquiring multiple multiplexed pulses through the same line. It is particularly useful for multi-qubit chips where the readout line is commonly shared among multiple qubits.

Feedback: the capability of executing conditional pulses, depending on the result of a measurement. This feature is required for live error correction schemes and, being already supported by the `Qick` logic, is under development in `Qibosoq`.

RTS frequency drive: RTS (*Real Time Sweeper*) refers to the capability of executing a pulse sequence multiple times with different values of, in this case, the frequency of a drive pulse. This feature facilitates faster qubit characterization and experiments.

RTS frequency readout: real-time sweeping of the frequency of a readout pulse.

RTS amplitude: real-time sweeping of the amplitude of a pulse.

RTS duration: real-time sweeping of the duration of a pulse.

RTS start: real-time sweeping of the start time of a pulse.

RTS relative phase: real-time sweeping of the relative phase of a pulse.

RTS N-Dimensional: the capability of combining N RTS scans on different parameters.

Hardware averaging: the capability of repeating the same experiment multiple times and obtain, directly from the device, averaged results.

Singleshot (No Averaging): the capability of obtaining from the devices all the non-averaged results.

Integrated acquisition: the capability of acquiring complex signals with IQ components demodulated and integrated for the measuring time.

Classified acquisition: the capability of performing 0-1 state classification after the integrated acquisition.

Raw waveform acquisition: the capability of acquiring non-integrated IQ waveform values.

This selection of features, commonly used in standard qubit experiments, are all supported by `Qibosoq`, with the notable exception of the feedback feature that is under development.

Using `Qibolab`, is also possible to access all of these features with the sole exception of sweepers on multiple different parameters with concurrent updates, that is also currently under development. This limitation, however, is outweighted by what `Qibolab` provides, in particular with the integration in the `Qibo` framework: a way of deploying algorithms and programs written in the form of circuits directly to `Qibosoq` and to the RFSoC. This is not supported, natively, neither by `Qibosoq` and by `Qick` that both work only at the pulse level, but it becomes possible through the `Qibolab` driver for `Qibosoq`.

	ZCU111	RFSoC4x2	ZCU216
Physical DACs	8	2	16
`Qick` activated DACs[a]	7	2	7
DAC sampling rate [GSPS]	6.554	9.85	9.85
Physical ADCs	8	4	16
`Qick` activated ADCs[a]	2	2	2
ADC sampling rate [GSPS]	4.096	5	2.5

[a] These numbers are related to the standard available firmware, they can vary with other firmwares.

TABLE II. Outline of the RFSoCs supported by `Qibosoq` and their characteristics.

Some more limitations come directly from the hardware. `Qibosoq` is compatible with all the boards supported by `Qick`, namely the RFSoC4x2, the ZCU216 and the ZCU111. In Table II, the three RFSoCs are presented along with some details.

Note that, to control flux-tunable qubits, the companion boards included in the standard Xilinx kits are not sufficient, since they usually include also baluns on the single-ended outputs/inputs connected to the DACs and ADCs that filter the DC currents required for controlling the qubits. This problem can be solved by using the available differential outputs along with some differential amplifiers (as Texas Instruments THS3217 [32]) to convert the signal from double to single-ended or with the use of custom companion boards [8, 10].

III. RESULTS

All the experiments presented in this section were performed using `Qibocal`, the runcards required to reproduce them are available at [33].

A. Cross-platform benchmark

We tested `Qibosoq` using the `Qibolab` API, benchmarking its speed on different RFSoCs and against commercial instruments (Quantum Machines, QBlox and Zurich Instruments). The tested boards were the three Xilinx boards supported by `Qick`, better detailed in Table II. For better significance, in the plots only the results of Quantum Machines are shown to represent commercial instruments, as QM was the fastest among the three in the majority of cases [16].

The results are presented in Figure 4. The black bar in this plot provides the ideal time required for each routine, which is calculated as

$$\text{ideal} = n_{\text{shots}} \sum_i (T_{\text{sequence},i} + T_{\text{relaxation}}) \qquad (1)$$

where $T_{\text{sequence},i}$ is the duration of the whole pulse sequence in the i-th point of the sweep, $T_{\text{relaxation}}$ the time we wait for the qubit to relax to its ground state between experiments, n_{shots} the number of shots in each experiment and the sum runs over all points in the sweep. The ideal time denotes how long the qubit is really used during an experiment and provides the baseline for our benchmark. A description of the experiments performed, along with some of the main parameters used, is given in Appendix V A. Note also that the code required to reproduce these experiments is available at [33].

Considering only the RFSoCs, we can see that the RFSoC4x2 is always faster than the ZCU111 and the ZCU216. On the other hand, comparing it also with Quantum Machines, we can see that the `Qibosoq`-controlled devices are faster in six out of the eight performed experiments.

The key to explain the difference in speed between the `Qibosoq`-controlled boards and the commercial instruments can be found in the *Ramsey detuned* experiment timings. This routine involves sweeping the relative phase and the start time of a pulse, a feature not supported by `Qibolab`. Therefore, the experiment is performed via the execution of various pulse sequences that get generated once at a time from the client. This translates itself in a high number of communications between the client and the device. In this regard, it is possible to note that the RFSoCs present a much smaller overhead in the communication in respect to commercial instruments, partially being explained with being composed of a single device and not with multiple synchronized modules. The communication overhead, including also any time required by the instrument to set up, approximately corresponds to the first point of the plot presented in Figure 5, where the sequence execution time is negligible with respect to the overhead itself.

The only experiment where the RFSoCs are visibly slower than the commercial systems is the *resonator spectroscopy experiment* that can be conducted with a real-time sweeper, on hardware, for all the other considered devices, while it's currently not supported by `Qibosoq` and by the standard `Qick` firmware as shown in Table I

On the other hand, also the *Qubit spectroscopy* experiment, performed with a single real-time sweeper, is slower for the RFSoCs than for Quantum Machines. It is difficult to completely explain why this is the case, by looking only at Figure 4. More clear, in this regard, is Figure 5.

In this figure, the results of a benchmark on the scaling capability of sweepers (increasing the number of points swept) and of circuits (increasing the number of circuits executed) are presented. For the readout and drive frequency, the experiment-template of resonator and qubit spectroscopy was used, while for pulse amplitude and

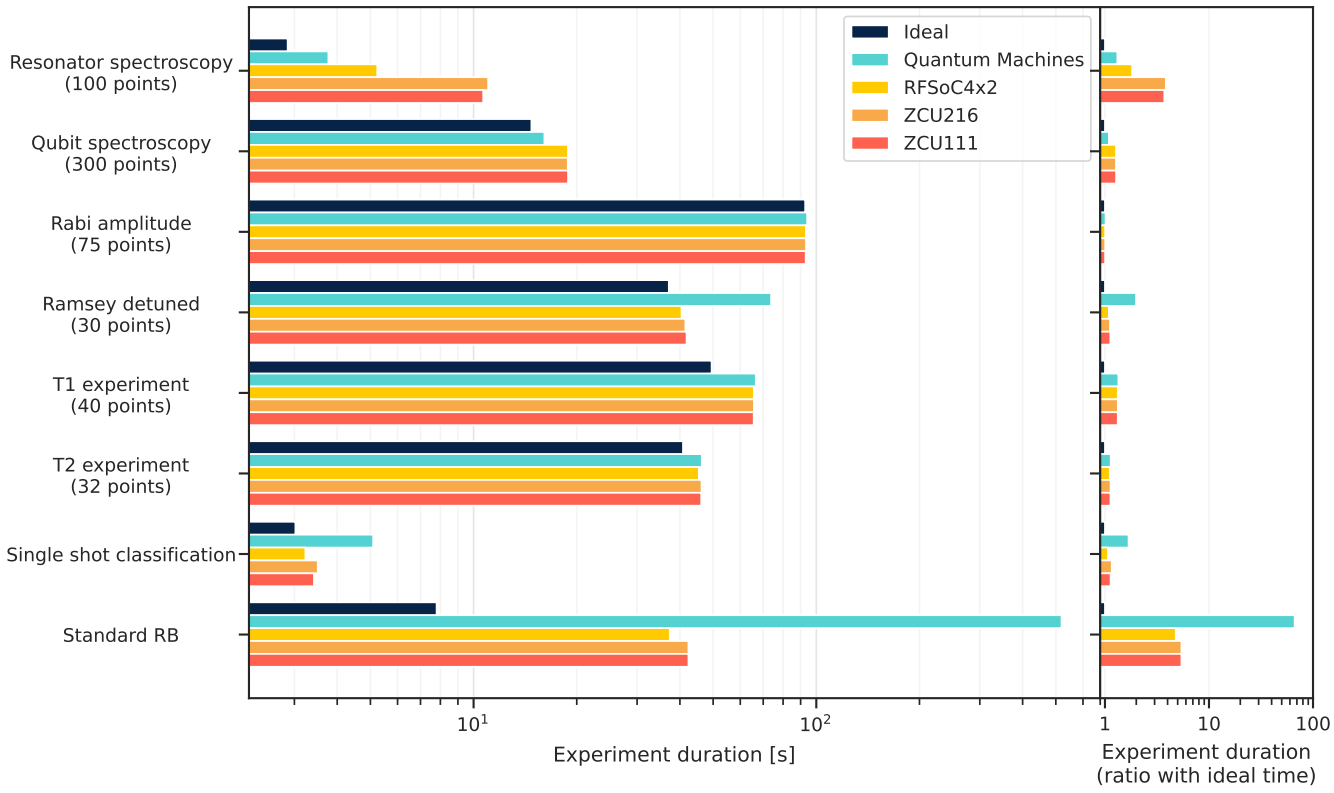

FIG. 4. Execution time of different qubit calibration routines on various electronics. On the left side we show the absolute times in seconds for each experiment. On the right side we calculate the ratio between actual execution time and ideal (minimum) time the qubit is affected. The ideal time corresponds to how long the qubit is really used during the experiment, the difference from ideal and real time comes from software delays and communication latency.

length a Rabi-like experiment was performed. For the pulse start, a standard T1 experiment was used and for the circuits a standard RB. The focus of this plot is, however, the parameter swept (or the number of circuits executed) and not the experiments themselves.

We can clearly see which sweepers are not implemented with real-time sweepers (on the readout frequency and on the pulse length), not considering the "Circuits" plot that cannot logically being implemented with sweepers. By looking at the "Drive frequency" plot we can get an idea of the reason why the *Qubit spectroscopy* experiment appeared "slow". Initially, where the effective duration of the experiment is given almost completely by the overhead, the RFSoCs perform much better than the commercial systems. Increasing the number of swept values, however, decreases the difference of duration between instruments, with the commercial ones that even becomes slightly better than the RFSoCs for many-points scans.

B. Calibration experiments

Alongside the presented speed benchmarks, we conducted tests of `Qibosoq` on quantum hardware, connecting the available boards to various single qubits, without flux dependency, manufactured by TII (Technology Innovation Institute) and a flux-tunable multi-qubit chip, by QuantWare [34].

While performing a comprehensive quality benchmark is challenging due to the numerous variables in calibration, the RFSoC-based system demonstrated competitiveness with commercial instruments. There is also evidence that the direct RF synthesis [35], which avoids sidebands produced by IQ mixer up-conversion, generally yields higher quality factors and signal-to-noise ratios.

Examples of the obtained results are shown in Figure 6.

The initial two plots feature the Xilinx RFSoC4x2 connected to a single qubit within a 3D cavity. Achieving an assignment fidelity of 0.95 and gate fidelities in agreement with theoretical limitations [36] from the observed T_1 and T_2 values, that in turn align with design parameters.

The third plot involves a multi-flux-tunable-qubit setup controlled by the Xilinx ZCU216 RFSoC. This experiment

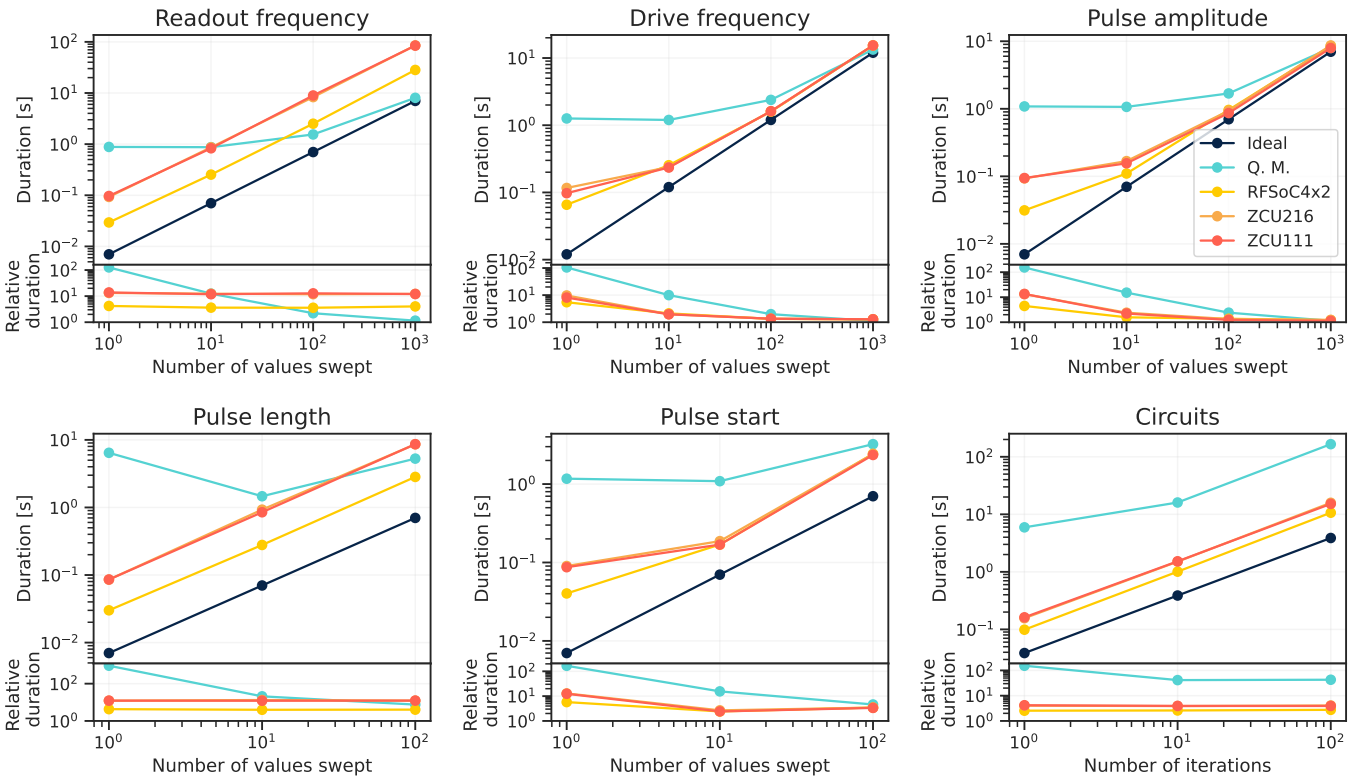

FIG. 5. Scaling of execution time as a function of the number of points in a sweep. Bottom plots show the ratio between real execution on different instruments and minimum ideal time. "Q.M." stands for Quantum Machines. The corresponding values were obtained using the same `Qibocal` interface. The ideal time corresponds to how long the qubit is really used during the experiment, the difference from ideal and real time comes from software delays and communication latency.

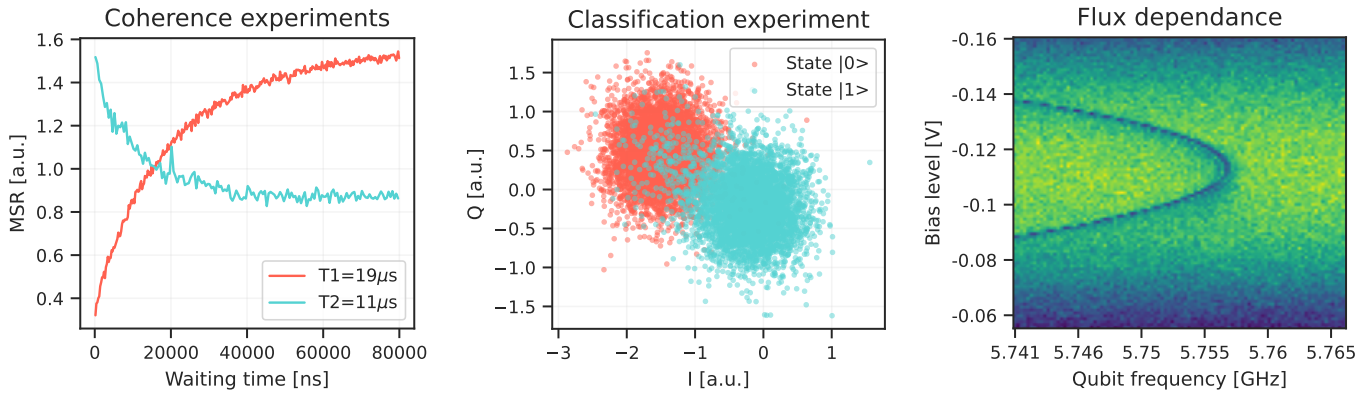

FIG. 6. Examples of calibration experiments. In the first experiment, the relaxation and dephasing time of a single qubit are computed. The second plot present a classification experiment from which an assignment fidelity of 0.95 was computed. In the last experiment, the dependency of the 0-1 transition frequency to an applied bias was analyzed.

example aimed to investigate the correlation between the 0-1 transition frequency and a bias current.

IV. OUTLOOK

In this paper we have presented `Qibosoq`, an open-source server-side software for RFSoC control electronics through `Qick`. This setup simplifies operation of self-host quantum hardware platforms through `Qibo`, a full-stack

quantum computing middleware framework.

We have described the project structure with the major features implemented in release `0.1.0`. The software is at the stage of allowing applications related to performance benchmarks through arbitrary pulse control and physics experiments based on the quantum circuit representation respectively with the APIs of `Qibolab` and `Qibo`.

In the future releases of `Qibosoq`, we plan to extend its capabilities to support multi-qubit configurations controlled by synchronized RFSoC boards and test the framework on novel quantum technologies such as trapped ions, neutral atoms and photonics among others. On top of the challenges related to physics and experimental setup, we believe that `Qibosoq` accelerates development in the field of control electronics, with particular emphasis on training of researchers by providing a full open-source base for FPGA-based control electronics.

The `Qibosoq` module and all results can be reproduced using the code at:

https://github.com/qiboteam/qibosoq.

ACKNOWLEDGMENTS

This project is supported by TII's Quantum Research Center. The authors thank the `Qick` team for helpful discussion, comments on this manuscript and support. S.C. thanks CERN TH hospitality during the elaboration of this manuscript. A.G. acknowledges support by the Horizon 2020 Marie Skłodowska-Curie actions (H2020-MSCA-IF GA No.101027746).

[1] D. Riste, S. Fallek, B. Donovan, and T. A. Ohki, IEEE Microwave Magazine **21**, 60 (2020).
[2] T. D. Ladd, F. Jelezko, R. Laflamme, Y. Nakamura, C. Monroe, and J. L. O'Brien, Nature **464**, 45 (2010).
[3] Qblox, `https://www.qblox.com/`.
[4] QuantumMachines, `https://www.quantum-machines.co/`.
[5] ZurichInstruments, `https://www.zhinst.com/others/en/quantum-computing-systems/qccs`.
[6] A. Javaid, T. Ahmed, and S. Ali, in *2022 19th International Bhurban Conference on Applied Sciences and Technology (IBCAST)* (IEEE, 2022).
[7] R. Gebauer, N. Karcher, and O. Sander, in *2021 International Conference on Field-Programmable Technology (ICFPT)* (IEEE, 2021).
[8] M. O. Tholén, R. Borgani, G. R. D. Carlo, A. Bengtsson, C. Križan, M. Kudra, G. Tancredi, J. Bylander, P. Delsing, S. Gasparinetti, and D. B. Haviland, Review of Scientific Instruments **93**, 104711 (2022).
[9] U. Singhal, S. Kalipatnapu, P. K. Gautam, S. Majumder, V. V. L. Pabbisetty, S. Jandhyala, V. Singh, and C. S. Thakur, IEEE Transactions on Instrumentation and Measurement **72**, 1 (2023).
[10] L. Stefanazzi, K. Treptow, N. Wilcer, C. Stoughton, C. Bradford, S. Uemura, S. Zorzetti, S. Montella, G. Cancelo, S. Sussman, A. Houck, S. Saxena, H. Arnaldi, A. Agrawal, H. Zhang, C. Ding, and D. I. Schuster, Review of Scientific Instruments **93**, 10.1063/5.0076249 (2022).
[11] R. Carobene, A. Candido, J. Serrano, S. Carrazza, and Edoardo-Pedicillo, qiboteam/qibosoq: Qibosoq 0.0.4 (2023).
[12] S. Efthymiou, S. Ramos-Calderer, C. Bravo-Prieto, A. Pérez-Salinas, D. García-Martín, A. Garcia-Saez, J. I. Latorre, and S. Carrazza, Quantum Science and Technology **7**, 015018 (2021).
[13] S. Efthymiou, M. Lazzarin, A. Pasquale, and S. Carrazza, Quantum **6**, 814 (2022).
[14] S. Carrazza, S. Efthymiou, M. Lazzarin, and A. Pasquale, Journal of Physics: Conference Series **2438**, 012148 (2023).
[15] S. Efthymiou, S. Carrazza, R. Mello, Edoardo-Pedicillo, A. Pasquale, C. Bravo-Prieto, A. Sopena, M. Robbiati, AdrianPerezSalinas, shangtai, S. Ramos, D. García-Martín, M. Lazzarin, BrunoLiegiBastonLiegi, N. Zattarin, Simone-Bordoni, vodovozovaliza, L. Zilli, Paul, A. Candido, A. Mak, J. Serrano, atomicprinter, and J. M. Cruz-Martinez, qiboteam/qibo: Qibo 0.1.12 (2023).
[16] S. Efthymiou, A. Orgaz-Fuertes, R. Carobene, J. Cereijo, A. Pasquale, S. Ramos-Calderer, S. Bordoni, D. Fuentes-Ruiz, A. Candido, E. Pedicillo, M. Robbiati, Y. P. Tan, J. Wilkens, I. Roth, J. I. Latorre, and S. Carrazza, Qibolab: an open-source hybrid quantum operating system (2023).
[17] Xilinx-(AMD), Zcu111 specifications, `https://www.xilinx.com/products/boards-and-kits/zcu111.html` (2022).
[18] Xilinx-(AMD), Rfsoc 4x2 specifications, `https://www.xilinx.com/support/university/xup-boards/RFSoC4x2.html` (2022).
[19] Xilinx-(AMD), Zcu216 specifications, `https://www.xilinx.com/products/boards-and-kits/zcu216.html` (2022).
[20] J. Bryon, D. Weiss, X. You, S. Sussman, X. Croot, Z. Huang, J. Koch, and A. A. Houck, Physical Review Applied **19**, 10.1103/physrevapplied.19.034031 (2023).
[21] J. G. C. Martinez, C. S. Chiu, B. M. Smitham, and A. A. Houck, Flat-band localization and interaction-induced delocalization of photons (2023).
[22] S. Xie, L. Stefanazzi, C. Wang, C. Pena, R. Valivarthi, L. Narvaez, G. Cancelo, K. Kapoor, B. Korzh, M. Shaw, P. Spentzouris, and M. Spiropulu, Entangled photon pair source demonstrator using the quantum instrumentation control kit system (2023).
[23] A. Anferov, K.-H. Lee, F. Zhao, J. Simon, and D. I. Schuster, Improved coherence in optically-defined niobium trilayer junction qubits (2023).

[24] I. N. Moskalenko, I. A. Simakov, N. N. Abramov, A. A. Grigorev, D. O. Moskalev, A. A. Pishchimova, N. S. Smirnov, E. V. Zikiy, I. A. Rodionov, and I. S. Besedin, npj Quantum Information **8**, 10.1038/s41534-022-00644-x (2022).
[25] A. Pasquale, Edoardo-Pedicillo, DavidSarlle, S. Efthymiou, S. Carrazza, aorgazf, A. Sopena, maxhant, A. Candido, M. Robbiati, vodovozovaliza, S. Ramos, and wilkensJ, qiboteam/qibocal: Qibocal 0.0.2 (2023).
[26] Xilinx-(AMD), Pynq: Python productivity for zynq (2018).
[27] Xilinx-(AMD), Xilinx, `https://www.xilinx.com` (2022).
[28] J. M. Gambetta, F. Motzoi, S. T. Merkel, and F. K. Wilhelm, Physical Review A **83**, 10.1103/physreva.83.012308 (2011).
[29] L. E. Franks, *Signal Theory*, Prentice-Hall electrical engineering series (Prentice Hall, Old Tappan, NJ, 1969).
[30] P. S. V. Naidu, *Modern Digital Signal Processing* (Alpha Science International, 2003).
[31] Z. Kanmai, in *2020 International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI)* (IEEE, 2020).
[32] T. Instruments, Ths3217 specifications, `https://www.ti.com/lit/ds/symlink/ths3217.pdf` (2016).
[33] The Qibo Team, Qibosoq paper runcards, `https://github.com/qiboteam/qibosoq/tree/main/extras/qibosoq_paper_runcards` (2023).
[34] QuantWare, `https://www.quantware.eu/`.
[35] W. D. Kalfus, D. F. Lee, G. J. Ribeill, S. D. Fallek, A. Wagner, B. Donovan, D. Riste, and T. A. Ohki, IEEE Transactions on Quantum Engineering **1**, 1 (2020).
[36] T. Abad, J. Fernández-Pendás, A. F. Kockum, and G. Johansson, Physical Review Letters **129**, 10.1103/physrevlett.129.150504 (2022).
[37] Y. Y. Gao, M. A. Rol, S. Touzard, and C. Wang, PRX Quantum **2**, 040202 (2021).
[38] M. Naghiloo, Introduction to experimental quantum measurement with superconducting qubits (2019), arXiv:1904.09291 [quant-ph].

V. APPENDIX

A. Cross-platform benchmark

In this section we provide some more details on the experiments performed for the performance benchmark presented in Section III A. A more detailed description of these routines is given by [37, 38]. All these experiments were repeated for 4096 shots. For spectroscopies, a relaxation time of 5 μs was used, while for the other experiments it was set at 300 μs. Relaxation time is the waiting time between consecutive shots to let the qubit relax back to the ground state before the next shot is started.

Resonator spectroscopy: consists of a single-tone spectroscopy where a pulse is sent through the readout line and acquired through the feedback line. The frequency of the pulse is swept in a specific range, in our case probing 20 or 100 different frequencies. In the calibration of a 3D (2D) resonator, the amplitudes acquired present a positive (negative) peak at the resonance frequency of the resonator.

Qubit spectroscopy: consists of a two-tone spectroscopy where a first pulse is sent to the drive line and a measurement (a readout pulse and an acquisition) is performed right after. The frequency of the drive pulse is swept in a specific range. In the example used for the benchmark, 300 frequencies were analyzed. As per the resonator spectroscopy, the amplitude acquired presents a peak for a specific frequency that, in this case, will be used as the drive pulse frequency.

Rabi amplitude: first a drive pulse, at the frequency identified with qubit spectroscopy, is sent through the drive line and a measurement is performed right after. The amplitude of the first pulse is swept in a range composed of, in this case, 75 points. This experiment is used to calibrate the amplitude of the pi-pulse (Pauli-X gate) which rotates the qubits from the $|0\rangle$ state to $|1\rangle$.

Ramsey detuned: a first pulse is sent through the drive line. Then, after a delay, a new drive pulse is sent with a delay dependent phase and finally a measurement is performed. The delay between the two drive pulses, and therefore the phase, are swept. This experiment is used to fine tune the drive pulse frequency.

T1 experiment: the qubit is excited using a calibrated pi-pulse, then measured after a variable time. The characteristic decay shown by this experiment is used to measure the relaxation time T1 of the qubit.

T2 experiment: this experiment is almost identical to the Ramsey detuned experiment, but no additional phase is introduced in the second drive pulse. This enables to compute the characteristic dephasing time T2.

Single shot classification: The qubit is first just measured at the initial $|0\rangle$ state, and then excited and measured in the $|1\rangle$ state. The results are used to calibrate the classification between measured states.

Standard RB: First, a certain number (iterations) of circuits composed of Clifford gates is randomly generated. These circuits are executed and an average fidelity is computed. Then, new circuits are generated with increased depth and the procedure is repeated. The fidelity is supposed to decrease exponentially with the number of gates per circuit, leading to an estimation of the average error per gate.