2022 IEEE International Conference on Quantum Computing and Engineering (QCE) | 978-1-6654-9113-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/QCE53715.2022.00077

Modular software for real-time quantum control systems

Leon Riesebos*†, Brad Bondurant*, Jacob Whitlow*, Junki Kim‡, Mark Kuzyk*, Tianyi Chen§, Samuel Phiri*, Ye Wang*, Chao Fang*, Andrew Van Horn*, Jungsang Kim* and Kenneth R. Brown*
*Department of Electrical and Computer Engineering, Duke University, NC 27708, USA
†Email: leon.riesebos@duke.edu
‡SKKU Advanced Institute of Nanotechnology (SAINT) and Department of Nanoengineering, Sungkyunkwan University, Suwon 16419, Korea
§Department of Physics, Duke University, NC 27708, USA

***Abstract*—Real-time control software and hardware is essential for operating quantum computers. In particular, the software plays a crucial role in bridging the gap between quantum programs and the quantum system. Unfortunately, current control software is often optimized for a specific system at the cost of flexibility and portability. We propose a systematic design strategy for modular real-time quantum control software and demonstrate that modular control software can reduce the execution time overhead of kernels by 63.3% on average while not increasing the binary size. Our analysis shows that modular control software for two distinctly different systems can share between 49.8% and 91.0% of covered code statements. To demonstrate the modularity and portability of our software architecture, we run a portable randomized benchmarking experiment on two different ion-trap quantum systems.**

***Index Terms*—real-time control systems, modular software, software portability, quantum computing**

I. Introduction

The field of quantum computing is rapidly evolving in the areas of software and hardware. On the software side, quantum programming languages and compilers are becoming more available and feature-rich [1]–[6]. At the same time, quantum hardware is becoming increasingly powerful with recent systems demonstrating computations on tens of qubits [7]–[13]. An often underexposed area in the field of quantum computing is the control software and hardware that bridges the gap between the quantum program and the targeted quantum system. Recent papers [7], [11], [14], [15] have shown that current state-of-the-art quantum systems already require tens to hundreds of devices to be controlled with high precision and strict real-time requirements, proving to be a significant challenge for the control system. Existing control hardware as described in [16]–[20] provides the required real-time control of devices, but it is up to the real-time control software to close the remaining gap between device-level control hardware and quantum programs as illustrated in Figure 1.

Control software for quantum systems that runs on the real-time controller is similar to high-performance and resource-constrained embedded software. The software is responsible for real-time control of a set of devices while capturing and processing data simultaneously. Real-time controlled devices include direct digital synthesis (DDS) devices, digital I/O, and digital-to-analog converters (DACs). Additionally, the real-time control system for a quantum system functions as a coprocessor and maintains a connection with a host computer. Due to the performance requirements and the strong dependence between the software and hardware, real-time control software is often tailored for a specific quantum system at the cost of flexibility and portability. Since quantum computing is still an emerging technology, most quantum systems are unique. As a result, real-time control software is often redeveloped for each system which causes significant development overhead.

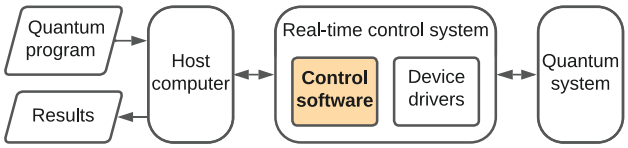

Fig. 1. A real-time control system bridging the gap between the quantum program and the quantum system.

In this paper, we propose a systematic design strategy for real-time quantum control software. We present an open-source software framework for the advanced real-time infrastructure for quantum physics (ARTIQ) open-source software and hardware ecosystem [16], [21] to apply our design concepts to real-time quantum control software. Our framework supports the development of modular control software to enhance flexibility and portability on the level of real-time system code. Portability on the application level is achieved by introducing software abstractions. We show that modular control software developed with our framework can reduce the execution time overhead of real-time software and achieve high degrees of code portability. Reduced execution time overhead is achieved by fine-grained timing management and data offloading features of the modular software. We demonstrate the capabilities of our software framework by running a portable randomized benchmarking experiment on two different ion-trap quantum systems that are fully controlled by software based on our framework.

The remainder of this paper is structured as follows. Related work is discussed in Section II. In Section III we present our design strategy and modular software architecture for real-

978-1-6654-9113-6/22/$31.00 ©2022 IEEE
DOI 10.1109/QCE53715.2022.00077

time quantum control software. Our performance analysis and code portability analysis can be found in Section IV and V, respectively. In Section VI we present experimental results from two ion-trap quantum systems. We conclude our paper in Section VII.

II. Related work

Control software for ARTIQ systems can be developed without the use of our proposed framework. Programmers will have access to a classically complete programming environment, basic data storage utilities, and device drivers to program real-time devices. However, ARTIQ is set up as a fully generic control system and no utilities are provided to support modular software development. As a result, control software is often tightly coupled to the hardware and needs to be completely redeveloped for each system. Especially the real-time timing of the software is often highly dependent on the controlled devices. The tight coupling of the hardware and software makes it difficult to change devices in existing systems since any modification likely introduces timing issues. At this moment, we are not aware of any other frameworks or libraries to support the development of modular control software for ARTIQ. Other real-time control systems similar to ARTIQ, such as M-ACTION [11], [17] and IonControl [18], suffer from the same limitations.

QCoDeS [22] is a modular data acquisition framework mainly intended to orchestrate the setup and data collection of instruments and devices part of a quantum control system. Some of these instruments can have real-time features, but QCoDeS does not control instruments in real-time while an experiment runs. Instead, all code involving QCoDeS runs on the host computer in a Python environment. Real-time instruments are also much more coarse compared to ARTIQ. A single ARTIQ real-time controller with many real-time I/O devices would correspond to a single QCoDeS instrument. The concepts behind QCoDeS show some similarities with the non-real-time components of the ARTIQ host environment. What sets ARTIQ apart from QCoDeS is its seamless integration and combination of real-time software within the host environment. ARTIQ puts the real-time controller in the center based on the principles of the accelerator model, while QCoDeS behaves more as a hypervisor for devices. The concepts presented in this paper apply to low-level real-time control software and its interaction with the host, and QCoDeS is not involved in the former. QCoDeS does have features for software modularity but these are limited to the instrument level without introducing any form of hierarchy. Other software based on or derived from QCoDeS, such as PycQED [23], is built on the same principles and has the same limitations.

Qiskit [3] is an open-source library for creating, compiling, and executing quantum programs at the gate level. While it does allow users to execute quantum programs, Qiskit itself does not transparently connect to any device-level drivers and is not directly involved in the real-time control of devices when the compiled quantum program runs. Hence, Qiskit merely describes a circuit and is not directly part of the real-time control software that runs the circuit on hardware. Even the pulse-level control provided in Qiskit is an opaque abstraction over the device-level drivers. The same holds for related and similar tools such as OpenQASM [2], OpenPulse [24], Q#[1], and Cirq [6].

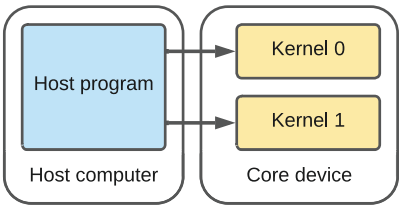

Fig. 2. Schematic overview of the accelerator model with a host program and one or more kernels.

III. Software architecture

Our software architecture targets the advanced real-time infrastructure for quantum physics (ARTIQ) open-source software and hardware ecosystem [16], [21] which is used by dozens of research groups and has deployed over 200 real-time control systems worldwide. ARTIQ follows the principles of the *accelerator model* [1], [4], [19], [25]–[29] where a program consists of a host program and one or more *kernels*. The host program executes on a host machine and can offload the execution of kernels to an accelerator. For ARTIQ, the host program runs on a classical computer while kernels run on the real-time control hardware as illustrated in Figure 2. ARTIQ kernels are classically complete and have access to real-time devices that interact with the quantum system (e.g. direct digital synthesis (DDS) devices, digital I/O, and digital-to-analog converters (DACs)). The real-time control hardware, referred to as the *core device*, contains a classical CPU and a real-time I/O (RTIO) subsystem that schedules events for real-time devices on a timeline using a timeline cursor as described in [19], [30]. The ARTIQ system is programmed in a Python host environment and kernels are functions written in the ARTIQ domain-specific language (DSL), which is a Python-like language containing additional constructs for manipulating the timeline cursor and inserting events. ARTIQ allows host and kernel code to be combined in a single file, and kernels are functions or methods decorated with the `@kernel` decorator. When the host calls a kernel function, the host will invoke the ARTIQ compiler that compiles the kernel to a binary. The resulting binary is uploaded to and executed by the core device. While the core device executes the kernel, the host serves any synchronous or asynchronous remote procedure calls (RPCs) from the core device. RPCs are often used to stream real-time data to the host, access peripheral (i.e. non-real-time) devices, and offload computationally heavy tasks to the host. Once the kernel finishes execution, the host program resumes execution. The ARTIQ programming environment is set up generically and does not provide additional utilities for organizing real-time control software. In this section, we will

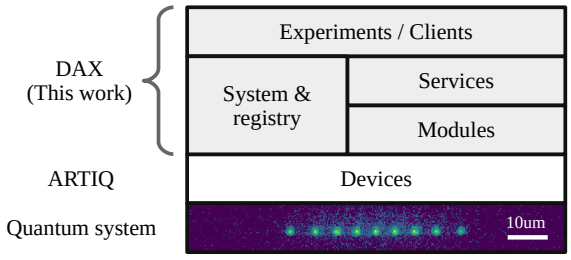

Fig. 3. Schematic overview of the software components in our modular architecture controlling a quantum system, in this case, trapped atomic ions.

present our design principles for real-time control software written for an ARTIQ control system.

The first step towards code organization is to separate common functionality of the system (i.e. *system code*) from experiment-specific routines (i.e. *experiment code*). System code can be collected in a base class, and each experiment can inherit from such a class and add experiment-specific routines. This approach is already common practice for most ARTIQ experiments. Our software architecture focuses on the development of modular system code. We propose to break the system code into two components: device organization with *modules* and extensible system-wide functionality using *services*. Additionally, we will introduce the notion of a central and searchable *registry* in which all modules and services of a system are registered. To improve code portability, we introduce abstractions with *interfaces* and *clients*. Figure 3 shows an architectural overview of the different components, which we will describe in the remainder of this section.

A. Modules, services, and the registry

While most experiments require a large set of real-time devices to collaborate closely, subsets of devices that perform basic procedures often have a tighter relation from a control perspective. A subset of devices might have strict control or safety requirements independent from other devices in the system. For example, two devices might always need to be switched simultaneously to achieve some desired functionality. We introduce the concept of modules to group such a logical collection of devices.

A module is self-contained and controls zero or more devices that depend on each other to perform basic procedures. For example, a detection module can contain devices required to apply a readout signal to the system together with the input devices that read the state of the qubits during detection. To guarantee that modules are *independent* from a control perspective, a device can only be assigned to a single module. Each module has access to its own persistent data storage to store configuration and calibration data related to its operation. The collective behavior of devices in a module can be described using module functions. For example, a detection module could have a function that controls the readout signal and the input devices in parallel to perform a detection procedure. Module functions are not solely used for collective device behavior and can also be used to manipulate devices separately, read or write configuration data, or update calibration parameters.

A system contains one or more modules that are organized in a tree structure. Every module in the system can contain zero or more sub-modules, and the root module is known as the *system module*. Parent modules can access features of all their child modules, allowing hierarchical and transparent structuring of devices and functionality. Because each device can only be assigned to a single module, modules in non-overlapping sub-trees of the system hierarchy are *device-independent*. Two independent modules can be controlled in parallel, which means they can both add events to the RTIO event timeline without device conflicts. Hence, the width of the module tree represents the amount of control- and device independence between different parts of the system. Device dependencies are encoded in the tree structure, and more independent modules lead to more available control parallelism in the system. All modules are added to the central registry of the system such they can be easily found later. Modules form the first level of system organization and introduce fundamental abstractions for device control and dependencies.

Modules introduce a straightforward device and system organization, but each separate module has limited power due to its local scope. Only the system module (i.e. the root module) can control all devices in the system, a requirement for most meaningful operations on the quantum system. With only modules, all system-wide functionality would have to be implemented in the system module, reducing the modularity of the software architecture. To overcome this issue, we introduce services as a technique to organize system-wide functionality.

A service is a component that can control multiple modules or even the whole system if desired. Through the registry, a service can obtain any modules required for its functionality. A single module in the system can be accessed by any number of services. The functions of services usually describe the collective behavior of multiple modules, and functionality can vary from short operations to lengthy procedures. If modules are device-independent, a service is allowed to control them in parallel. Just as modules, services have access to their own persistent data storage, and every service is added to the central registry of the system.

Services are not limited to calling just modules. Through the registry, services can also find other services to use their functionality. Building services on top of other services allows transparent layering of increasingly complex system behavior. Hence, services are organized in a directed acyclic graph (DAG). Services contain powerful functions and enable the organization of system-wide functionality, but because of their system-wide control, services can not operate on the system in parallel with any other module or service as this could potentially lead to device-control conflicts.

B. Interfaces and clients

System code can be organized with the concepts described in Section III-A, but because most quantum systems are

unique, the majority of modules and services are still developed and optimized for a specific system. To abstract system code, we introduce standardized interfaces. Interfaces describe a set of functions that must be implemented by a module or service. A single module or service can implement multiple interfaces and a single interface can be implemented by multiple modules or services in a system. For example, a gate interface could expose a set of functions that implement operations to perform quantum gates. Multiple instances of a gate interface within a single system could represent different gate implementations. To prevent device- and control conflicts, interfaces should be considered as implemented by a service. Hence, an interface can not operate on a system parallel with any other module, service, or interface.

System code that implements one or more standardized interfaces can run portable experiments, which we call clients. Clients exclusively control a system through interfaces. A client is instantiated against a system, and the necessary interfaces will be obtained at runtime using the system registry. With clients, we can develop portable experiments that can run on different systems or different implementations of an interface in a single system. The system code functions as middleware between the generic experiment described in the client and the system-specific implementation of the utilized interfaces.

C. Implementation

We have implemented a software framework to support the development of real-time control software based on the presented concepts. The framework is part of our open-source library Duke ARTIQ extensions (DAX) [31] , which integrates tightly with the ARTIQ open-source software and hardware ecosystem. DAX implements a set of base classes that developers must inherit when defining modules and services. All these base classes provide direct access to data storage functions and the central registry of the system. When modules or services are instantiated, a unique hierarchical key and data storage location is assigned to the object based on the module tree or service DAG. In addition, the DAX library contains standard modules and services with portable functionality that can be used by any system. Such modules and services include functionality for processing measurement data, device-safety control, and common device control. Additionally, we have developed a generic class that defines the standard control flow of a single- or multi-dimensional scanning-type calibration experiment. Our DAX framework relies heavily on multiple inheritance to combine features of multiple classes, and fortunately, the Python host environment supports this well.

DAX defines various interfaces that can be implemented by modules and services. The two interfaces of interest for this paper are the *operation interface* and the *data-context interface*. The operation interface contains functions for common gate-level quantum operations, including single- and two-qubit Clifford operations, arbitrary rotation gates, and qubit state preparation/measurement. The data-context interface is used to store and process obtained measurement results. We developed clients to perform randomized benchmarking (RB) [32]–[34] and gate set tomography (GST) [35] which use the operation interface and the data-context interface to execute benchmark circuits. The RB and GST clients work with every system that uses DAX-based control software and implements the required interfaces. Both clients are based on the open-source pyGSTi library [36] which is used to generate benchmarking circuits and analyze results. Finally, we have defined an application programming interface (API) that can be used to write portable quantum programs in an ARTIQ environment given an operation interface and a data-context interface. Using this API, we implemented a program to perform single-qubit state tomography (SQST) [37]. Such portable programs can be executed by dynamically linking the program to the interfaces of a system using the program loader client we developed.

IV. Performance evaluation

To evaluate the benefits and overhead of modular control software developed with the DAX framework, we re-implemented the control software for the software-tailored architecture for quantum co-design (STAQ) system, an experimental trapped-ion quantum processor . The real-time control hardware of STAQ is based on a Kasli 2.0 controller [21], which is part of the ARTIQ hardware ecosystem. The old ARTIQ control software for STAQ is designed with a system-specific and monolithic architecture while the new modular control software is developed using our DAX framework. In this section, we will compare the old control software with the new DAX-based control software.

The old STAQ control software separates system code from experiment code, but the system code has a monolithic architecture and is highly hardware dependent. For example, code related to RTIO timing is highly dependent on the latencies introduced by the programming of real-time devices. Any changes to the devices will likely cause RTIO timing constraints violations throughout the code. Hence, the old control software is not modular or portable. We did make modifications to the old control software to optimize its performance and make the comparison to the new control software fair. All delays inserted on the event timeline used to compensate for device programming times larger than 200 us were reduced to 200 us or replaced by other efficient solutions that satisfy timing constraints. Such delays are often necessary to not violate any timing constraints of the RTIO system. The new control software also uses 200 us delays to compensate for device programming time, which has been found empirically to be sufficient. Minor bugs found in the old code were also fixed to ensure the old and new experiments are functionally equivalent.

The new DAX-based system code for STAQ is modular and organized in 11 modules and 11 services. Most modules and services are system-specific, but two services use portable DAX data-processing modules, and one module extends a DAX module with safety-related functionality. The DAX-based system code implements various DAX interfaces, including the data-context interface and four implementations

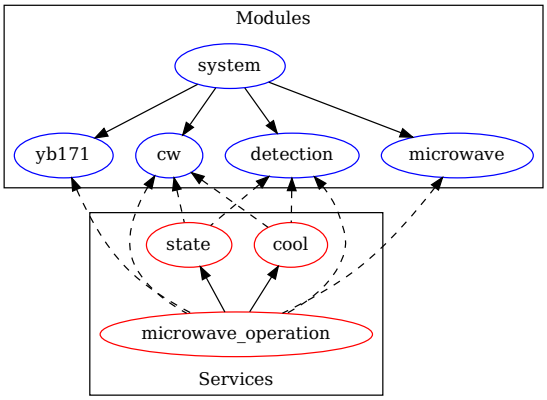

Fig. 4. Subset of the STAQ modules and services relevant for the microwave operation service.

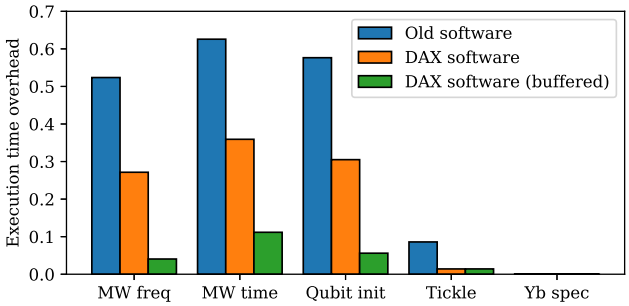

Fig. 5. Kernel execution time overhead for the old control software and new DAX-based control software of the STAQ system.

of the operation interface. The new system code packs more features and complexity, including control over more real-time devices (increased from 23 to 35) and external devices (increased from 4 to 8). Figure 4 shows a subset of the STAQ modules and services relevant for the microwave operation service, which implements the DAX operation interface. Solid arrows show the tree structure and DAG dependencies for modules and services, respectively. The dashed arrows indicate modules that are directly used by services. The microwave module controls a DDS to apply microwave pulses to the ions. The photomultiplier tubes (PMTs) and lasers used for detection are controlled by the detection module. The continuous wave (CW) module controls various other lasers for cooling and pumping while the Yb171 module stores ion calibration data. The cool service contains various subroutines for cooling ions while the state service implements the data-context interface and is used for state initialization and detection. Finally, the microwave operation service uses all the mentioned modules and services to perform microwave gates, qubit state preparation, and measurements.

We chose five relevant experiments with a single real-time kernel available in both the new and the old STAQ control software for comparison. The selected experiments include two microwave (MW) experiments (MW freq/time), a qubit initialization experiment (qubit init), a tickle experiment (tickle), and an Ytterbium spectroscopy experiment (Yb spec). All experiments are one-dimensional (1D) scanning-type experiments and scan over 20 data points. The new control software utilizes the generic DAX scanning infrastructure while the old control software has defined scanning control-flow procedures as part of the system code. Each experiment takes 100 samples per point except for Yb spec, which takes 30 samples per point. We ran each experiment with the same configuration using the old and new control software. Additionally, we run each experiment using the new control software with buffering enabled. Buffering allows the real-time control software to schedule the operations for the next samples while the incoming data of earlier samples are kept temporally in hardware buffers. ARTIQ supports such hardware buffers, but the real-time software must be designed appropriately to utilize them. Buffering can further increase the throughput and performance of kernels by reducing stalling time at the cost of increased latency between receiving and processing input events. None of the mentioned experiments are sensitive to the increased latency and will benefit from increased throughput. We configure a buffer size of 16 samples, which should be large enough to get the maximum performance gain achievable with buffering. The old control software does not include features for buffering. We measured the execution time of the kernel with nanosecond precision using the real-time clock available in the Kasli controller (i.e. the core device). An execution time measurement starts when the kernel starts execution, after the kernel binary is compiled and uploaded to the core device, and stops when the kernel finishes execution. Any RPCs from the core device to the host are included in the execution time measurement. We will use the execution time measurements to calculate the overhead of the real-time software. The kernel binary size is measured on the host at the output of the ARTIQ compiler and is used to calculate any binary-size overhead caused by our software framework. All our measurements are performed with ARTIQ version 6.7659.c6a7b8a8 and the results are presented in Figure 5 and 6.

A. Execution time overhead

The results in Figure 5 show the execution time overhead of the kernel for each experiment using the old and new DAX-based control software. For each experiment, we calculate the minimal execution time t_{min} based on the pulse lengths, detection times, and intentional wait times of the experiment. Given the measured execution time of an experiment t_{exe}, the execution time overhead is defined as $(t_{exe} - t_{min})/t_{min}$. Figure 5 shows that the old control software has an execution time overhead between 52.4% and 62.6% for the two MW and the qubit init experiments. These experiments consist of relatively short and quick operations which increase the operation density and induce more strain on the RTIO subsystem. Any inter-sample execution overhead introduced by the real-time control software will quickly increase the total execution overhead. The tickle experiment consists of slower operations

resulting in a measured overhead of 8.6% for the old control software. Any software overhead will be less significant due to the longer total duration of the experiment. The Yb spec experiment has very slow operations and includes a 500 ms wait time for each sample. Any overhead introduced by the real-time control software will be negligible on the timescale of the experiment.

If we look at the results of the DAX-based control software (without buffering) in Figure 5, we see that the new control software significantly reduces the execution time overhead compared to the old control software. On average, the new control software reduces the execution time overhead by 63.3% compared to the old control software. This average overhead reduction also includes the Yb spec experiment which already has a negligible execution time overhead for the old control software. When not including the Yb spec experiment, the average execution time overhead reduction is 55.4%. When we include buffering, we see that experiments with short and quick operations benefit the most (i.e. the two MW and the qubit init experiments). Buffering reduces inter-sample overhead by scheduling multiple samples ahead before retrieving results. Hence, the experiments most affected by inter-sample overhead benefit the most from buffering. The execution time overhead of the tickle experiment is not further reduced by buffering. The new control software already reduced its overhead to 1.4%, and inter-sample overhead does not appear to be a significant part of that. Compared to the old control software, the DAX-based control software with buffering enabled reduces execution time overhead by 88.7% and 87.1% on average with and without the Yb spec experiment, respectively.

We further analyzed our measurements to understand why the DAX-based control software performs better than the old control software. We attribute the reduced overhead to two main sources: timing management and data offloading. As mentioned earlier, real-time control software often inserts some delays on the event timeline to compensate for device programming times. The new control software groups devices in modules which in turn provides functions to manipulate those devices. The inserted delays can be optimized for each function which reduces the overhead. The old control software is less structured which often leads to larger worst-case delays or redundant delays to be inserted. Modular and well-designed real-time software allows us to insert more fine-grained delays, which reduces the total execution time overhead. Modular software design also leads to code that is more flexible and robust to changes. When a module has any modifications to its real-time devices or their behavior, its function might need to be optimized again, but other modules and services are not affected by the change. If devices in a module completely change, a module might need to be redeveloped. Fortunately, if the function signatures of the new module are compatible with the old one, the modules could be swapped without affecting other parts of the system.

The second major contributor to overhead reduction is data offloading. Measurement data for an experiment is often offloaded to the host using asynchronous RPCs while the kernel is running. Such offloading can be very efficient and transfers parts of computational tasks from the kernel to the host while also reducing memory usage on the core device. The new control software uses portable DAX data-processing modules which are highly optimized to maximize the benefit of the data offloading. As a result, the complexity and execution time overhead of the kernel is reduced.

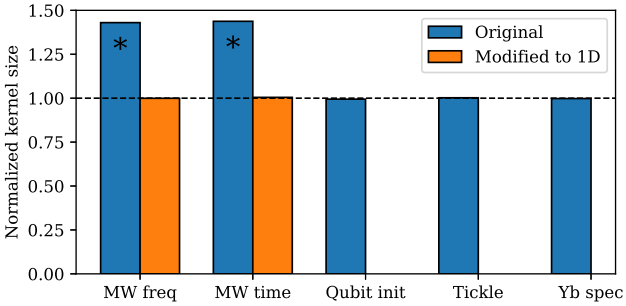

Fig. 6. Kernel binary size of the new control software normalized to the kernel binary size of the old control software.
* In the new control software, the two microwave (MW) scan experiments merged into a single 2D scan experiment causing an increased kernel size.

We would like to mention that better timing management and data offloading is also achievable with monolithic control software, but modular software makes it much easier. Devices will always be addressed through the functions of the module it is part of, making it easy to optimize the inserted delays for each scenario and improve timing management. For data offloading, the DAX data-processing module is portable between systems (see Section V) and only has to be developed and optimized once thanks to the modular software architecture.

B. Kernel binary size

The results in Figure 6 show the kernel binary size of the new control software normalized to the kernel size of the old control software. We see that for the two MW experiments the kernel size is increased by 43.0% to 43.8% while for the other experiments the kernel size changed less than 1%. While all experiments are 1D scanning-type experiments, the two MW experiments merged into a single two-dimensional (2D) scanning experiment in the new control software. For our tests, we configured one dimension to be static to reduce the experiment to a 1D scan. While this will result in a functional 1D scan, the DAX scanning infrastructure still stores data for the static dimension for each point in the scan. Hence, the kernel binary size increases. We manually modified the new MW experiment to a 1D scan for frequency and time storing the fixed value of the other dimension as a constant. When we measure the kernel binary size again, the difference with the old control software is less than 1%. From our results, we can conclude that the ARTIQ compiler works well with modular control software, and modular real-time software does not cause extra overhead that increases the kernel binary size.

V. Code portability

A principal benefit of modular control software is the potential for code portability between different quantum systems. The ARTIQ ecosystem already successfully abstracts real-time hardware with drivers and gateware to hide differences between hardware configurations or even hardware platforms. The DAX framework tries to achieve portability and abstraction on a higher level, more specifically the system-level and application-level software. On the system-level, generic DAX modules, services, and scanning infrastructure (see Section III-C) allow portability of real-time control code between systems. Portability for application-level software is achieved by using interfaces and clients.

To evaluate the amount of code portability between two different systems, we have implemented DAX-based control software for a second experimental trapped-ion quantum system known as the red chamber (RC) system [38] . The real-time control hardware of RC is based on a KC705 [39] evaluation board with custom breakout boards that contain digital I/O and DDS devices. The control software for the RC system consists of 20 modules and 7 services. Two services use portable DAX data-processing modules and one module extends the DAX module for safety-related functionality. The RC system code implements multiple DAX interfaces, including the data-context interface and two implementations of the operation interface. The system code controls 30 real-time devices and 1 external device. Notable is that the RC system has more modules than the STAQ system even though there are fewer real-time devices. The software of the RC system was developed after that of the STAQ system, and we learned it was better for modularity and portability to have a deeper system tree with more and smaller modules.

Figure 7 shows a subset of the RC modules and services relevant for the microwave operation service, which implements the DAX operation interface. The graph looks very similar to the one for the STAQ system shown in Figure 4 despite the real-time devices and controlled hardware being significantly different. The Yb171 and microwave modules are similar to their STAQ counterparts while the main differences are found in the CW and PMT modules. One key difference between the systems is that the detection laser shares an upstream master switch with other continuous wave lasers. Due to the master switch, it is impossible to control the detection laser independently from other lasers in the CW module without potential conflicts. Hence, the detection laser is controlled by the CW module and the PMTs are contained in an independent module. A detection subroutine now requires the CW and PMT module to work in parallel which is captured in the detection service. The remaining services in the RC system are similar to their STAQ equivalents. Figure 4 and 7 show that two systems with significantly different real-time control systems and devices can still have real-time control software with similar architectures. Modules and services can successfully abstract such differences.

To evaluate code portability between the STAQ and RC

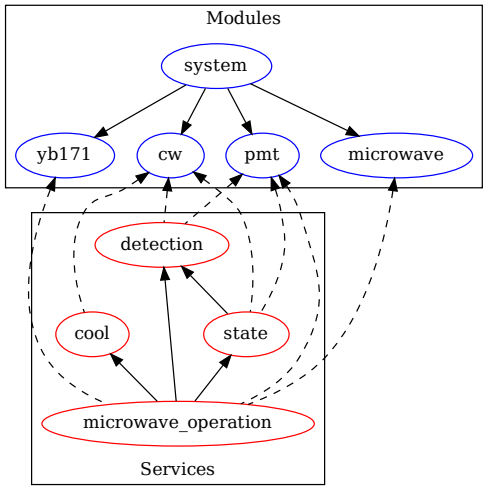

Fig. 7. Subset of the RC modules and services relevant for the microwave operation service.

system, we disabled modules and services not relevant for microwave operations. We then run a set of six experiments on each system. Three are MW calibration experiments and include a MW frequency calibration, a MW Ramsey frequency calibration, and a MW gate experiment that executes a sequence of X rotations to fine-tune the microwave Rabi gate time. These calibration experiments are hardware-specific and therefore have system-specific implementations. Two other experiments are the DAX clients for RB and GST that use the operation interface and the data-context interface. The last experiment is the portable SQST quantum program that is dynamically linked to the system using the program loader client. The mentioned clients and portable experiments are described in Section III-C.

All ARTIQ experiments have four execution phases: build, prepare, run, and analyze. The build phase is used to instantiate objects and process arguments. The prepare phase is the first moment where code directly relevant to the experiment can execute. This phase allows experiments to execute code on the host without accessing any devices or data storage. The run phase is the only moment where the experiment has access to devices and data storage. Kernels can only execute in the run phase of the experiment and any data analysis for calibration purposes should also be done here. Finally, the analysis phase is used for the post-experiment analysis of data. The run, prepare, and analyze phases are separated to pipeline experiments and maximize usage of the real-time control system. Our code portability evaluation focuses on the prepare and run phase, which are the two phases directly relevant to the functionality of the experiment. We decided not to add the build and analysis phase to not give ourselves a potentially unfair advantage by including more code in the coverage analysis.

For our code portability evaluation, we will run the six mentioned experiments on both systems while keeping track

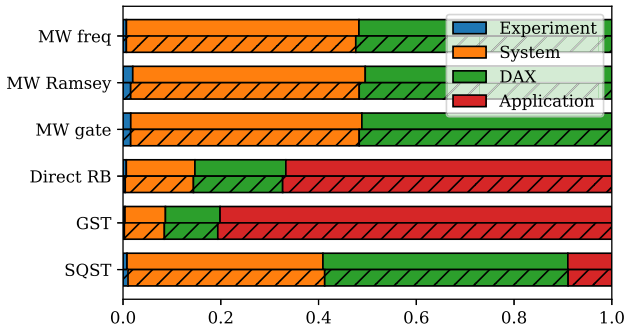

Fig. 8. Categorized and normalized proportions of covered statements for the STAQ (solid bars) and RC (hatched bars) control software.

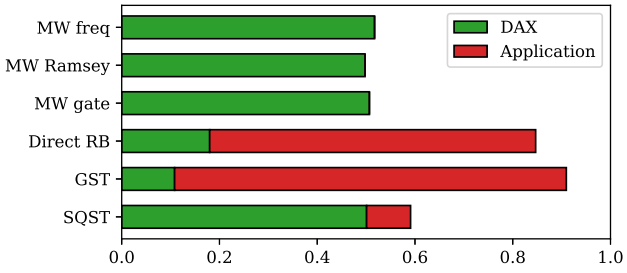

Fig. 9. Categorized and normalized proportions of covered statements from STAQ that are shared with RC.

of the statement coverage of the prepare and run phase for each experiment. Statement coverage is a technique often used for testing and keeps track of code statements evaluated at least once during program execution. The resulting data gives insight into the quantity of code that is used during program execution and does not provide information about the execution time spent for each statement. We measure coverage by simulating our kernel code using the DAX simulator [40] in conjunction with Coverage.py [41]. Our statement coverage data includes statements executed as part of host code, kernels, and RPCs. For our analysis, we are interested in the coverage of four categories of code: experiment code, system code, DAX library code, and *application code*. The first two categories are already defined in Section III and the third category is self-explanatory. We define application code as high-level and portable code that extends or utilizes the real-time control software to achieve its functionality. For the experiments we chose, application code includes the pyGSTi library and the SQST program. Coverage in other supporting libraries, such as ARTIQ or the standard library, is not included in this analysis. The coverage results are shown in Figure 8.

The results in Figure 8 show that for all experiments, the proportions of code in each category do not differ much between the STAQ and RC system. What can not be seen from the figure is that the total number of statements covered for each experiment does not differ more than 1.8% between the two systems. Figure 8 shows that the three MW calibration experiments have very similar results with 2.0% or less experiment code, between 46.8% and 47.6% system code, and 50.4% to 52.4% of DAX code. The covered statements of DAX library can mainly be found in its data-processing module, scanning infrastructure, and system initialization-related code. The Direct RB and GST experiments both have a large proportion of application code that covers parts of the pyGSTi library. These are procedures used to generate the benchmarking circuits. Both experiments also use pyGSTi for measurement data analysis, but those procedures are not included in our coverage data because they are part of the analysis phase of the experiment. For the remaining portion of covered statements for the Direct RB and GST experiments, more than half is DAX library code which includes the code of the client itself and the data processing module. Finally, the SQST program contains 9.0% and 8.9% application code, which is the portable SQST code itself, for the STAQ and RC system, respectively. The DAX library code mainly includes statements from the program loader client and the data-processing module in addition to the initialization code used by the loader to create and dynamically link the portable program to the system.

The results in Figure 8 show that with a modular software architecture, large portions of covered statements do not have to be system-specific and can be shared as application code or as part of a shared library for system code, such as DAX. For each unique quantum system, only the experiment code and system code would have to be developed which would significantly reduce the development time. For the code that does need to be developed, most of it is part of the system code which is shared between experiments for a single system and reduces development time even further.

Only covered statements in the DAX and application categories in Figure 8 are potentially portable between the STAQ and RC systems. We took the coverage data for each experiment and compared how many statements in the DAX and application categories were covered by both systems. These are the statements that are directly shared between the two systems. The results, which are normalized to the total number of covered statements for STAQ, are shown in Figure 9.

The results in Figure 9 show that even the system-specific MW calibration experiments consists of 49.8% to 51.7% of shared statements. Data-processing modules and scanning infrastructure add a significant number of covered statements during an experiment and are relatively easy to make portable. By sharing portable modules and services, we achieve portability on the system-code level. The Direct RB and GST experiments consist of 84.7% and 91.0% of shared statements, respectively. The application code is a major contributor to the proportion of shared statements but portable system code also contributes a significant part. Application code on the quantum operation level is inherently portable and we show that by introducing interfaces and clients, we can successfully connect to application-level software. Finally, the SQST program contains 59.1% of shared statements of which 9.0% is

application code. The SQST program is small and therefore does not contribute a lot of covered statements. The remaining shared statements all originate from the system code.

To further increase the amount of shared code between the STAQ and RC systems, we could generalize more modules and services. For example, the microwave module, the state service, and the microwave operation service of both systems contain very similar code and could probably be converted to a portable DAX module or service. So far, we have not done that in favor of flexibility and code simplicity. A module or service part of the system code can easily be modified for testing or mitigating device-related issues. Especially in an academic setting, flexibility can sometimes be more important than code portability. Additionally, portable modules and services are often required to have many configuration and customization capabilities to function correctly for different systems. Hence, portable code is often more complex which might not always be desired. Instead, we keep such modules and services part of the system code and manually "port" them to new systems. Overall development time can still be reduced by porting modules and services while full flexibility is preserved.

VI. Experiments

To demonstrate the capabilities of modular DAX-based control software and the portability of clients, we used the RB client presented in Section III-C to perform benchmarking on two experimental quantum systems. The RB client uses the operation interface which is implemented on both systems by their respective MW operation services. These services utilize a microwave horn to excite a dipole transition between the hyperfine states of Ytterbium 171 ($^{171}Yb^{+}$), which is where the qubit is encoded. This performs X and Y rotations on the Bloch Sphere. To perform cooling, state preparation, and measurement, the two systems use a 370nm laser to excite the dipole transition between $^2S_{1/2}$ and $^2P_{1/2}$ [42].

Before performing coherent operations, very accurate knowledge of the hyperfine frequency difference between the qubit states is needed, along with the Rabi frequency corresponding to oscillations between these states. The hyperfine splitting between the states is very well known [43]. However, a strong magnetic field is installed in our systems, slightly altering this value. Assuming we have some knowledge of what the frequency change should approximately be, three calibration experiments are still needed. The first experiment performs a sequence of timed microwave pulses near the qubit transition frequency to get Rabi oscillations. We then fit this data to get a rough estimate of the Rabi frequency. The second experiment uses Ramsey interferometry to fine-tune the qubit transition frequency. This experiment can be done with incrementally smaller frequency ranges to get greater precision of the resonance. Lastly, we perform increasingly longer sequences of π-pulse rotations designed to end with the qubit in the ground state in order to fine-tune the Rabi frequency.

After calibration, we perform Direct RB, with circuit lengths starting at 1 and scaling up exponentially to 1024. Direct RB is a modification of the original RB proposal which implements randomized circuits by sampling a system's native gates from a user-provided distribution Ω [33]. For microwave gates on ion trapping systems, the native gate sets are X and Y rotations, and we chose Ω to be uniform. The sequences are provided by the pyGSTi library [36] using the DAX RB client. For each circuit length, we performed 10 different circuits with 100 samples for each. The circuits were designed such that output was randomized to avoid skewed data because of bias toward a particular outcome. For example, the detection process in this experimental setup is designed such that the ground state is dark when shining the 370nm laser on the ion. Thus, losing the ion during an experiment would lead to always measuring the ground state.

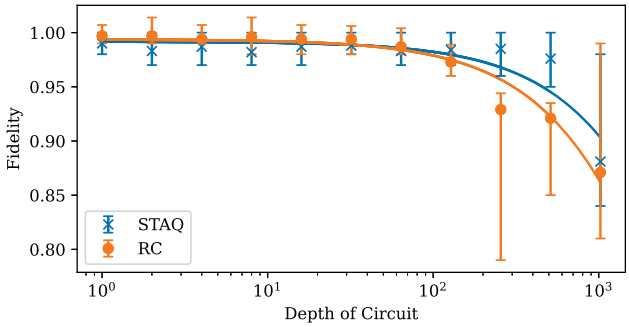

Fig. 10. Single-qubit Direct randomized benchmarking fidelity results for the STAQ and RC system using microwave gates. Error bars are calculated using the 10th to 90th percentile as boundaries. RC starts at higher fidelity due to better SPAM but decays quicker due to lower gate fidelity.

To demonstrate the flexibility of DAX and the portability of the RB client, we perform Direct RB with two different experimental setups: the STAQ and RC systems. Besides the different real-time control systems and devices, the main difference between these two setups is that STAQ is at cryogenic temperatures while RC is at room temperature. However, this shouldn't have any drastic effect on microwave operations and the data between the two systems should be quite comparable. The results from this experiment can be found in Figure 10. Here, the error per gate r is estimated to be $r = 4(1-p)/3 = 1.45 \times 10^{-4} \pm 2.58 \times 10^{-5}$ for the STAQ system and $r = 2.28 \times 10^{-4} \pm 1.94 \times 10^{-5}$ for the RC system, where p is calculated from fitting to the function $P(m) = 0.5 + Bp^m$. This number can also be interpreted as $1 - F_g$, where F_g is the average gate fidelity of the system. The difference in errors at low circuit depth is simply a result of different state preparation and measurement (SPAM) error, while the STAQ system can be seen to overtake RC at higher circuit depth due to better gate calibration.

VII. Conclusion

We have presented a systematic design strategy and a modular architecture for real-time quantum control software that organizes devices and system-wide functionality in modules and services, respectively. Our architecture supports the develop-

ment of modular control software and enhances the flexibility and portability of real-time control software. We implemented a software framework to develop real-time control software based on our proposed architecture, which is part of our open-source library Duke ARTIQ extensions (DAX). Our evaluation shows that modular control software can reduce the execution time overhead of kernels by 63.3% on average while not increasing the binary size. Software portability is achieved on the system level by introducing portable modules, services, and scanning control flow. We achieve application-level portability using interfaces and clients. Our analysis shows that modular control software for two distinctly different systems can share between 49.8% and 91.0% of covered code statements. Finally, we have shown that we can run a portable Direct randomized benchmarking (RB) experiment on two different ion-trap quantum systems that are fully controlled and calibrated by software based on our framework.

ACKNOWLEDGMENT

This work is funded by EPiQC, an NSF Expeditions in Computing (1832377), the Office of the Director of National Intelligence - Intelligence Advanced Research Projects Activity through an ArmyResearch Office contract (W911NF-16-1-0082), the NSF STAQ project (1818914), the U.S. Department of Energy (DOE), Office of Advanced Scientific Computing Research award DE-SC0019294, and DOE Basic Energy Sciences award DE-0019449.

REFERENCES

[1] K. M. Svore, A. Geller, M. Troyer, *et al.*, "Q#: Enabling scalable quantum computing and development with a high-level domain-specific language," *arXiv preprint arXiv:1803.00652*, 2018.

[2] A. W. Cross, A. Javadi-Abhari, T. Alexander, *et al.*, *Openqasm 3: A broader and deeper quantum assembly language*, 2021. arXiv: 2104.14722 [quant-ph].

[3] M. S. ANIS, H. Abraham, AduOffei, *et al.*, *Qiskit: An open-source framework for quantum computing*, 2021. DOI: 10.5281/zenodo.2573505.

[4] X. Fu, J. Yu, X. Su, *et al.*, "Quingo: A programming framework for heterogeneous quantum-classical computing with nisq features," *arXiv preprint arXiv:2009.01686*, 2020.

[5] D. S. Steiger, T. Häner, and M. Troyer, "ProjectQ: An open source software framework for quantum computing," *Quantum*, vol. 2, p. 49, Jan. 2018, ISSN: 2521-327X. DOI: 10.22331/q-2018-01-31-49. [Online]. Available: https://doi.org/10.22331/q-2018-01-31-49.

[6] C. Developers, *Cirq*, version v0.10.0, See full list of authors on Github: https://github.com/quantumlib/Cirq/graphs/contributors, Mar. 2021. DOI: 10.5281/zenodo.4586899. [Online]. Available: https://doi.org/10.5281/zenodo.4586899.

[7] F. Arute, K. Arya, R. Babbush, *et al.*, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, Oct. 2019, ISSN: 1476-4687. DOI: 10.1038/s41586-019-1666-5. [Online]. Available: https://doi.org/10.1038/s41586-019-1666-5.

[8] C. Ryan-Anderson, J. G. Bohnet, K. Lee, *et al.*, "Realization of real-time fault-tolerant quantum error correction," *Phys. Rev. X*, vol. 11, p. 041058, 4 Dec. 2021. DOI: 10.1103/PhysRevX.11.041058. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevX.11.041058.

[9] L. Postler, S. Heußen, I. Pogorelov, *et al.*, *Demonstration of fault-tolerant universal quantum gate operations*, 2021. DOI: 10.48550/ARXIV.2111.12654. [Online]. Available: https://arxiv.org/abs/2111.12654.

[10] Y. Wang, Y. Li, Z.-q. Yin, and B. Zeng, "16-qubit ibm universal quantum computer can be fully entangled," *npj Quantum information*, vol. 4, no. 1, pp. 1–6, 2018.

[11] I. Pogorelov, T. Feldker, C. D. Marciniak, *et al.*, "Compact ion-trap quantum computing demonstrator," *PRX Quantum*, vol. 2, p. 020343, 2 Jun. 2021. DOI: 10.1103/PRXQuantum.2.020343. [Online]. Available: https://link.aps.org/doi/10.1103/PRXQuantum.2.020343.

[12] R. Acharya, I. Aleiner, R. Allen, *et al.*, *Suppressing quantum errors by scaling a surface code logical qubit*, 2022. DOI: 10.48550/ARXIV.2207.06431. [Online]. Available: https://arxiv.org/abs/2207.06431.

[13] G. Pagano, A. Bapat, P. Becker, *et al.*, "A quantum approximate optimization algorithm in a trapped-ion quantum simulator," en, Oct. 2020. [Online]. Available: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=928237.

[14] J. Kim, T. Chen, J. Whitlow, *et al.*, "Hardware design of a trapped-ion quantum computer for software-tailored architecture for quantum co-design (staq) project," in *Quantum 2.0*, Optical Society of America, 2020, QM6A–2.

[15] M. Blok, V. Ramasesh, T. Schuster, *et al.*, "Quantum information scrambling in a superconducting qutrit processor," *arXiv preprint arXiv:2003.03307*, 2020.

[16] S. Bourdeauducq, R. Jördens, P. Zotov, *et al.*, *Artiq 1.0*, version 1.0, May 2016. DOI: 10.5281/zenodo.51303. [Online]. Available: https://doi.org/10.5281/zenodo.51303.

[17] V. Negnevitsky, "Feedback-stabilised quantum states in a mixed-species ion system," Ph.D. dissertation, ETH Zurich, 2018.

[18] P. Maunz, J. Mizrahi, and J. Goldberg, *Ioncontrol v. 1.0, version 00*, Jul. 2016. [Online]. Available: https://www.osti.gov/biblio/1326630.

[19] X. Fu, L. Riesebos, M. A. Rol, *et al.*, "Eqasm: An executable quantum instruction set architecture," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 224–237. DOI: 10.1109/HPCA.2019.00040.

[20] C. A. Ryan, B. R. Johnson, D. Ristè, B. Donovan, and T. A. Ohki, "Hardware for dynamic quantum computing," *Review of Scientific Instruments*, vol. 88, no. 10, p. 104 703, 2017.

[21] G. Kasprowicz, P. Kulik, M. Gaska, *et al.*, "Artiq and sinara: Open software and hardware stacks for quantum physics," in *OSA Quantum 2.0 Conference*, Optical Society of America, 2020, QTu8B.14. DOI: 10.1364/QUANTUM.2020.QTu8B.14. [Online]. Available: http://www.osapublishing.org/abstract.cfm?URI=QUANTUM-2020-QTu8B.14.

[22] J. H. Nielsen, M. Astafev, W. H. Nielsen, *et al.*, *Qcodes/qcodes: V0.30.0.dev0*, version v0.30.0.dev0, Oct. 2021. DOI: 10.5281/zenodo.5595929. [Online]. Available: https://doi.org/10.5281/zenodo.5595929.

[23] M. Rol, C. Dickel, S.Asaad, *et al.*, *Pycqed_py3*, version v0.2, Dec. 2019. DOI: 10.5281/zenodo.3574563. [Online]. Available: https://doi.org/10.5281/zenodo.3574563.

[24] D. C. McKay, T. Alexander, L. Bello, *et al.*, *Qiskit backend specifications for openqasm and openpulse experiments*, 2018. arXiv: 1809.03452 [quant-ph].

[25] L. Riesebos, X. Fu, A. Moueddenne, *et al.*, "Quantum accelerated computer architectures," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019, pp. 1–4. DOI: 10.1109/ISCAS.2019.8702488.

[26] T. Nguyen, A. Santana, T. Kharazi, D. Claudino, H. Finkel, and A. McCaskey, "Extending c++ for heterogeneous quantum-classical computing," *arXiv preprint arXiv:2010.03935*, 2020.

[27] R. S. Smith, M. J. Curtis, and W. J. Zeng, "A practical quantum instruction set architecture," *arXiv preprint arXiv:1608.03355*, 2016.

[28] F. T. Chong, D. Franklin, and M. Martonosi, "Programming languages and compiler design for realistic quantum hardware," *Nature*, vol. 549, no. 7671, pp. 180–187, 2017.

[29] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.

[30] X. Fu, M. A. Rol, C. C. Bultink, *et al.*, "An experimental microarchitecture for a superconducting quantum processor," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17, Cambridge, Massachusetts: Association for Computing Machinery, 2017, pp. 813–825, ISBN: 9781450349529. DOI: 10.1145/3123939.3123952. [Online]. Available: https://doi.org/10.1145/3123939.3123952.

[31] L. Riesebos, B. Bondurant, and K. R. Brown, *Duke artiq extensions (dax)*, 2021. [Online]. Available: https://gitlab.com/duke-artiq/dax.

[32] E. Magesan, J. M. Gambetta, and J. Emerson, "Scalable and robust randomized benchmarking of quantum processes," *Physical review letters*, vol. 106, no. 18, p. 180 504, 2011.

[33] 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. 030 503, 3 Jul. 2019. DOI: 10.1103/PhysRevLett.123.030503. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.123.030503.

[34] J. M. Epstein, A. W. Cross, E. Magesan, and J. M. Gambetta, "Investigating the limits of randomized benchmarking protocols," *Physical Review A*, vol. 89, no. 6, p. 062 321, 2014.

[35] 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. DOI: 10.48550/ARXIV.1310.4492. [Online]. Available: https://arxiv.org/abs/1310.4492.

[36] Erik, L. Saldyt, Rob, *et al.*, *Pygstio/pygsti: Version 0.9.10*, version v0.9.10, Oct. 2021. DOI: 10.5281/zenodo.5546759. [Online]. Available: https://doi.org/10.5281/zenodo.5546759.

[37] R. Schmied, "Quantum state tomography of a single qubit: Comparison of methods," *Journal of Modern Optics*, vol. 63, no. 18, pp. 1744–1758, 2016.

[38] Y. Wang, S. Crain, C. Fang, *et al.*, "High-fidelity two-qubit gates using a microelectromechanical-system-based beam steering system for individual qubit addressing," *Phys. Rev. Lett.*, vol. 125, p. 150 505, 15 Oct. 2020. DOI: 10.1103/PhysRevLett.125.150505. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.125.150505.

[39] *Xilinx kc705*. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html.

[40] L. Riesebos and K. R. Brown, "Functional simulation of real-time quantum control software," in *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2022.

[41] *Coverage.py*. [Online]. Available: https://github.com/nedbat/coveragepy.

[42] S. Olmschenk, K. C. Younge, D. L. Moehring, D. N. Matsukevich, P. Maunz, and C. Monroe, "Manipulation and detection of a trapped yb+ hyperfine qubit," *Physical Review A*, vol. 76, no. 5, p. 052 314, 2007.

[43] P. T. Fisk, M. J. Sellars, M. A. Lawn, and G. Coles, "Accurate measurement of the 12.6 ghz" clock" transition in trapped/sup 171/yb/sup+/ions," *IEEE transactions on ultrasonics, ferroelectrics, and frequency control*, vol. 44, no. 2, pp. 344–354, 1997.