Universal Graph-Based Scheduling for Quantum Systems

Leon Riesebos, Brad Bondurant, and Kenneth R. Brown, *Duke University, Durham, NC, 27708, USA*

High fidelity operation of a quantum system requires precise tuning of control parameters. Calibration of a quantum system is often achieved by running complex series of dependent experiments and a full system calibration can require tens of calibration experiments to complete. Optimal control parameters drift over time, and components of experimental quantum systems are susceptible to failure. Hence, continuous operation of a quantum system requires automated background processes such as frequent recalibration and monitoring. In this article, we present a scheduling toolkit that schedules experiments based on a directed acyclic graph using a configurable traversal algorithm. Our scheduler can be triggered from any process, enabling universal feedback between the scheduler and the quantum control system. To demonstrate the capabilities of our system, we implemented a complex system calibration algorithm based on our scheduling toolkit.

Operations on quantum systems are realized by applying analog signals to various components of the system containing the quantum bits (qubits). To perform useful computations with such systems, it is critical that analog signals are precisely tuned to ensure high fidelity operations on the qubits.[1–3] The calibration of all different operations often consists of running complex series of dependent experiments in a specific order. Full system calibration can require over 40 calibration experiments and take up to tens of hours to complete (see supplementary information in Arute *et al.*[3]), and the complexity of this time-consuming process will further increase when scaling the number of qubits in the system. Due to the analog nature of the operations, optimal control parameters drift over time, requiring continuous monitoring and frequent recalibration of the system. To keep quantum systems continuously operational, we need automated background processes that can be simple periodic tasks such as system monitoring as well as complex sets of dependent experiments for system calibration.

In this article, we present an open-source scheduling toolkit that can manage generic background processes for quantum systems:

1) Our scheduling toolkit schedules experiments based on a directed acyclic graph (DAG) using a configurable traversal algorithm.
2) We introduce triggers to support universal feedback for our scheduler.
3) We implement a complex system calibration algorithm based on our scheduling infrastructure to demonstrate the capabilities of our system.

Our article is organized as follows. We first introduce the execution model of our system in the "Execution Model" section. In "Scheduling Graph," we present the scheduling graph and the traversal algorithm that is at the core of our scheduling toolkit. The implementation of the Optimus scheduling algorithm for system calibration is outlined in the "Optimus Scheduling Algorithm" section. The implementation of our scheduling toolkit is discussed in "Implementation," and our simulation results are presented in "Results." We conclude our paper in "Conclusion."

EXECUTION MODEL

Our goal is to make a practical scheduling toolkit and therefore we need to understand the execution model of our quantum system. We envision the quantum system to work according to the *accelerator model* as described by Riesebos *et al.*[4–8] In such a model, quantum programs are considered hybrid programs that consist of a classical host program combined with one or more quantum kernels that can be mapped to a

0272-1732 © 2021 IEEE
Digital Object Identifier 10.1109/MM.2021.3094968
Date of publication 7 July 2021; date of current version 14 September 2021.

TABLE 1. Job specification and state.

Name	Type	Description
Experiment	Spec	The experiment to submit
Arguments	Spec	Experiment arguments
Interval	Spec	Submit interval time (optional)
Dependencies	Spec	Set of nodes (optional)
Last submit time	State	The last submit time

quantum coprocessor. Note that experiments or calibrations are quantum programs in the context of the accelerator model and the terms can be used interchangeably. An analysis of the accelerator model itself is outside the scope of this article and will not be further discussed.

IN THIS ARTICLE, WE PRESENT AN OPEN-SOURCE SCHEDULING TOOLKIT THAT CAN MANAGE GENERIC BACKGROUND PROCESSES FOR QUANTUM SYSTEMS.

We aim to run the scheduling algorithm as a classical host-only process that submits experiments asynchronously to a *pipeline*. Experiments submitted to the pipeline are sequentially executed by a separate process, which allows them to have exclusive access over the quantum coprocessor. The pipeline functions as a priority queue where higher priority experiments run first and experiments with equal priority run in the order in which they were submitted.

SCHEDULING GRAPH

The calibration of a quantum system consists of a set of dependent calibration experiments. Consider a universal gate set consisting of H, T, and CNOT on n qubits that are fully connected. There are $n^2 + n$ gates that need to be tuned up. Experimental quantum information scientists do not first directly tune these gates, but instead tune the underlying electromagnetic signals that drive these gates. Additionally, the gates themselves cannot be fully calibrated in isolation, and benchmarking experiments are needed to check that the gates perform well together. A common structure is to first carefully calibrate the radio frequency signals, then optimize individual gates, and finally test with a high-level calibration. Such a set of dependent *jobs*, where jobs represent calibration experiments or other periodic tasks, can be represented by a graph where the jobs are nodes and their dependencies are directed edges. Based on such a graph, which we will call a *scheduling graph*, it is possible to derive an execution schedule for the jobs. Circular dependencies are not allowed because they lead to impossible schedules. Hence, the scheduling graph is a DAG. DAGs have been used previously to schedule single-qubit calibrations[9]—here we extend that to multi-qubit and global calibrations as well as any other periodic tasks. A scheduling graph does not need to be weakly connected and can therefore contain multiple components. To schedule jobs, we will traverse over the scheduling graph and submit the experiments represented by the jobs to the pipeline. In the remainder of this section, we will lay out the components of the scheduling graph and the traversal algorithm.

TABLE 2. Node action enumeration.

Action	Description
PASS	Pass node
RUN	Run node
FORCE	Force to run node

Jobs

A job is a node in the scheduling graph that represents a specific experiment with a set of fixed arguments. Additionally, a job specification optionally contains a submit interval time and a set of dependencies. Every node in the graph has its persistent state and stores the last time it was submitted. A list of job specs and states is shown in Table 1.

It is possible to perform a few functions on jobs. If a job is submitted, its specified experiment will be submitted to the pipeline, the last submit time of the job state will be updated, and the function returns. Our scheduling process *visits* a node while traversing the graph, which will return a *node action*. A node action is an enumeration and if a node has expired (i.e., the current time minus the last submit time is greater than the submit interval), visiting that node will return node action RUN. If a node is not expired or has no submit interval, node action PASS will be returned. At the start of the graph traversal, it can be useful to force a run irrespective of the node state. This third node action FORCE will be discussed in the "Wave Algorithm" section. All node actions are listed in Table 2.

Wave Algorithm

The *wave algorithm* is a recursive algorithm used to traverse over the scheduling graph depth-first. The

TABLE 3. Maps of the scheduling policies. Entries in parenthesis are not reachable and therefore undefined, but included for completeness.

Previous, current action	LAZY policy	GREEDY policy
PASS, PASS	PASS	PASS
PASS, RUN	RUN	RUN
(PASS, FORCE)	-	-
RUN, PASS	PASS	RUN
RUN, RUN	RUN	RUN
(RUN, FORCE)	-	-
FORCE, PASS	RUN	RUN
FORCE, RUN	RUN	RUN
(FORCE, FORCE)	-	-

goal is to have a simple yet highly configurable algorithm that can be used to traverse over parts of the scheduling graph. Regardless of its configuration, the wave algorithm will always submit visited nodes, if any, in an order that satisfies the dependencies of the scheduling graph.

The wave algorithm uses node actions and a *scheduling policy* to determine if nodes need to be submitted or not. A scheduling policy is a map from two node actions to a single node action. During the graph traversal, the scheduling policy determines how the state of a node influences its dependents. We define two scheduling policies, LAZY and GREEDY. The LAZY scheduling policy will only submit expired and forced nodes while the GREEDY policy will additionally submit the dependencies of those nodes. The complete definitions of the scheduling policies are shown in Table 3.

Given a scheduling graph, the wave algorithm can be configured by providing a root node (or nodes), a root action, and a scheduling policy. The wave algorithm visits the current (root) node to obtain the current node action. Using the provided scheduling policy, the previous (root) action and current action are mapped to a new action. The wave algorithm recursively calls the dependents of the current node while passing the new action. When all dependents are visited, the current node will be submitted if the new action for the node equals RUN. The algorithm keeps a set of submitted nodes to make sure every node is not submitted more than once during a traversal. A single run of the wave algorithm is an atomic operation that will update the state of the nodes and submits zero or more experiments to the pipeline. The basic wave algorithm is shown in Listing 1.

```
def wave(node, prev_action,
         policy, submitted=None):
    """
    :param node: The current node
    :param prev_action: Previous node action
    :param policy: The scheduling policy
    :param submitted: Set of submitted nodes
    """
    if submitted is None:
        submitted = set()

    # Visit node to obtain current action
    curr_action = node.visit()
    # Compute new action based on policy
    new_action = policy(prev_action,
                        curr_action)

    # Recursion
    for n in node.dependents():
        submitted = wave(n, new_action,
                         policy, submitted)

    if new_action is NodeAction.RUN:
        if node not in submitted:
            # Submit node
            node.submit()
            submitted.add(node)

    return submitted
```

LISTING 1. The basic wave algorithm.

For a given scheduling graph, different wave configurations lead to different results based on the current state of the nodes. Figure 1 shows a simple scheduling graph with four nodes. Assuming job C is expired, the submitted jobs for various wave configurations are shown in Table 4. We can see that node action FORCE can be used as a root action to force one or more nodes to be submitted.

Configuration options for the wave algorithm not shown in Listing 1 are *depth*, *start depth*, and *priority*. The depth parameter is an integer that can limit the recursion depth of the wave algorithm. With the start depth parameter, it is possible to only visit and submit nodes beyond a given recursion depth. By default, nodes are visited and submitted starting from depth zero and the recursion depth is not limited. The priority parameter is passed to the pipeline as the experiment priority when

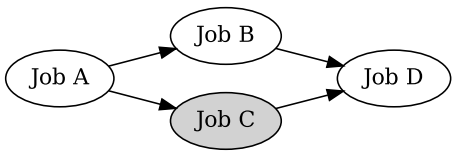

FIGURE 1. A simple scheduling graph with four jobs. For our example, we assume that job C is expired.

TABLE 4. Submitted jobs for different wave configurations for the scheduling graph shown in Figure 1 with job C expired.

Root node	Root action	Policy	Submitted jobs
Job A	PASS / RUN	LAZY	C
Job A	FORCE	LAZY	C, A
Job A	PASS	GREEDY	D, C
Job A	RUN / FORCE	GREEDY	D, C, B, A

submitting a job. A default priority can be configured in the scheduler, and if a trigger is submitted without the specification of a priority, the default will be used.

Scheduler Process

As mentioned in the "Execution Model" section, we envision running our scheduler as a host process. At startup, the scheduler is given a scheduling graph and a *trigger queue* will be created. Any process on the host can submit triggers to this queue through the scheduler process where a trigger is defined as a tuple containing a list of root nodes, a root action, and a scheduling policy. The scheduler process waits for a trigger to be submitted after which it will read the request and run the wave algorithm using the parameters given by the trigger. Note that a running experiment is also a host process that can trigger the scheduler and an experiment could for example request a full system calibration on demand by triggering the scheduler.

When time passes, nodes will expire, and to detect such events, the scheduler will periodically submit preconfigured triggers to its queue. The desired system for background tasks is realized by these periodic triggers.

Universal Feedback

Some scheduling scenarios cannot be conveniently expressed by a dependence graph and require universal classical logic to determine how the scheduler should proceed. For example, based on the results of job J, subgraph G_A or G_B needs to be evaluated by the scheduler. Such scenarios can still be part of our scheduling infrastructure by using the schedulers' trigger feature to feedback a decision to the scheduler. We can create an experiment for job J, which decides based on its results what subgraph needs to be evaluated next. Once the decision is made, the experiment submits a trigger to the scheduler with the root nodes of the subgraph that needs to be evaluated. When the scheduler runs job J it will receive feedback from the experiment through a trigger and the scheduler will proceed by evaluating the appropriate subgraph.

OPTIMUS SCHEDULING ALGORITHM

As mentioned earlier, a use case of particular interest for the scheduler is that of automated system calibration. Kelly *et al.*[9] proposed the "Optimus" algorithm for the intelligent calibration of qubit systems. Much like the approach we have outlined so far, they use a DAG to represent the calibration experiments and their dependencies. In this section, we will explain how the Optimus algorithm can be implemented using our scheduling infrastructure.

Algorithm Summary

The goal of the Optimus algorithm is to maintain accurate system control parameters while spending as little time as possible on calibrations. They achieve this goal by introducing three levels of interaction with each calibration in the graph: `check_state`, `check_data`, and `calibrate`. `check_state` relies solely on the specification of a timeout period (roughly corresponding to the drift timescale of the parameter in question) and previous system knowledge—namely the last time the parameter in question was calibrated or verified by `check_data` and the last time any dependencies were calibrated. Based on that knowledge, `check_state` will either report a pass or a failure. `check_data` is intended to be a minimal experiment to determine whether a parameter value is still valid—referred to as *in-spec* or *out-of-spec*. If `check_state` reports a failure, the `check_data` experiment will run and report either in-spec, out-of-spec, or *bad-data* (explained in further detail below).

Each level of interaction is increasingly time-consuming and is designed to execute only if the previous level fails. If `check_data` reports out-of-spec, then `calibrate` will run the full calibration and update the value of the parameter in question. The primary graph traversal routine used by Optimus (dubbed `maintain` by the authors) is a simple greedy, depth-first traversal. While the calibration interactions are executed lazily, to `maintain` traversal is greedy in the sense that each calibration job must be submitted in order for `check_state` to run. There is also a secondary traversal routine called `diagnose`, which only runs in the special case that `check_data` reports bad data. In that case, the assumption is that some part of our knowledge of the system is inaccurate. Thus, `diagnose` triggers a traversal of the subgraph containing the immediate dependencies of the calibration in question, in which `check_state` is skipped and each calibration is forced to run (at least)

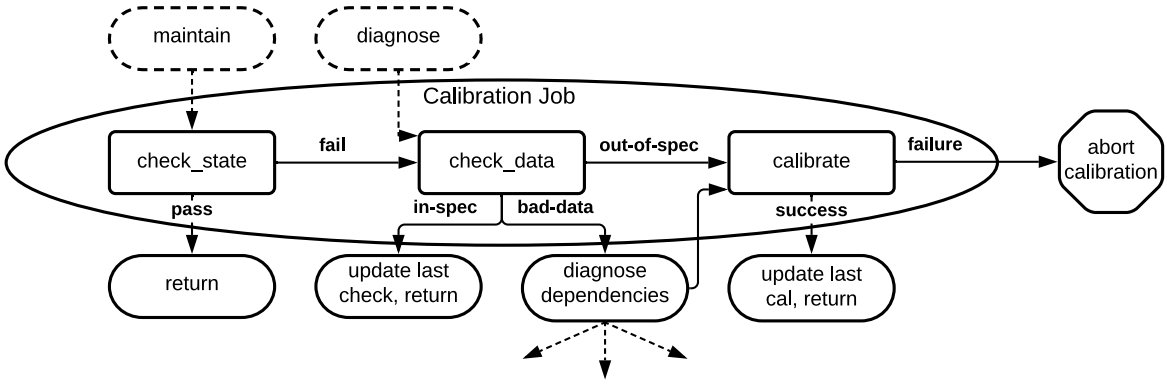

FIGURE 2. A state machine representation of the behavior of a single calibration in our implementation of the Optimus algorithm. The dashed lines represent entry and exit points for the different triggers. Returning from a calibration continues the current wave.

`check_data`. If any of the dependencies in question also report bad data, then another `diagnose` subgraph traversal will be recursively triggered on that node. The execution model for a single calibration is illustrated in Figure 2.

Algorithm Implementation

To integrate the Optimus algorithm into our scheduling toolkit, we introduce a new type of job called a *calibration job*. The implementation of the calibration job builds upon the scheduling infrastructure presented in the "Scheduling Graph" section and adds the traversal logic specific to the Optimus algorithm. While a job represents a single experiment to be run, a calibration job represents two experiments—one for `check_data` and one for `calibrate`. The `check_state` logic is incorporated into the calibration job definition via the specification of a timeout. The full list of calibration job specs and states is shown in Table 5.

TABLE 5. Calibration job specification and state.

Name	Type	Description
Check experiment	Spec	`check_data` experiment
Cal. experiment	Spec	`calibrate` experiment
Check arguments	Spec	Check experiment arguments
Cal. arguments	Spec	Cal. experiment arguments
Cal. timeout	Spec	Calibration interval (optional)
Dependencies	Spec	Set of nodes (optional)
Last cal. time	State	The last successful cal.
Last check time	State	The last successful check

In our framework, the `maintain` traversal is initiated by a periodic trigger and the `diagnose` traversals are initiated by triggers submitted by any calibration jobs that report bad data from the `check_data` experiment. The root nodes of the `maintain` trigger are all calibrations with no incoming dependencies, and the trigger has a `GREEDY` scheduling policy with root action `FORCE` to ensure that the `check_state` logic is executed for every calibration. The `diagnose` triggers are `GREEDY` (root action `FORCE`), with the root node specified as the calibration job representing the current experiment, and a depth and start depth of one so that only the immediate dependents are submitted. Additionally, it has a priority that is one level higher than the priority of the current experiment to ensure that the diagnose wave takes precedence over previously submitted experiments.

In implementing the Optimus algorithm, we also had to address a question that was only partially addressed by Kelly *et al.*[9]: What do we want to do if the algorithm fails to successfully calibrate a parameter? Kelly *et al.*[9] mention that they raise a "`DiagnoseError`" in the case that a `diagnose` wave is triggered without resolution. We take a slightly different approach and introduce a generic `FailedCalibrationError`, which is to be raised whenever a calibration runs but fails to achieve the desired accuracy. Then, we execute the `diagnose` waves blindly, without checking whether or not any dependencies were recalibrated as a result. After the `diagnose` wave, we simply run the calibration and allow it to pass or fail independently of the result of `diagnose`. If a calibration does fail for any reason, we assume that external intervention is required and halt all execution indefinitely until it is manually resumed.

IMPLEMENTATION

Our scheduling toolkit is implemented in Python as part of our open-source library Duke ARTIQ extensions (DAX),[10] which is tightly integrated with the advanced real-time infrastructure for quantum physics (ARTIQ) open-source software and hardware ecosystem.[11,12] The ARTIQ control infrastructure provides real-time control of our quantum systems and is used by dozens of research groups with over 200 systems deployed worldwide. Our software can schedule any ARTIQ experiment and is therefore fully compatible with existing systems that use ARTIQ.

The DAX scheduling toolkit allows users to define jobs and calibration jobs as classes that inherit the DAX `Job` and `CalibrationJob` base classes, respectively. The job specifications shown in Tables 1 and 5 are provided as class variables. The Optimus algorithm is implemented as part of the `CalibrationJob` class and the implementation builds upon the implementation of the `Job` class. Users can define a scheduler by creating a class that inherits the `DaxScheduler` base class, which is the class that contains the implementation of the wave algorithm as defined in the "Wave Algorithm" section. The set of nodes in the scheduling graph and the specifications for the periodic triggers as described in the "Scheduler Process" section are provided as class variables of the user-defined scheduler class. Additionally, a user-defined scheduler class inherits the ARTIQ `Experiment` class, which marks it as an executable experiment for the ARTIQ runtime. Once the user-defined scheduler is started, the scheduling graph will be constructed based on the specified nodes, a timer will be set for periodic triggers, and the process will start a Python asyncio TCP server to receive incoming trigger requests. When the running scheduler receives a trigger, the scheduler will run the wave algorithm using the parameters given by the trigger.

ARTIQ includes a management system, the ARTIQ master, that queues experiments, handles data storage, and supervises the quantum coprocessor. The pipeline for experiments, as described in the "Execution Model" section, is already implemented by the ARTIQ master, and experiments submitted to the pipeline by a scheduler will be executed by the ARTIQ master process. The ARTIQ master manages a centralized database, which we use to store the state of the nodes. The same database also stores system configuration and calibration results that need to be communicated from one experiment to the next. The part of our DAX library dedicated to system organization, real-time data processing, and data organization will be covered in an upcoming paper.

All our scheduling code is tested thoroughly using static test cases as well as random testing. We tested our implementation of the Optimus algorithm by generating random DAGs containing calibration jobs whose interaction methods return random results. The DAGs are generated by randomly populating the upper triangle of an n by n adjacency matrix (where n is the number of nodes in the graph) with ones with some probability p, resulting in a random partial order over the set of nodes from which a DAG can be derived. We run our implementation over a single call to `maintain` (including any subcalls to `diagnose`), and verify that our traversal matches the specification by Kelly *et al.*[9] Due to the existence of multiple valid traversals for a single call to `maintain`, we developed a recursive matching algorithm to determine if a traversal is valid given the graph structure and the calibration states/results.

RESULTS

To benchmark the efficiency of the Optimus algorithm compared to a naive full calibration, we chose to simulate our implementation over a variety of conditions. For our simulations, we choose a fixed number of nodes and run our implementation of the `maintain` traversal (and any `diagnose` sub-traversals) over randomly generated DAGs. To reflect realistic conditions, there must be a cause–effect relationship for a `check_data` experiment to return bad data. The purpose of the bad-data condition is to indicate that there are dependencies that are out-of-spec but did not time out. In other words, bad-data indicates an unresolved out-of-spec dependence. Hence, in the initial graph construction, we completely randomize the results of `check_state` (whether or not a node has timed out), but we only randomize the results of `check_data` to select between in-spec and out-of-spec. At runtime, if a node reaches the `check_data` stage and any of its dependencies (or subdependencies) remain out-of-spec, the node will return bad data. Additionally, in order to create a fair comparison between the Optimus algorithm and a naive full calibration of the system, we must guarantee that the system is fully calibrated by the end of the `maintain` wave. To achieve that, we force any root nodes to always time out, so that any unresolved out-of-spec nodes will be properly diagnosed and calibrated via the above bad-data logic.

The parameters for our simulation are as follows:

1) the probability that a node is out-of-spec in our initial graph construction;

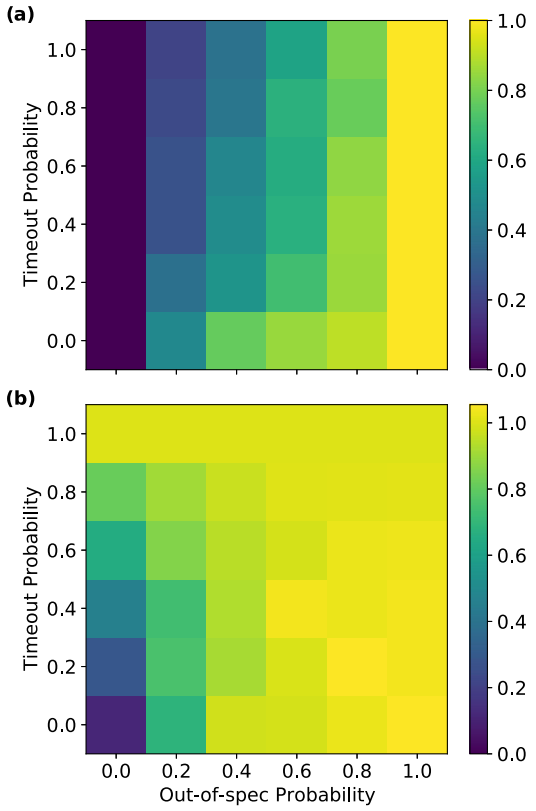

FIGURE 3. (a) Proportion of calibrations performed and (b) proportion of check experiments performed versus timeout and out-of-spec probability.

2) the probability that a node has timed out (excluding the forced timeouts on the root nodes).

We sweep each of these probabilities from 0 to 1 in intervals of 0.2, conducting 20 iterations at each point, each over a different random DAG. We choose the number of nodes to be $n = 20$ as a middle ground between the number calibrations required for ion-trap systems in our lab ($n \approx 10$) and current superconducting systems ($n \approx 40$).[3] The DAG is generated via the method described in the "Implementation" section, with $p = 0.5$, resulting in a moderately connected graph.

The results of our simulation can be found in Figure 3. Figure 3(a) shows the total number of calibrations performed normalized to the number of nodes versus the timeout and out-of-spec probability. We normalize the number of nodes in the graph to represent the relative efficiency of the Optimus algorithm compared to a full calibration. We can see that for out-of-spec probability 0 and 1 the proportion of calibrations performed is 0 and 1, respectively, as expected. For other out-of-spec probabilities, we see that the number of performed calibrations reduces when the timeout probability increases. This result is explained by the behavior resulting from a bad-data return value in the `check_data` stage. A higher likelihood of timeouts makes it more likely that any out-of-spec nodes are detected at or near the source. Conversely, a lower timeout probability means out-of-spec nodes are more likely to go unresolved, increasing the likelihood of bad-data results (and subsequent calibrations) in the parent nodes. In the worst case scenario, the out-of-spec node goes undetected until the root nodes of the graph are reached, triggering a cascade of bad-data results and calibrations until the out-of-spec node is reached and resolved.

In Figure 3(b), we see a graph with the number of check experiments performed relative to the number of nodes. The relative number of executed check experiments increases when the timeout probability or the out-of-spec probability increases. More timed-out experiments result in more check experiments to run while an increased number of out-of-spec nodes will cause more bad-data conditions, triggering more check experiments to run. It is possible for a calibration to be checked more than once throughout a traversal, in the case that it times out, gets checked, and then a parent calibration triggers a `diagnose` traversal due to an unresolved out-of-spec on another node. The result is that in some cases—particularly when the out-of-spec probability is high and the timeout probability is low—the total number of checks can exceed the number of nodes in the graph.

OUR SCHEDULER IS DRIVEN BY A CONFIGURABLE WAVE ALGORITHM AND IS CAPABLE OF RUNNING SIMPLE PERIODIC TASKS AS WELL AS COMPLEX SYSTEM CALIBRATION ROUTINES AS BACKGROUND PROCESSES.

To understand the overall efficiency of the Optimus algorithm, we need to take into account both the number of calibrations as well as the number of check experiments that are executed. As described in the "Algorithm Summary" section, the check experiment is a quick experiment to check if

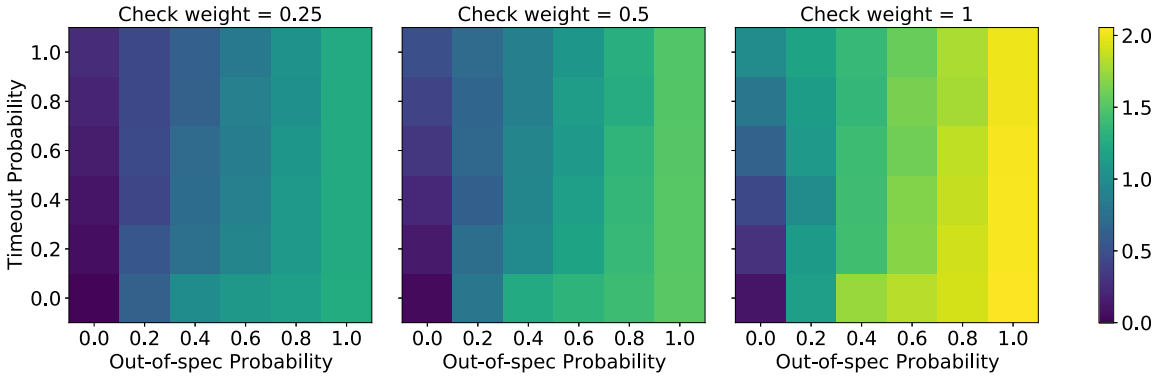

FIGURE 4. Relative efficiency of the full Optimus algorithm (checks and calibrations) with the cost of the check experiments weighted relative to the calibrations.

a parameter is still calibrated, which normally takes significantly less time compared to a full calibration of the same parameter. We have estimated the overall efficiency of the Optimus algorithm for different weights of the check experiments compared to the calibration experiments and the results are shown in Figure 4. The weights represent how long the check experiment takes relative to the calibration itself. While lower cost check experiments are ideal, we can see in Figure 4 that the Optimus algorithm can be beneficial even for more costly checks, depending on the out-of-spec probability. Specifically, the Optimus algorithm is more efficient than a naive full calibration for out-of-spec probabilities ≤ 0.4 and ≤ 0.6 with check weights 0.5 and 0.25, respectively. In the case that the check experiment takes the same amount of time as the calibration (i.e., check weight 1), Figure 4 shows that the Optimus algorithm is essentially always detrimental to performance, as expected.

WE HAVE PRESENTED AN OPEN-SOURCE SCHEDULING TOOL-KIT THAT SCHEDULES JOBS BASED ON A DIRECTED ACYCLIC GRAPH. OUR SCHEDULER IS DRIVEN BY A CONFIGURABLE WAVE ALGORITHM AND IS CAPABLE OF RUNNING SIMPLE PERIODIC TASKS AS WELL AS COMPLEX SYSTEM CALIBRATION ROUTINES AS BACKGROUND PROCESSES.

In terms of the limitations of our simulations, Figure 4 likely represents a lower bound of the efficiency of the Optimus algorithm. In a more realistic scenario, the timeouts would reflect the relative stability of the calibrations (i.e., how often a calibration goes out-of-spec), rather than the two results being drawn from independent probability distributions. Careful selection of timeouts would result in fewer bad-data results, and thus fewer unnecessary calibrations. Hence, we expect the Optimus algorithm to perform better in scenarios where timeouts are chosen carefully.

CONCLUSION

We have presented an open-source scheduling toolkit that schedules jobs based on a directed acyclic graph. Our scheduler is driven by a configurable wave algorithm and is capable of running simple periodic tasks as well as complex system calibration routines as background processes. We enable universal feedback between the scheduler and any running experiment or process by using triggers that will start new traversals on the scheduling graph. To demonstrate the capabilities of our toolkit, we have implemented the Optimus system calibration algorithm using our scheduling infrastructure and used our implementation to benchmark the efficiency of the algorithm.

ACKNOWLEDGMENTS

This work was supported by EPiQC, a National Science Foundation (NSF) Expeditions in Computing (1832377), the Office of the Director of National Intelligence—Intelligence Advanced Research Projects Activity through an Army Research Office contract (W911NF-16-1-0082), the NSF STAQ project (1818914), and the U.S. Department of Energy, Office of Advanced Scientific Computing Research QSCOUT Program.

REFERENCES

1. V. Schäfer *et al.*, "Fast quantum logic gates with trapped-ion qubits," *Nature*, vol. 555, no. 7694, pp. 75–78, 2018.
2. N. Wittler *et al.*, "An integrated tool-set for control, calibration and characterization of quantum devices applied to superconducting qubits," 2020. [Online]. Available: https://journals.aps.org/prapplied/abstract/10.1103/PhysRevApplied.15.034080
3. F. Arute *et al.* "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, 2019.
4. L. Riesebos *et al.*, "Quantum accelerated computer architectures," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2019, pp. 1–4.
5. X. Fu *et al.* "eqasm: An executable quantum instruction set architecture," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 224–237.
6. K. M. Svore *et al.*, "Q#: Enabling scalable quantum computing and development with a high-level domain-specific language," 2018, doi: 10.1145/3183895.3183901.
7. R. S. Smith, M. J. Curtis, and W. J. Zeng, "A practical quantum instruction set architecture," 2016, *arXiv:1608.03355v2*.
8. 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.
9. J. Kelly, P. O'Malley, M. Neeley, H. Neven, and J. M. Martinis, "Physical qubit calibration on a directed acyclic graph," 2018, *arXiv:1803.03226*.
10. L. Riesebos, B. Bondurant, and K. R. Brown, "Duke artiq extensions (Dax)." 2021. [Online]. Available: https://gitlab.com/duke-artiq/dax, 2021.
11. S. Bourdeauducq *et al.*, "Artiq 1.0," May 2016. [Online]. Available: https://github.com/m-labs/ARTIQ
12. G. Kasprowicz *et al.*, "Artiq and Sinara: Open software and hardware stacks for quantum physics," in *Proc. OSA Quantum 2.0 Conf.*,2020, Paper QTu8B.14.

LEON RIESEBOS is currently working toward a Ph.D. degree with the Department of Electrical and Computer Engineering, Duke University, Durham, NC, USA. His research interests include quantum computer architectures, control software, and full-stack system architectures. Riesebos received an M.Sc. degree in embedded systems from Delft University of Technology in 2016. Contact him at leon.riesebos@duke.edu.

BRAD BONDURANT is currently working toward a Ph.D. degree with the Department of Electrical and Computer Engineering, Duke University, Durham, NC, USA. His research interests include the control and calibration of quantum hardware. Bondurant received a B.S. degree in electrical engineering, computer engineering, and physics from North Carolina State University, Raleigh, NC, USA, in 2018. Contact him at brad.bondurant@duke.edu.

KENNETH R. BROWN is a Professor of electrical and computer engineering with Duke University, Durham, NC, USA and the Director of the NSF Software Enabled Architectures for Quantum codesign project developing applications, software, and hardware for ion trap quantum computers. He is a Senior Member of IEEE. Contact him at kenneth.r.brown@duke.edu.