1. Summary

In this paper we introduce Rasqal, a low-level symbolic analysis runtime used for quantum-classical hybrid computation. Its primary function is being able to use contextual clues and analysis from quantum and classical interwoven code to aggressively optimise any circuits sent to the QPU.

2. Background

With more quantum-classical hybrid algorithms being developed there was always a question about how to represent the intermingling of both concepts. Existing mid-level languages such as QASM [x] focused on gates and the specific operations that quantum computers had the capabilities to process directly, such as arithmetic and conditionals.

With QIR [4] gaining traction, and tools that emit it from popular high-level languages such as as Cuda Quantum [1], Catalyst [2] and Q# [3], hybrid algorithms can now be represented in a way which fully encodes their classical subtleties alongside quantum operations and features. This provides fascinating opportunities to explore the role of what a quantum runtime could look like, taking inspiration from existing ones such as the CLR [x] and JVM [x].

In this paper we're going to introduce a quantum-classical runtime that pulls inspiration from static analysis techniques, SAT solvers and interpreters to execute hybrid IR. This runtime, called Rasqal [x], can be found on Github.

As for the structure of the following paper, first we'll cover some of the key concepts that allow Rasqal to function, along with some of the transformations/optimizations this allows. Following that is an in-depth description of the data-structures that power execution, its circuit synthesis and deferred execution schemes.

x. Concepts

While dynamically executing hybrid IRs allow for many opportunities for experimentation, we're going to cover the three main considerations that were meant to power Rasqal as a runtime:

1. Dynamically analysing classical branching logic, such as if statements and loops, then lowering them into FPGA instructions to run directly on the QPU.
2. Embedding all inputs into the IR, performing aggressive constant folding and then using these values to optimise synthesised circuits. This includes folding quantum results into variables as the IR is executed.
3. Being able to perform complex hoisting/lowering operations because it has a full view of the entire code of the algorithm being run. Not just fragments of quantum or classical.

To help give some foundation for the ideas being talked about, Rasqal shares many traits with both symbolic execution engines [x] and interpreters [x]. It executes the IR itself like an interpreter, but analyses that information to build and then execute quantum circuits, similar

to a symbolic executor. Where-as symbolic execution engines are used for satisfiability, we use the constraints and data to simplify, optimise and build our circuits.

With this in mind we'll now run through the three concepts above in sequence, using pseudo-IR to help visually showcase the transformation.

xx. Dynamic Lowering

Lowing is informally defined as taking an abstract instruction and 'lowering' it into a less abstract one [x], usually into hardware-specific instructions. A trivial example for quantum computers is lowering gates to pulses and fundamental arithmetic operations into specific FPGA instructions.

This is commonly done two ways: statically or dynamically. Statically, it is done as a part of the compilation pass and is a fundamental part of building a binary for a particular platform. You know what CPU architecture you're targeting, so know what instructions to lower into. C++ and C are languages which do this.

Dynamic lowering is performed by VM/runtime system that consume more abstract IRs and lower on-the-fly to whatever architecture they detect they are running on, usually called just-in-time (JIT) compilation. Java and C# are examples of this.

The benefits and drawbacks of both approaches are well-known [x] so those arguments will not be rehashed here. But Rasqals approach falls very solidly in the latter, so JIT compilation can be performed. More interestingly, because the nature of hybrid quantum computation is multiple computing paradigms feeding into each other - QPU, CPU, GPU - you have the opportunity to perform novel lowering as you feed the results of one machine back into the other.

We're going to run through and showcase an example of this more complex dynamic lowering. Showing examples where multiple calls to the QPU in quick succession can be used to optimise each other, as well as the classical code it's running.

Note: all following examples are written to highlight the syntax and transformation, they may not produce meaningful results, or even work on every type of machine.

We're going to start with a relatively trivial mid-circuit measurement [x] block to introduce basic lowering. In our following code we perform a measurement on a qubit, look at the result and then conditionally perform more rotations:

```
Unset
q1result = measure q1;
if q1result == 1 {
  x q2;
  y q2;
}
```

This can be compiled statically with minimal effort as long as the QPU supports it, because all of these instructions will be lowered into the QPU and have no outside inputs or outputs.

But when we start adding more traditional classical computations things get more complex:

```
Unset
// This is an argument passed in from outside the code block. We don't know
// it's value.
arg = ...

q1result = measure q1;
if q1result == 1 {
  rx pi/2 * arg q2;
  ry pi/4 * arg q2;
}
```

This block could still be statically compiled but it's now demanding more capabilities of the QPU. Now we need full arithmetic operations, assignments and classical external arguments to be supported. This is getting closer to having equivalent operations to a traditional CPU, which shifts more of the complexity to the QPU, which increases the non-quantum computation it's performing, whether within coherence time or without.

For completeness there are a few tricks we can perform with the previous example that reduces the complexity for the QPU. We can embed our incoming value into the instructions as a constant and then propagate that forward making both multiplications against a static value. This removes some of the complexity from this code as we are now just doing arithmetic against known quantities, but is still not trivial as those values are then used as the arguments for a gate.

Both previous examples can be done by static or dynamic lowering, neither having any real advantage over the other. So now we introduce an example where dynamic lowering does things static cannot:

```
Unset
// This is an argument passed in from outside the code block. We don't know
// it's value.
qubit_count = ...

// This calls a QPU to generate a random number.
random_number = generate_random_number_quantumly(qubit_count)

q1result = measure q1;
if q1result == 1 {
  rx pi/2 * random_number q2;
  ry pi/4 * random_number q2;
```

```
}
```

Attempting to lower this statically is difficult as it involves logic that does not easily map to current QPU capabilities: quantum sub-calls whose results are then used later to influence a quantum gate or operation. To attempt this you would need to deconstruct such a call and embed into the circuit that is currently being lowered. This comes up against all the issues previously raised, along with the difficulties of attempting to merge two different streams of logic into a single circuit, along with the increase of qubits required and gate depth.

Whereas if you're dynamically lowering this block you can do such sub-calls at runtime as you are building the circuit as you go. All the complications of attempting to perform analysis of the sub-call is replaced by simply running it and then forwarding the results.

If we were to dynamically optimise and lower this code, with an assumption that our random number generator returns 5 this is the code that we would then execute:

```
Unset
q1result = measure q1;
if q1result == 1 {
  rx 24.6740... q2;
  ry 3.92699... q2;
}
```

Because Rasqal executes as it goes, it's able to perform any classical operations locally before embedding them into the circuit, while also taking into account the results of quantum sub-calls as well. This means that code that previously wouldn't have been able to be lowered into QPU instructions now can, because we've dynamically executed and resolved all the operations it can't deal with natively.

This technique also allows for dynamically lowering based upon the QPUs capabilities at the moment of runtime. If a QPU was able to process subcalls the system would've lowered as-is, whereas if a machine wasn't able to do so it would have simplified until it found a combination of instructions the QPU was able to run.

This example is relatively straight-forward, but a dynamic system like this can also be used to implement very complicated transformations from various heuristics. We've kept the output similar to help comprehension, but there is no limitation that the output of such lowering resembles the input logic.

xx. Contextual Optimization

What's mentioned here is not included yet in Rasqal itself

Constant folding, sometimes called constant propagation [x], is the act of replacing pointer access and arithmetic with the results of the operation. This means that ‘a+b’ is replaced with ‘7’ if a = 3 and b = 4. This then continues, taking the new constants and replacing them, until it can no longer do so. In an extreme scenario, if we had code which was fully constant, it would simplify away to a single return value.

Constant propagation is a very powerful optimization because it can cut off computationally expensive pathways if it knows they can never be accessed. But it joins other techniques, such as abstract interpretation [x], that all have one goal: knowing what a variable in code can point to at any one time. A variable being resolved as constant is the perfect outcome as we know definitely what the value is. Usually things aren’t as simple and instead variables can be a collection of results. But even winnowing variables down to 5, 10, 20 distinct values has its benefits for optimization since you can check these against conditionals and trim execution pathways.

A lot of these ideals are what powers bounded model checkers [x] and formal verification tools [x]. Rasqal takes inspiration from these and attempts to reduce nondeterminism in the IR as it processes it to then use these to optimise its own executions.

This is done by:

1. Embedding all arguments and inputs as constants directly in the IR.
2. Ignoring any intrinsic methods [x] that are not recognized as well-known quantum operations or a system call that can be resolved into a constant value. An example of the latter is loading a file as a string into memory and treating it as a constant.

Any other intrinsics get ignored. This can leave the IR in an invalid state if any calculations rely on results from the ignored intrinsic, but it’s expected any system passing the IR in knows about this restriction and has taken measures to avoid it.

With this done the only variable values are the results from a QPU execution, and when these are returned they then get propagated as well.

```
Unset
// Seed value for the random number generator.
random_quantum_number = qrng(7)

if random_quantum_number > 500 {
  X 3.14159...
} else {
  // CNOT across every qubit available.
}
```

In the above example all variables have been turned into constants for clarity. Rascal will then execute this line-by-line, first retrieving the result of the ‘qrng’ method and then dynamically evaluating the condition using it. If the condition is true we’ve avoided adding a

large amount of complexity to the circuit being built. This can then trigger further optimizations, or allow the complexity of the built circuit to be below the threshold for simulation or more niche optimization passes.

This not only allows for conditional splicing, but allows calculated gate rotations and qubit target logic to be done without that complexity leaking into the QPU instructions themselves.

xx. Quantum Circuit folding

We now combine both previous concepts together to show off a more novel transformation: hybrid circuit folding. This allows disparate circuits that feed values into one another to be merged, along with any classical operations that would have affected them. This is not currently implemented in Rasqal but is a good example of non-trivial optimizations its features enable.

We'll run through a concrete example as it's easier to show the steps via pseudo-code.

```
Unset
result = list()

// Generate 20 random numbers via our QPU, then generate some random seed
// values.
for 20 iterations:
  random_number = quantum_rng()
  first_seed = generate_quantum_seed(random_number);
  second_seed = (first_seed * random_number) + 42;
  final_seed = generate_quantum_seed(second_seed);
  result.add(final_seed);
```

This code has arbitrarily interwoven quantum and classical data-flow, each feeding into the other in non-trivial ways:

1. Multiple quantum results feeding into separate quantum methods.
2. Quantum results being operated on with classical values before being used as an argument to another quantum method.
3. Results being added to a list that is outside the loop.

One feature of Rasqal that hasn't come up yet that is critical to the following example is that when it executes a measure statement, it returns a deferred quantum result. Until that deferred result is used in a context where the value of it is actually required, it doesn't call out to the QPU.

This has many benefits, but the crucial one is that the conditions that trigger QPU execution are myriad. But for us, if we have an operation that we can detect is *foldable* into the deferred circuit, we do so, avoiding any QPU execution, continuing the deferral and adding to the quantum circuit being built.

Let's take a fragment of our example and do that.

```
Unset
random_number = quantum_rng()
first_seed = generate_quantum_seed(random_number);
second_seed = (first_seed * random_number) + 42;
```

In this fragment we have two potential targets for folding: random_number and second_seed. The former, while theoretically foldable, has no obvious benefits to be gained from folding. The circuit generated underneath may simply not be amenable to merging into other circuits effectively. As these intrinsics are not meant to be concrete circuits, and in attempting to fold we would need thus, our theoretical system will just bypass this analysis, tagging it as unfoldable.

As we are not folding, *random_number* can now be considered a constant, which then allows us to attempt folding *second_seed* into the deferred result, *first_seed*.

As quantum addition [x] and multiplication [x] already have circuit fragments that can perform them and our arguments to these operations are constant, folding these previously classic operations into our circuit is viable.

After folding our example now becomes this:

```
Unset
result = list()

// Generate 20 random numbers via our QPU, then generate some random seed
// values.
for 20 iterations:
  random_number = quantum_rng()
  first_seed = generate_quantum_seed(random_number);
  final_seed = generate_quantum_seed(first_seed);
  result.add(final_seed);
```

Even a small fold has managed to reduce the hybrid nature of this code, opening it up for more targeted optimizations.

If we take liberties and assume that: all our quantum functions can fold into each other; we have a machine that can run such a combined circuit; the machine can run loops and post-process the results natively.

When all this happens the code then becomes this:

```
Unset
// We've synthesized a new quantum function that performs the entireity
// of our previous code purely quantumly.
result = twenty_quantum_seeds()
```

With all our quantum code in one place it also allows optimizers to see the full picture and be able to perform optimization that was impossible when the circuit was split into three.

This is all done at runtime. As the system runs it looks at incoming operations for foldability, then looks forward and back to see how much it should contextually fold, performs (or not) the fold operation, then continues. This incremental folding is what powers the above example.

We've now covered three distinct keystone concepts that allow Rasqal to perform novel optimization on the hybrid circuits it processes. Further work is being done

x. Rasqal Implementation

For the rest of this paper we're going to talk about the algorithms and data structures in detail which constitute Rasqal and power the concepts previously mentioned. Everything mentioned can be found in its Github repository [x].

To we're going to talk about QIR [x] and how Rasqal generates its structures from it, introduce its closed-contxt DAG [x] called a logic graph, run through the symbolic execution and deferral systems, then finally do a round-up of similar systems and where systems like Rasqal sit.

xx. QIR / LLVM

QIR [x] (quantum intermediate representation) is a subset of LLVM IR [x] which adds in gate-level quantum operations and specific metadata like requested result formats, qubit count and hardware capabilities required to execute it. It is Rasqal's primary input format right now as it allows for fully expressive quantum-classical hybrid code to be written, as well as all supporting tools - such as parsers, validators, optimizers - open source and actively maintained.

LLVM itself is a well-regarded and mature compilation toolchain and its IR good at representing abstract classical instructions in a way that can then be targeted at numerous different backend architectures. The IR itself focuses purely on pointers with no mention of registers or memory beyond requests to store and load specific. This allows many types of target architectures to be supported because it makes no assumption on how the hardware itself should deal with memory or processing.

This capacity is something we leverage fully as we are not compiling to a backend but a particular analysis structure. We reconstruct the execution graph that the IR was generated from even though it's now devolved into pointers, arithmetic, intrinsics and array operations.

Beyond the parser and some basic optimizations further LLVM features are not used, as we're not compiling into an ISA [x] to target a machine. LLVMs own interpreter/JIT is also not used because we want to dynamically transform branching logic and do more invasive changes than would be available easily via the Rust wrappers we use. Specifically, Inkwell [x] and llvm-sys [x] which expose LLVMs C API to Rust.

Future enhancements may look into reusing LLVMs JIT capabilities in C++ to write a more robust QIR to logic graph generator.

xx. Logic Graph

After parsing we end up with a collection of logic graphs, a directed acyclic graph [x] (DAG) that's intent is a mixture of control-flow graph [x] (CFG) and abstract syntax tree [x] (AST). It encodes branching and jumps in the edges of the graph and its nodes are the expressions we want to run, both classical and quantum.

But as mentioned in the concepts this graph has a special constraint: all values that are used in, or are the result of, expressions have to be generated during runtime or passed in as arguments. If there are any system calls or I/O operations they need to be resolved before building the graph, not during runtime. The only operations that don't follow this rule are the expressions that demand running something on specialist hardware like QPUs and GPUs.

Future work may allow more harmless system operations to be used such as getting the current time, but this decision was made consciously. The more nondeterminism the graph containers the less optimizations you can run on it. Conceptually a logic graph is viewed as a fragment of a pre-existing program that is meant to be wholly runnable on, or near, a quantum computer.

These are the nodes that are allowed in the graph.

- Assignments.
- Arithmetic and bitwise operations.
- Call to another logic graph.
- Equality comparisons.
- Quantum operations.
    a. Initialise.
    b. Activate/deactivate qubit.
    c. Gates.
    d. Reset.
- Labels.
- Returns.
- Logging.
- Throws.

Branches, jumps and other control-flow logic don't have individual nodes as they are embedded into the structure of the graphs themselves.

A node can only have multiple outward edges if one or more have conditions applied to them

and that edge will then only be taken if its expression is satisfied. Variable liveness is scoped to these edges as well, called an edge assignment. When an edge is taken all assignments are performed, which can include expressions.

The values that are embedded in the graph before runtime are either references to variables, which are effectively pseudo-pointers, or primitives.

With these our graph fully represents a QIR programs logic in its lowest form. This has benefits beyond lowering and execution, because it means such algorithms are also closer to their algebraic form. But this aspect of it is only being used in future work, not in its current implementation.

xx. Runtime

When the runtime gets passed a graph to execute it treats that as the entrypoint, takes the current arguments being passed to it, loads those into the local execution context mapping input name to value, then executes from the entry node until it reaches an exit point.

While it's doing this it will be performing some of the runtime contextual optimization and analysis mentioned in the previous section, such as conditional lowering, unrolling and squashing quantum executions.

The runtime continues walking the graph, performing operations and loading variables into states until it reaches a quantum operation. When it does so it creates a *quantum projection* that will record and scope the QPU execution currently being built. This projection receives every operation and instruction that should be run against the QPU, including any previously-classic operation that has been lowered to QPU instructions.

After projection creation the execution continues evaluation until it reaches a measure operation. Measures are considered special because they return a result which can be used by the symbolic executor, but as it's still building the circuit we don't want to execute against a QPU at the first measure operation. In many situations this would preemptively execute a circuit that hasn't been fully built yet, as most quantum languages have implied scoping blocks around when a circuit is finished. For QASM it's the entire file itself, for Q# it's when every qubit drops out of scope for example.

Rasqals scoping boundaries are far more malleable by design, but it has one rule that is absolute: if you're attempting to use the result of a measure, and that result can't be lowered or folded, then it's assumed your circuit has been fully built. If you're attempting to use a quantum result for a purely classical calculation then it's assumed your circuit is finished and you want to then use that value.

This allows the IRs being consumed to be almost entirely freeform as all information about quantum-classical boundaries are inferred from the data-flow itself. It also allows for some highly complex interwoven code, since two projections can be active at the same time and be executed - or folded - alongside one another.

Furthermore due to deferred execution and folding, potentially entire hybrid programs can be squashed into a single quantum circuit. Due to the cyclic nature of the optimization and folding techniques hybrid programs that get executed might have little in common to the IR that got passed in.

While currently under development it would be remiss to not mention the reverse of this scenario: that if a quantum circuit is trivial enough to simulate in the runtime, that it gets solved fully classically. This can then trigger a chain reaction to simplify other quantum circuits and operations, which bring them below the acceptable simulation threshold, turning a hybrid algorithm fully classical.

Both quantum and classical folding are currently being developed and will likely have their own associated papers, but are worth highlighting as novel execution/optimization techniques that are viable due to how Rasqal is built.

But without either sort of folding, once a deferred quantum result clashes with a classical operation the circuit is traditionally optimised then run against a QPU backend returning the classical value and injecting it in as a constant to be forwarded.

QPU backends in Rasqal are defined in Python with a subset of their interfaces being exposed via Rust bindings. When a deferred quantum result needs to be executed the RUst code calls these API's to define the sort of circuit it wants to run using gate semantics along with custom lowering-specific API's such as embedded conditionals, loops and classical arithmetic.

Rasqal can also be attached to multiple QPU's, and when deciding what QPU to run a circuit against it will introspect the finalised circuit, then choose the best target based upon the feature demands of the circuit itself. This includes checking for fundamental requirements such as qubit count and embedded classical arithmetic/logic. As time goes on such requirements can become more nuanced in their decision-making, taking into account the algorithm being run and if a certain QPU is a better fit for it. But initial release only checks against features the algorithm needs to run at all, such as required qubits and fundamental gates/instructions.

The symbolic executor continues this processing until it reaches a natural exit point, whether by an exit node in the initial entry graph or via an exception. In the former, if a return statement exists it will return the variable associated with that, otherwise return nothing. If an exception is thrown it will cancel execution and be propagated to the caller.

x. Conclusion

In this paper we've introduced Rasqal, a hybrid classical-quantum analysis runtime, its core concepts and the data structures that power it. The system was built to try and answer how effective fully dynamic compilation and optimization strategies could be when applied to hybrid code. Initial findings are promising and we've outlined here some basic strategies that such a dynamic approach is well suited for.