1. Summary

In this paper we introduce Munchkin, 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. Introduction

Quantum-classical hybrid execution - running a quantum circuit against a quantum computer, looking at the returned results and refining the next run - is the lynchpin of most current algorithms. The name is a bit of a misnomer since people will always be using classical code to prepare circuits and arguments and then consume and process the results, so this phrase is used when meaning a quantum computer can process *both* classical and quantum code on the same machine or cluster of machines.

This idea can be interpreted multiple ways. Some, such as IBMs hybrid computation model, involve sending classical code to a CPU node in a cloud and it sends purely-quantum requests to a remotely-accessible quantum computer node. Others process the classical code directly on the quantum computer itself, normally using field programmable gate arrays (FPGAs).

Note: Need references and add more ways about people doing hybrid computation.

Neither of these are mutually exclusive and the community as a whole is coming up with different ways at quite a swift pace. Many of these ideas can be combined.

This paper will focus on one particular piece of the hybrid computation puzzle: a runtime which consumes hybrid code and then is responsible for optimising, then executing, both classical and quantum sections before returning a result.

We'll be introducing some of the foundation ideas which power our runtime, called Munchkin, before delving into the libraries and intermediate representations (IRs) we use to power it. These include the quantum intermediate representation (QIR) and low-level virtual machine project (LLVM).

Munchkin itself is open-sourced and hosted on Github. Everything talked about here can be seen, fully or in part, there.

x. Contribution

In the following sections we introduce Munchkin an full-spec QIR hybrid runtime capable of running arbitrarily interwoven classical-quantum code against a local or remote QPU.

We delve into its symbolic executor which allows it to perform heavy execution deferral, dynamic lowering of stretches of instructions into the FPGA and fully constant parameterized circuit synthesis among other ideas.

Using symbolic execution and heavily dynamic interpretation of quantum-classical hybrid code to power optimization and execution is our primary contribution to state-of-the-art.

Along with this we also provide a Rust implementation of the runtime which uses everything mentioned in this paper.

x. Concepts

While there are many complexities with fully interwoven quantum-classical code we're going to focus on the three that Munchkin solves well:

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. With restricting analysis to purely in-memory manipulations and no system calls, you have a full view of the state of a program so all values are effectively constant. You can aggressively optimise off the back of this.
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.

The key intuition of this paper is that symbolic execution/analysis is a novel way to solve these problems than attempting to do it via a more static approach.

While we will cover our specific approach in depth later, a basic description of symbolic execution/analysis is when you take code, usually in IR form, and run it line-by-line using another program to work out the global and local state of variables at each point in the program. This is then used for any number of optimizations, transformations or just straight-up executing the program and is similar to how just-in-time (JIT) compilation or interpreted languages such as Python work.

Munchkin then uses the variable state information gathered while stepping through the code to: help improve the circuit to be run on a QPU using surrounding contextual information; making circuits entirely constant even if the arguments are variable; and heavily defer QPU execution until the actual result of the value is used in a non-quantum context.

We will now talk about how these ideas help solve the previously-mentioned complexities of interwoven hybrid code.

xx. Dynamic Lowering

As QPU hardware improves, more classical logic will be able to be run on the machine itself whether it's associated with a mid-circuit measurement or not. Compilers in the near future are going to need to be able to analyse when such logic can be lowered into the machine and if so, how. They also need to be able to detect when this is not possible and decide if they can emulate the same logic in another fashion or if it's impossible.

In logically trivial code such as:

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

You can statically analyse whether the result operation can be lowered or not. In this case the operations within our condition are purely quantum and the condition is also only reliant upon a measure result. So this block is purely quantum with no traditional classical operations. We can build whatever instructions are required for the hardware to perform such an operation pretty easily with no further analysis required.

But what if we have something that interweaves more classical code:

```
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 gets more interesting as our argument - arg - is nondeterministic. We don't know what it will be until we get passed the value and it will vary between runs. It may even be a result computed earlier in the script via quantum code so is entirely unknowable until we run it.

We can still do this if we lower absolutely everything into the QPU but every time you need to do any form of calculation that isn't taking advantage of the quantum nature of the machine you are wasting time. This is absolutely critical depending on the coherence time of your qubits. Superconducting machines are fast but have a low coherence time as a trade-off, so for them running long classical calculations drastically reduces the complexity of the quantum code you can run and expect to get a decently precise result out of.

To be able to selectively lower only the code that is absolutely necessary into the QPU and pre-process everything else requires quite a bit of up-front analysis of this interwoven classical/quantum code.
Symbolic execution here helps with this problem because when you reach blocks of logic that require complicated lowering rules you can look ahead at the incoming branches, as well as the surrounding state of the system, and make nuanced decisions that may be difficult or impossible to do via static analysis.

Let us go back to our immediate example that takes a nondeterministic argument in. But let's make it even more interesting by making our argument the result of another quantum call that itself takes in a nondeterministic value.

```
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;
}
```

This modified example is incredibly difficult, if not impossible, to try and solve statically. You can attempt to simulate the QRNG method to help, but as the algorithm uses more qubits it will reach the point where it's unviable to simulate. At that point all attempts to solve it break down.

Not the case with symbolic execution. It can just run the QRNG function directly and use the value as a constant to embed in our rotations as the system executes. So let's say our quantum randomness generator returns a 5. Our code simplifies away to:

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

This then means we can answer our initial question easily: we can indeed lower this directly into the machine. This then gets turned entirely into hardware instructions.

xx. Fully Constant Code

If you write a method that has no variance in how it runs, it can be entirely optimised away to return just its result.

```
Unset
ratio = 5*1.52
result = (7 \ 242) * ratio
```

The above will always return 0.219834710(…). If this was in a traditional method, the compiler would just embed the result in all places that called it and the method itself would vanish from the compiled code.

You cannot do this as easily with quantum code, as simulating it with any precision is incredibly difficult past a certain point.

This means that any value returned from a quantum computer is unknowable at compilation time and anything that value interacts with is equally unknowable. This nondeterminism means that an optimizer's view of the world is incomplete, so it can't do its job as effectively.

As we move into the era of hybrid algorithms and quantum computers being able to run more classical concepts directly on the machine, increasingly complicated algorithms will require hybrid optimization techniques. Techniques which will process both the quantum algorithm as well as its surrounding classical code, optimising both parts at the same time.

We talked about one of these in the previous section: being able to contextually simplify and lower classical logic into a quantum computer. Such optimization techniques do their best work when they have full knowledge of both the classical and quantum state at any point in a program. But as we've ascertained, being able to do this statically is incredibly difficult.

Symbolic execution is the answer here as well, but we have to define a few important characteristics of our particular approach. We've said nondeterminism is the death of optimization passes, so, logically, we want to remove as many instances of it as possible. Unfortunately there are lots of points of nondeterminism in code. External arguments to a system, OS/system calls and disk I/O being some of the most prominent.

So we remove all of these. Arguments are embedded to become constants, system calls and disk reads are ignored, stubbed if possible, or rejected. This means as we run our symbolic execution and analysis we will know every value at every point, except for a result of a quantum execution. But as we are not just compiling the code, but executing it as well, we can get this result as well. You just run the built-up quantum code against a quantum computer and insert the result as a constant directly back into your symbolic execution.

This means that there are no holes at all from an analysis standpoint. Both classical and quantum results are 'constant' in the eyes of the system. This is incredibly powerful when considering optimizations because we can now dynamically optimise code as we execute, in some cases dramatically simplifying it.

```
Unset
args = [...]

random_quantum_number = qrng(arg[0])

if random_quantum_number > 500 {
  X arg[1]
} else {
  // CNOT across every qubit available.
}
```

In the above example we can entirely cut off one of these branches depending on the results. In the first case we've reduced the execution to a level that we can classically simulate results, sparing a run against the quantum computer.

This also allows us to run sub-executions of a circuit to simplify an outer one, perform circuit splicing using contextual clues of the surrounding code, or a whole range of experimental approaches to simplify the overall complexity of a quantum circuit by using dynamic hybrid optimizations.

xx. Contextual Optimizations

Being able to fully represent both classical and quantum sides of an algorithm in a unified form allows for some interesting transformations and optimizations even when we're performing static compilation.

For a trivial example, let's look at this code:

```
Unset
for 20 iterations:
  // ... perform various gates to do something.
  result = measure q1-q20;
```

This just has a flat loop that runs its quantum operations 20 times with no values escaping the loop, early returns or complicated syntax. It also implies that the measure is the final block of this execution, as nothing else happens after it.

Since the body of the loop is entirely quantum and it iterates 20 times this is pretty easy to deduce that it might be a good target for some form of optimization.

Some machines would be able to lower the loop directly into the hardware to drastically speed up the execution loop. Others batch them together to run across multiple machines. This is before the fact that even executing such a loop as it's written will likely be on a CPU near, or on, the QPU itself. None of these are mutually exclusive, but each one is faster than attempting to send through 20 individual requests to a QPU.

Let's move on to a more interesting example.

```
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);
```

Let's break down the parts that make this one tricky to reason about.

1. While we still have 20 iterations the loop contains multiple quantum sub-calls, classical arithmetic and a result that leaks outside of its scope.
2. There are three quantum calls whose results we can't reliably reason about.
3. We're combining two separate quantum results and then adding a flat number. This then gets used as an argument.
4. The final seed gets added to a results list which is outside the loop.

While some of this can be dealt with statically there are some unique opportunities we can exploit due to our symbolic executor heavily deferring quantum results.

This means that even when a circuit is considered finished in the code and we have a variable that holds our measure result, it hasn't been run on our QPU yet. In fact it goes more than simply deferring, whenever an expression is performed on our deferred result it attempts to calculate whether that expression can be *built into* our quantum circuit. If not it then calls our QPU and we get a result, otherwise our circuit stays deferred just slightly bigger than it was previously.

With this in mind let's look at a part of our previous example and see where this could be applied with some benefit.

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

Our target is that third line. Is there any way using deferred results could simplify this?

random_number in this example has already been used as an argument to generate our first seed, let's assume that its result has been embedded into whatever circuit that method builds. So that's no longer deferred and is considered a constant value.

first_seed has just been returned so is currently deferred. The expression it's currently being used in is composed of a literal and an already-returned quantum execution, both considered constants by the symbolic executor. So it will attempt to fold them into our deferred circuit built from generate_quantum_seed.

For this example we're going to say that our backing quantum computer is both perfect and infinite in size so we sidestep qubit usage and length. If we attempted this on a real machine it would evaluate whether the newly generated circuit could fit current hardware and fail if not, whether fully or in part.

Quantum adders and multipliers already exist and as these numbers are being applied to the results of our circuit we can slot fragments in just before the measure. Again, if there are situations where this can't be done accurately the folding will simply fail and our system will run the code as-is. But for this situation let's assume we can.

```
Unset
random_number = quantum_rng()
first_seed = generate_quantum_seed(random_number);

// Both of these calls are added to the deferred circuit, so we still
// haven't called to a QPU.
first_seed.multiply_by(random_number);
first_seed.add(42);
```

This entire fragment is now fully quantum as we have folded the classical sections into our deferred circuit. But there's one more thing we can do here.

Now that we're using random_number in a purely quantum fashion we no longer need to treat the value as classical. The results of one quantum execution are only used directly in another. This means we can attempt to fold the two circuits into one another, eliminating an entire run on the QPU. The folding may still fail for multiple reasons, including the fact that the system simply deems it not a worthwhile optimization, but let's assume it succeeds.

Now our example fragment becomes:

```
Unset
first_seed = generate_random_quantum_seed();
first_seed.add(42);
```

We can then fold this once more just by moving the addition directly into our quantum method, turning it into a single line.

Our full example now looks like this:

```
Unset
result = list()

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

There are more things we could try here - such as folding both the loop and list creation into our quantum method - but those are less likely to succeed in a real situation, so our example system has deemed them unviable.

This has shown that when dealing with complicated mixed-context code a symbolic execution approach with heavy result deference can enable some pretty interesting techniques.

xx. Summary

Above we have shown three approaches to dynamically optimising hybrid code that would be incredibly difficult to do via a more static solution or simply be less accurate due to nondeterminism.

But this is not an argument for or against either dynamic or static compilation. They are not mutually exclusive and either approach should be used if it's best suited to a particular problem. This philosophy very closely mirrors more traditional compilation with the rise of JIT and interpreters in recent times, as well as more interesting static analysis tooling.

xx. Future Work

These three points are the most easily quantifiable benefits of a more dynamic approach to running hybrid quantum code, but there are more problems that may have unique advantages for using this approach. QEC and continual calibration and maintenance of hardware would likely benefit here, as well as more novel approaches to splitting a single execution across multiple machines with circuit splicing/weaving.

It also could allow for far more fine-grained debugging on live or simulated systems as it steps through code much more closely than existing approaches, or provide real-time code completion or recommendations with a lightweight predictive simulator.

But this is purely conjecture for now. More work will be needed to be done to see in what areas such an approach could have significant impact.

x. Munchkin

In the previous sections we've talked about the concepts Munchkin uses and some of the problems it can solve well. Now we're going to talk about the algorithms and data structures which are used to power and enable these ideas.

As mentioned in the introduction everything here can be found in our Github repository. Feel free to reference that for more information.

Note: following sections are written with the assumption that readers are familiar with compilers, static analysis tools and the techniques they both use. Links will be provided for further reading but common concepts and phrases will not be elaborated upon in-line.

Munchkin can be loosely defined as a symbolic execution and optimization tool. Its goal is to execute hybrid code and last-minute optimise it for certain backend targets depending upon what the code is written to target: CPU, GPU or QPU. To do this it borrows heavily from a smorgasbord of techniques across compilers, runtimes, static analysis and verification tools.

To begin with we're going to talk about QIR and LLVM, specifically in how Munchkin parses QIR into its own structures. Following that we will delve into the logic graph which is its primary data structure, then how it is symbolically executed and optimised against myriad hardware. Finally we'll do a quick round-up of similar systems, where Munchkin sits in the current ecosystem and where future efforts may go.

xx. QIR / LLVM

QIR (quantum intermediate representation) is a subset of LLVM (low-level virtual machine) IR which adds in gate-level quantum operations and specific metadata like requested result formats, qubit count and hardware capabilities required to execute it.

LLVM itself is a well-known and well-regarded 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, no mention of registers or memory come into it beyond requests to store and load specific types. 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 at all.

But practically speaking we don't use much of LLVM beyond its IR and parsing capabilities. Since we're using symbolic execution registers don't come into it, all we care about is which variables point at which values. You also can't apply purely classical optimizations across arbitrarily interwoven classical/quantum code due to the subtle relations between them that need to be dealt with by more specific compilation.

We don't use any of its machine emission code either due to the effort tablegen requires to write non-trivial backends and that some of our concepts don't fit well into its existing structures. Nor do we use LLVMs built-in interpreter, symbolic analysis or IR linking and execution because we want to know about the branching syntax and what triggered it, which these methods (to their credit) don't need any implementer to care about.

As Munchkin is written in Rust we use wrappers to access LLVM namely Inkwell and llvm-sys. Using these we parse the IR, run any passes we want, and then walk the IR building our own graph from it.

xx. Logic Graph

After parsing we end up with a logic graph, a directed acyclic graph (DAG) that's intent is a mixture of control-flow graph (CFG) and abstract syntax tree (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 the graph has a rather special constraint: all values that are used in, or are the result of, our 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 QPU/GPUs.

In time some harmless system operations may be allowed such as getting the current time, but this decision was made consciously. The more nondeterminism you add into the graph the less optimizations you can run on the graph itself and the calculation you're running. A logic graph is viewed as a fragment of a pre-existing programing that is meant to be wholly runnable on, or near, a quantum computer. It's not the whole program.

The most important distinction here is that what's contained in a graph can have some impact on optimization of the hybrid code. Conditionals that can be lowered into the hardware, expressions that produce rotation angles, unrollable loops, or any values that can trigger these are basic examples.

All of this is reinforced by the nodes themselves and what they're allowed to represent:

- Assignments.
- Arithmetic and bitwise operations.
- A call to another graph.
- Equality comparisons.
- Quantum-related operations
- Labels.
- Returns.
- Logging.
- Throws.

Note that branches, jumps and control-flow logic is not mentioned here. That's because such information is embedded into the edges of the graphs themselves.

A node can only have multiple outward edges if one or more have conditions applied to them. That edge will then only be taken if its expression is satisfied. To help with scoping variables, edges can also have what's known as an 'edge assignment' which means when that edge is taken a variety of assignment expressions will be applied at the same time.

Finally, a quick mention of the values that inhabit the graph and flow through it. A value is an object that represents traditional programming primitives, strings, arrays, qubits or references to other values. While these are the only ones that exist in the graph before it gets executed, additional values are used during runtime for deference and external hardware execution. They'll be covered in more detail later on.

Once our logic graph has been built and statically optimised it then gets passed to the runtime.

xx. Runtime

Munchkins runtime is actually multiple mini-runtimes that each take care of executing against a specific piece of hardware. You pass it a hardware configuration for its current run which provides data about what sort of equipment is available and its capabilities, which it takes together with the logic graph and some arguments.

The main process takes care of symbolic execution, while a QPU adaptor is available to run synthesised circuits that are generated during graph execution. In the future we can also include GPU or adapters that call out to more discrete systems like HPC blocks.

It starts walking the graph by picking a designated entry-point, which is usually defined by the IR we're parsing from or via the user, load the arguments into the local context which consist of both variable name and value. It then executes that nodes particular operation and follows any edges, performing both conditional edge evaluation and edge assignment when doing so.

While it's doing this it will be performing some of the runtime contextual optimization and analysis mentioned in the concepts 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, which right now constitutes one of these:

- Initialise. Currently unused, but will be used to set the initial state of the QPU run.
- Activate/deactivate qubit. Signals when a qubit comes into and falls out of scope.
- Gate. All common gates that allow for universal quantum computation: X, Y, Z, CX, etc.
- Reset.

When reaching these the runtime creates a *quantum projection* that will record and scope our QPU execution. Dynamically lowered code will also be added here, but until a projection has been activated such analysis is not activated, so classical-looking code stays classical.

We then continue execution, evaluating all classical operations and recording all quantum ones until we get to a measure operation. Measures are special because they infer a result from an execution but we don't want to *actually* execute against the QPU until the entire circuit has been built. To solve this problem we defer the execution and while recording the measure operation we return a deferred quantum result to load into our classical context scope.

This deferred result holds a reference to the projection it's deferred against, the actual qubit targets of the measurement and any additional information it needs to filter its result when execution actually happens. This is where quantum folding and large swathes of the ideas talked about in the concepts section come into effect and start to have real value.

A deferred quantum result continues being deferred until it is part of a calculation that cannot be folded or is returned as a result from our first graph. In both situations the projection is frozen, its internal gates and hardware instructions will be collected, optimised and then passed on to the adapter to be run against a local QPU. After it gets the results back it caches them, then returns the value to be used in the triggering expression. In the situation where the deferred measure is only partial - that it spans only a few qubits and not the entire QPU - those results will be filtered before returning.

If another gate is added to a frozen projection then it will unfreeze, drop the cached results and will continue to be able to be used allowing for further synthesis and execution.

The capability of unfreezing projections comes with an interesting question around scoping. Because deferred results can potentially stay entirely unresolved they may pass a wholly new set of quantum operations before it's resolved, or that an unfrozen result gets used in a further logically separate quantum block.

While interesting conceptually - if we have two separate quantum blocks that bleed one value into another, should we merge them? - Initially a projection is only alive as long as its constituent qubits haven't been deactivated. When a deactivate qubit operation is processed, it removes that qubit from a projection. When the final qubit is deactivated that projection is closed and can no longer be added to. Any deferred results that are linked to the projection are unchanged however and can still access cached results or initialise an execution.

The runtime continues symbolically executing, building and closing projections, until it reaches the end of the graph. This happens when a node is reached that has no valid outward edges, or it reaches a return statement in the root logic graph. If the latter it will return the value to the calling code, resolving any deferred results as necessary.

xx. Similar systems

Munchkin and systems like it hold a rather specific position in the current ecosystem: as an optimising runtime that can provide hybrid execution capabilities even if the QPU it's attached to cannot natively consume QIR. Even if it can, it provides optimization processes that cannot be done if you're simply translating the QIR to machine instructions 1-to-1. As

long as a QPU can be run using circuit syntax with additional information about specialist lowered operations, such as mid-circuit measurements, you can use it.

Moreso it doesn't replace existing runtime systems such as Cuda Quantum and Catalyst. Both these systems, or anything that emits QIR, can sit before our system and then feed into it. Any system that views QIR as an input to a QPU can be treated as such too and potentially makes the QIR even better.

Note: Need to expand more here. I want to at least list some of the more interesting projects, if anything just as a reference for other curious people.

x. Conclusion

In the above we've introduced Munchkin, our symbolic execution optimising hybrid runtime, some of the concepts that are enabled when using symbolic execution and a description of the primary data structures and algorithms that power it.

We've covered deferring quantum results and quantum folding that allow for some rather interesting runtime side-effects, as well as their scoping mechanism using projections. Using these ideas as a template, multiple specialist pieces of hardware such as QPUs and GPUs can be run in concert to provide accelerated hybrid computation, though this has not been implemented yet.

Being able to write a hybrid algorithm which can then call into numerous specialist pieces of hardware to help run, or optimise, the code has the potential for some very ambitious algorithms to be run on minimal hardware. Following this vein of thought will be an interesting subject for further study.