A Dynamic Intermediate Representation for Hybrid Quantum-Classical Programs

Alex Rice
University of Edinburgh
United Kingdom
alex.rice@ed.ac.uk

Chris Heunen
University of Edinburgh
United Kingdom
chris.heunen@ed.ac.uk

Tobias Grosser
University of Cambridge
United Kingdom
tobial.grosser@cst.cam.ac.uk

Abstract

Most quantum compilers are founded on the circuit model of quantum computing, and represent a quantum program as a static list of gates. In many important applications, however, the list of gates can only be determined dynamically, at runtime. We introduce an Intermediate Representation (IR) for hybrid quantum-classical programs that represents gates as values. Instead of having a separate operation for each gate, the system only has to choose at runtime which gates to execute allowing classical data and computations to influence the control flow. Several case studies show the power of this IR: representing quantum noise, randomised compilation, quantum error correction, and measurement-based quantum computing. Each of these can be represented compactly in the IR, and we demonstrate optimisations that are not possible in the circuit model.

1 Introduction

Quantum circuits are the prevalent way to represent quantum computations [7]. However, it is increasingly realised that the circuit model is too restrictive. Many applications are more naturally represented as hybrid quantum-classical programs, containing classical data and control flow in addition to quantum data and operations [4, 21, 24, 25]. Current quantum toolkits have begun to support classical computation in their compilers, with IBM recently releasing OpenQASM 3.0 [5] and Quantinuum releasing Tket 2 [27].

Nevertheless, today's quantum circuits cannot represent subcircuits that depend on external classical or probabilistic computations. For example, consider noisy quantum channels, which are crucial in modelling the execution of a quantum program on today's noisy quantum computers [22]. The phase flip channel, a common form of error which occurs in quantum computers, can be represented as the probabilistic application of a Z gate to a qubit. Because of the dependency on a classical random variable, the phase flip channel cannot be represented in the circuit model without generating a fresh circuit for each sample of the random variable.

A better representation of such quantum computations should facilitate optimising across the boundary between

Listing 1 Our dynamic gates can represent two concatenated probabilistic phase flip channels without requiring control flow. Quantum-classical transformations can then reduce this to a single phase flip with updated probability, using the self-invertibility of the quantum Z gate.

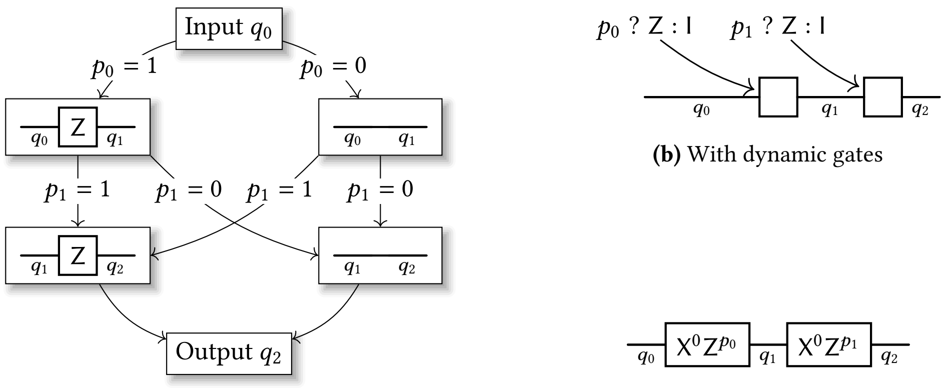
```
%id  = gate.constant #gate.id
%z   = gate.constant #gate.z
%p_1 = prob.bernoulli 0.1
%g_1 = arith.select %p_1, %z, %id : !gate.type<1>
%q_1 = qssa.dyn_gate<%g_1> %q
%p_2 = prob.bernoulli 0.1
%g_2 = arith.select %p_2, %z, %id : !gate.type<1>
%q_2 = qssa.dyn_gate<%g_2> %q_1
```

the quantum and classical components of a hybrid program, should be expressive enough to represent a variety of techniques and programs, and should be simple enough to optimise and transform efficiently. We contrast this situation to classical hardware in an ASIC or on an FPGA, where circuits cannot be changed at runtime except possibly at very high cost [17, 28]. Quantum programs are different, because the circuits are executed dynamically, and therefore changing the circuit layout during execution is possible. In other words, there is no inherent requirement to keep a static circuit model. We can choose a model that is better suited to optimising quantum programs.

In this work, we introduce *dynamic gates* to model hybrid quantum-classical computations. This representation does not define separate operations for each quantum gate, but instead specifies gates via a runtime value we call a *gate value*. These gate values can be manipulated by arbitrary classical computations, providing an interface between the quantum and classical components.

Listing 1 shows two consecutive phase flip noise channels in our dynamic gate intermediate representation (IR). The key feature is the operation `qssa.dyn_gate`, which takes the gate to be run as an operand `%g`. In the example, this gate is obtained using a classical selection operation, which returns a Z gate if the random variable `%p` is true, and the identity gate otherwise. The gate applied by `qssa.dyn_gate` is *dynamic*, in the sense that it can be chosen during the execution of the program, in contrast to the static nature of gates in the traditional circuit model.

Authors' Contact Information: Alex Rice, University of Edinburgh, United Kingdom, alex.rice@ed.ac.uk; Chris Heunen, University of Edinburgh, United Kingdom, chris.heunen@ed.ac.uk; Tobias Grosser, University of Cambridge, United Kingdom, tobial.grosser@cst.cam.ac.uk.



(a) With control flow

(b) With dynamic gates

(c) With XZ gadgets

Figure 1. Graphical representations of two sequential phase flip channels with (a) control flow, with (b) dynamic gates, and with (c) our XZ gadget. The dataflow is vastly simplified in (b) and (c), where the quantum component retains its circuit structure. The XZ gadgets in (c) remove the need for the selection operation, allowing the dynamic gates to be fused.

Dynamic gates make the boundary between the quantum and classical parts of the program more porous, enabling optimisations across this boundary. Furthermore, the dynamic gate model can be regarded as a natural extension of the quantum circuit model, maintaining the underlying circuit structure. Basic versions of specific instances of this dependency are built into existing IRs [5, 23], such as loops and classical control flow over basic blocks.

Our contributions are:

- A novel dynamic gate model for tightly-coupled hybrid quantum-classical programs and its instantiation as a compiler IR for effective optimisations at the quantum-classical interface (Section 2, Section 4).
- A new XZ gadget representing conditional combinations of X and Z Pauli gates, modelled as an operation producing a runtime gate value, and several optimisations enabled by our XZ gadget (Section 4.1).
- Three case studies demonstrating the efficacy of our IR for modelling classical-quantum interaction: (1) *randomised compilation*, (2) *quantum error correction*, and (3) *measurement-based quantum computing* (Section 5).

2 Dynamic Gates

Designing IRs for classical computation is a well-studied problem, as is designing IRs to represent the quantum circuit model. A union of a classical IR and quantum circuit IR can therefore represent both quantum and classical computation, but struggles to model the interaction between the two components, which is integral for many modern quantum computing applications [4, 21, 24, 25].

An example of an optimisation that crosses the boundary between the classical and quantum components of the program is fusing consecutive quantum phase flip channels (Figure 1). In our dynamic gate model this can be represented as a single circuit, whereas it becomes a complicated branching of circuits when mixing classical and quantum IRs.

A phase flip channel acts on a single qubit, and is equivalent to conditionally applying a Z gate to the qubit with some probability p. Phase flip channels may not immediately appear to be hybrid quantum-classical programs, but representing them requires quantum operations to depend on the result of classical computation, in this case the instantiation of a random variable. Thus they demonstrates the tools we have developed well.

Performing two such channels in sequence is equivalent to performing a single one (with an updated probability): with probability $(1-p)^2$ no Z gates are applied, with probability p^2 two Z gates are applied which cancel out, and with probability $2p(1-p)$ a single Z gate is applied. Such a transformation should be simple to perform in a compiler, but note that the necessary transformation is not purely quantum or purely classical. Thus, how easy it is to perform such a simplification depends on the quantum-classical interface exposed by the IR.

One way to represent the channel is to use classical control flow, either as blocks of computation linked by branching operations, or a more structured `if` conditional statement (Figure 1a). Such an approach is not ideal for the transformation described above, as performing it would require reasoning across multiple control flow blocks. Furthermore, this representation obscures the quantum dataflow of the program, because the same qubit can be used in multiple execution branches. This complicates simple analyses. For example, to check if a qubit is used linearly, you now need to check that no execution trace uses the qubit twice. Verifying this property is undecidable in general and common over-approximations require global control flow analysis.

```
%false = arith.constant false
%p_1 = prob.bernoulli 0.1
%g_1 = gate.xz %false, %p_1
%q_1 = qssa.dyn_gate<%g_1> %q
%p_2 = prob.bernoulli 0.1
%g_2 = gate.xz %false, %p_2

%q_2 = qssa.dyn_gate<%g_2> %q_1
```

(a) Double phase flip with XZ gadgets.

```
%false = arith.constant false
%p_1 = prob.bernoulli 0.1


%p_2 = prob.bernoulli 0.1
%p_3 = arith.xori %p_1, %p_2 : i1
%g   = gate.xz %false, %p_3
%q_2 = qssa.dyn_gate<%g> %q
```

(b) Double phase flip after `xzs-fusion`.

Figure 2. The XZ gadget allows the phase flip channel to be represented without any selection operations (a). This allows the consecutive dynamic gates to be fused, reducing the program to a single phase flip channel (b).

Instead, the dynamic gate model bridges the classical and quantum components of a program. The fundamental primitive is the *dynamic gate* operation. Unlike the usual representation of gates in a quantum circuit, the gate to be run is supplied by an operand, instead of a static piece of information. This input can be the result of arbitrary computation, providing the interface for classical data to affect quantum computation. We refer to operands that specify quantum gates as *gate values*.

Listing 1 represents two sequential phase flips in the dynamic gate model using our IR. The gate value is obtained from a selection operation on a randomly generated boolean. This shows the strength of the dynamic gate model: no complexity is added to the quantum dataflow by its interaction with classical dataflow. The output qubit from the first phase flip passes directly to the input of the second phase flip. Essentially, the quantum component of the program retains the structure of a quantum circuit (Figure 1b).

The dataflow is simpler, but two classical choices of quantum gates still need to be fused. In this small scale example a full case analysis is possible, but in a general scenario that would quickly become intractable. To remedy this we introduce the *XZ gadget*, a succinct way to represent conditional combinations of Pauli X and Z gates. So far, all gate values have been constant or classical selections between other constant gate values, but the dynamic gate model can be extended by further operations for generating gate values. Therefore the XZ gadget is implemented as a fresh operation that takes two boolean values x and z as operands and produces a gate value representing the gate $\mathsf{X}^x\mathsf{Z}^z$.

The full utility of XZ gadgets derives from the transformations which apply to it. A selection operation between two XZ gadgets can be replaced by a single XZ gadget with selection operations on each of its operands. Following this transformation with simple arithmetic simplifications reduces the double phase flip to the representation in Figure 2a, as illustrated in Figure 1c.

Moreover, XZ gadgets can be easily fused together by taking an exclusive disjunction of their parameters, which preserves the semantics of the gate value up to global phase. This reduces our example to a single phase flip channel, given in Figure 2b. As the XZ gadget considers X gates in addition to Z gates, the same reduction sequence can similarly simplify compositions of bit-flip and depolarising channels. Section 5 demonstrates that this gadget applies much more widely than the scenario presented here, because conditional Pauli operators are pervasive in quantum computing.

3 Background

Now the key idea of the paper has been introduced, let us step back to recall relevant background. We briefly review compilation, especially in the context of the MLIR framework, and the relevant parts of quantum computation.

Quantum computing

Instead of classical bits, the basic data in quantum computing is carried by *qubits*. The state of a one-qubit system is represented by a vector in $\mathbb{C}^2$. Qubits subsume bits, with 0 and 1 being represented by the unit vectors $|0\rangle := \left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$ and $|1\rangle := \left(\begin{smallmatrix}0\\1\end{smallmatrix}\right)$.

Computations on qubits are represented by *unitary* linear maps $\mathbb{C}^{2^n} \to \mathbb{C}^{2^n}$, that is, invertible matrices whose inverse is equal to their conjugate transpose. Some important one-qubit unitaries are listed in Figure 3. Especially the *Pauli gates* X, Y, and Z satisfy some useful relations, such as

$$\mathsf{X}^2 = \mathsf{Y}^2 = \mathsf{Z}^2 = -i\mathsf{XYZ} = \mathsf{I} = \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)$$

and $\mathsf{ZX} = i\mathsf{Y} = -\mathsf{XZ}$.

Multiple qubits can be combined to form a register of qubits using *tensor product*: the state of an n-qubit system is represented as a vector in $\mathbb{C}^2 \otimes \cdots \otimes \mathbb{C}^2 = (\mathbb{C}^2)^{\otimes n} \simeq \mathbb{C}^{2^n}$. Gates can be combined with tensor products to act in parallel. For example, $\mathsf{X} \otimes \mathsf{X}$ is a 4-by-4 matrix representing a 2-qubit operation. But there are also other 2-qubit unitaries. For example, if U is a one-qubit operation, the *controlled* U gate is a two-qubit operation represented by the block matrix

$$\begin{pmatrix}1 & 0\\ 0 & \mathsf{U}\end{pmatrix}.$$

$$\mathsf{R_X}(\theta) = \begin{pmatrix} \cos(\frac{\theta}{2}) & -i\sin(\frac{\theta}{2}) \\ -i\sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{pmatrix} \qquad \mathsf{X} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$\mathsf{R_Y}(\theta) = \begin{pmatrix} \cos(\frac{\theta}{2}) & -\sin(\frac{\theta}{2}) \\ \sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{pmatrix} \qquad \mathsf{Y} = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

$$\mathsf{R_Z}(\theta) = \begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix} \qquad \mathsf{Z} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

$$\mathsf{J}(\theta) = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & e^{i\theta} \\ 1 & -e^{i\theta} \end{pmatrix} \qquad \mathsf{S} = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$$

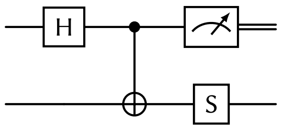
$$\mathsf{H} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \qquad \mathsf{T} = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix}$$

Figure 3. Some important one-qubit gates. The rotation gates are parametrised by an angle $\theta \in [0, 2\pi)$.

Important two-qubit gates are the controlled X and Z gates, denoted CX and CZ. These two gates, along with the Pauli gates, H, and S are known as *Clifford* gates, the gates that map Pauli gates to Pauli gates under conjugation. For example, the phase gate, S, satisfies the following equations:

$$\mathsf{ZS} = \mathsf{SZ} \qquad \mathsf{XS} = -\mathsf{SY} \qquad \mathsf{YS} = \mathsf{SX}$$

Gates can be combined into *quantum circuits*. Just like Boolean circuits, they can be combined in parallel (with tensor products) and in sequence (with matrix multiplication). Quantum circuits form the *de facto* standard hardware-agnostic low-level language for quantum computing. They are usually drawn graphically. For example, the quantum circuit



represents a 2-qubit operation that first performs an H gate on the first qubit, then a CX gate controlled on the first qubit and targeted at the second qubit, and then an S gate on the second qubit.

The last operation on the first qubit is a *measurement*, used to convert qubits into bits. This is a probabilistic operation. When measuring a qubit in state $\alpha v_1 + \beta v_2$ with respect to a basis $\{v_1, v_2\}$, the resulting bit will be 0 with probability $|\alpha|^2$, and 1 with probability $|\beta|^2$. Two states u and v are said to differ by a *global phase* if $v = e^{i\theta}u$ for some angle θ. Such states cannot be discriminated by measurement, and are hence identified.

Along with the noise channels introduced in Section 2, measurement is a special case of a *quantum channel*, which is roughly a convex combination of circuits. For more information on quantum computing, we refer the reader to [19, 31].

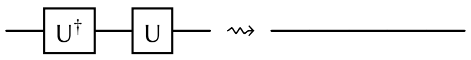
Compilers

A compiler converts a computation from one representation to another, typically from a higher-level program to some form of machine code. The term compiler is often used to refer to an optimising compiler, which attempts to minimise various metrics of the target code, such as program size or execution time. Instead of performing a direct translation, (optimising) compilers often morph a program into different IRs, each of which eases performing certain optimisations or analyses.

It is common for IRs to contain programs in Static Single-Assignment (SSA) form [26], where each identifier is assigned a value in exactly one location. To translate a program into this form, any reassignment of a variable must be replaced with the introduction of a fresh variable. This makes dataflow explicit, as it is trivial to determine the origin of any value and to obtain a list of its uses.

To put a quantum gate in SSA form, it should consume the qubits it acts on, producing new fresh qubits as output. We contrast this *value semantics* view of qubits to the *reference semantics* view of qubits commonly used in existing quantum IRs, such as QASM and QIR, where a quantum gate has no outputs and is understood to be mutating its input qubits. The value of SSA form for optimisation is immediate here; consider a simple optimisation that cancels gates with their inverses. To apply this to a single-qubit gate U with input q, it suffices to examine the unique origin of q. In the reference semantics view of a quantum circuit, q may have been mutated by any number of previously applied gates, complicating this analysis.

More generally, SSA form allows the transformation of IR by *pattern rewriting*. The dataflow of an SSA program can be represented as a graph, and pattern rewrites replace subgraphs of a certain form (where this form is the *pattern*) with a new graph. The example inverse-cancelling reduction above is a pattern rewrite performing the following graph replacement:



Such a rewrite can then be efficiently applied in any program context. Various rewrites can be grouped into a compiler *pass* which together perform some optimisation, and many passes may be composed to form a *pipeline*.

The IR introduced in this paper is specifically presented as a set of MLIR [16] dialects, implemented on top of xDSL [8], a Python clone of MLIR sharing the same textual representation. MLIR is an established library for writing compilers, based on the popular LLVM compiler toolkit, and provides a foundation for writing SSA-based IRs. Dialects in MLIR allow the user to add new types and operations to craft their own intermediate representations, while still having access to core data structures and optimisation machinery.

All IR snippets in this paper follow MLIR syntax in style. Consider the operation

```
%0 = arith.select %p, %lhs, %rhs : !qubit.bit
```

The names `arith` and `qubit` are names of dialects, which contain `select` and `bit` respectively, with the exclamation mark denoting that `!qubit.bit` is an MLIR type. Any symbol beginning with `%` is an SSA value, with operands to the left of the equality symbol and outputs to the right. This operation takes a boolean input `%p` and two inputs (`%lhs` and `%rhs`) or the given type, and returns the first if `%p` is true and the second otherwise.

Extra data can be attached to MLIR operations via attributes. Any symbol preceded by a `#` is an attribute, for example `#gate.id` in Listing 1.

4 A Hybrid Quantum-Classical IR

The IR represents quantum operations in SSA-form, with qubits having value semantics. It is realised as a set of MLIR dialects. In contrast to previous SSA-based quantum IRs [10, 18, 20], we do not introduce a new operation for each quantum gate, but instead add a single operation, `qssa.gate`, for the application of (static) gates. The specific gate to be executed is supplied to the operation by a gate attribute, which takes the form of static piece of compile-time data. Gate attributes each specify the number of qubits they act on, allowing the number of qubit operands and outputs to be verified for each operation.

Specifying gates via attributes has multiple advantages. Users can easily extend the gates available by introducing custom gates through new dialects. Gates added in this way will automatically cooperate with `qssa.gate` and its transformations. Furthermore, the set of available gates can be reused between different applications. Our implementation uses this to define a `qref` dialect in parallel to the `qssa` dialect, which represents quantum circuit operations with memory semantics instead of SSA semantics. All (user) defined gates in the system can be used in both dialects, and transformations are given between the two dialects without the need to inspect the gate attribute.

Most importantly, the various gate attributes can be reused to build our dynamic gate infrastructure. The dynamic gate application operation `qssa.dyn_gate` (and its corresponding `qref` operation) takes the gate to be executed as an operand instead of an attribute. This operand must have type `!gate.type`<n> for some n, the type of n-qubit *gate values*, runtime values storing an n-qubit quantum gate.

Gate values are primarily generated with our constant gate operation, `gate.constant`, which creates a gate value from a gate attribute. Passing a constant gate value directly into a dynamic gate operation is equivalent to applying the corresponding static gate operation, and we give this simplification as a canonicalisation pattern on each respective dynamic gate operation.

Table 1. Common gates as XZ or XZS gadgets. The `convert-to-xzs` compiler pass converts constant gate values of gates in the right hand column to XZ(S) gadgets.

Gate	Gadget Representation
I	`gate.xz  %false, %false`
X	`gate.xz  %true,  %false`
Y	`gate.xz  %true,  %true`
Z	`gate.xz  %false, %true`
S	`gate.xzs %false, %false, %true`
$S^\dagger$	`gate.xzs %false, %true,  %true`

As shown in Listing 1, constant gate values already allow more expressivity than the circuit model when combined classical selection operations. Applying such a gate value is equivalent to branching using an `scf.if` operation. The compiler pass `lower-dyn-gate-to-scf` performs this transformation. Further lowering to basic block control flow gives a representation that is compatible with existing representations – such as QIR [23], which inherits LLVM's control flow structures.

Parallel with dynamic gates, the IR contains a similar system for quantum measurements: a static measurement operation, `qssa.measure`, takes a measurement attribute; and a dynamic measurement operation, `qssa.dyn_measure`, takes a *measurement value* as an operand. In contrast to dynamic gates, a measurement attribute can be omitted, defaulting to a measurement in the computational basis.

4.1 Gadgets for X, Z, and Phase (S) gates

Our implementation contains two extra operations for producing gate values, extending the dynamic gate model: the XZS gadget `gate.xzs`, and the XZ gadget `gate.xz`. The XZS gadget creates compositions of Pauli X, Pauli Z, and Phase gates, and its simplification, the XZ gadget, introduced in Section 2, omits the Phase component.

The XZS gadget takes three boolean values x, z, and s as input, each representing whether an X gate, Z gate, or Phase (S) gate respectively should be included in the output composition $X^x Z^z S^s$, while the XZ gadget has only two inputs for the inclusion of the Pauli X and Z gates. Table 1 translates various gates to their definition with XZS (or XZ) gadgets, with the convention from MLIR that `%false` holds the constant boolean false, and `%true` holds the constant true. The compiler pass `convert-to-xzs` implements this transformation. With this convention we have:

`gate.xzs %x, %z, %false` ≡ `gate.xz %x, %z`

The rewrite from the left-hand side to the right-hand side is added as a canonicalisation pattern on `gate.xzs`. As we only consider gates up to global phase, we can soundly define the Pauli Y gate as the composition of Pauli X and Pauli Z gates, meaning the XZ gadget can represent all Pauli gates.

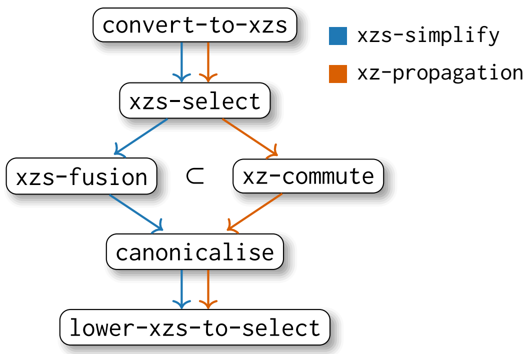


Figure 4. Optimisation pipelines for XZS gadgets. The left path, `xzs-simplify`, fuses all adjacent conditional Pauli and Phase gates. The right path, `xz-propagation`, attempts to move Pauli gates towards the end of the program while performing fusion.

The inputs to XZ and XZS gadgets can be arbitrary SSA-values, and not just constant values, allowing the gadget to represent conditionally applied gates. The power of this construction is derived from the optimisations that this allows. These optimisations organise into two pipelines, `xzs-simplify` and `xz-propagation` (Figure 4). Both pipelines convert (possibly conditional) Pauli and Phase gates to XZS gadgets, perform some transformation on these generated gadgets, and then lower back to selections between constant gates. The `xzs-simplify` pipeline fuses adjacent XZS gadgets, resulting in the entire pipeline simplifying consecutive conditional gates. The `xz-propagation` pipeline pushes XZ gadgets towards the end of the circuit while fusing them, performing Pauli propagation with conditional gates.

Let us we explore each optimisation in more detail.

***Interaction with selection:* `xzs-select`.** If the inputs to an `arith.select` operations are XZS or XZ gadgets, then this selection can be converted to a single XZ(S) gadget with parameters given by selection operations between the parameters of the original gadgets. More concretely,

```
%g_1 = gate.xzs %x_1, %z_1, %s_1
%g_2 = gate.xzs %x_2, %z_2, %s_2
%g_3 = arith.select %p, %g_1, %g_2
```

is converted to

```
%x_3 = arith.select %p, %x_1, %x_2
%z_3 = arith.select %p, %z_1, %z_2
%s_3 = arith.select %p, %s_1, %s_2
%g_3 = gate.xzs %x_3, %z_3, %s_3
```

with similar conversions being given for XZ gadgets and selections between an XZ and XZS gadget.

While this appears to make the program more complex, it is often the case that selections between boolean values can be trivially simplified, often to constant values, reducing the total number of operations. When preceded by the pass `convert-to-xzs` and combined with canonicalisation for `arith.select`, this pass converts any conditional Pauli or Phase gate to its canonical XZ(S) gadget representation. For an example, see Figure 2a.

***Merging sequential gadgets:* `xzs-fusion`.** A key optimisation is fusing together consecutive dynamic applications of XZ(S) gadgets. Given two XZS gadgets with parameters (x, z, s) and (x', z', s') respectively, we have the equation (up to global phase):

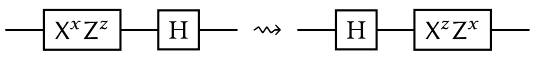
$$\begin{aligned} \mathsf{X}^x\mathsf{Z}^z\mathsf{S}^s \circ \mathsf{X}^{x'}\mathsf{Z}^{z'}\mathsf{S}^{s'} &\simeq \mathsf{X}^x\mathsf{Z}^z(\mathsf{X}^{x'}\mathsf{Z}^{s\wedge x'}\mathsf{S}^s)\mathsf{Z}^{z'}\mathsf{S}^{s'} \\ &\simeq \mathsf{X}^x\mathsf{X}^{x'}\mathsf{Z}^z\mathsf{Z}^{z'}\mathsf{Z}^{s\wedge x'}\mathsf{S}^s\mathsf{S}^{s'} \\ &\simeq \mathsf{X}^{x\oplus x'}\mathsf{Z}^{z\oplus z'\oplus(s\wedge x')}\mathsf{S}^{s\oplus s'} \end{aligned}$$

with the first line following from the equation SX ≃ XZS and a case analysis. The corresponding (much simpler) equation for XZ gadgets is:

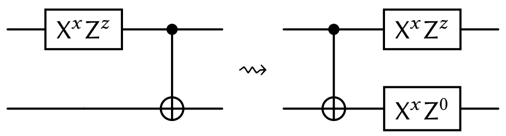
$$\mathsf{X}^x\mathsf{Z}^z \circ \mathsf{X}^{x'}\mathsf{Z}^{z'} \simeq \mathsf{X}^{x\oplus x'}\mathsf{Z}^{z\oplus z'}$$

These equations are implemented through simple pattern rewrites and can often be followed by simplifications of the generated arithmetic expressions.

***Pauli propagation:* `xz-commute`.** As the XZ gadget represents all (conditional) Pauli gates, it is a good tool for performing Pauli propagation, which commutes Pauli gates through other quantum gates, pushing them towards to the end of the circuit. The rules for commuting an XZ gadget through a gate are specific for each gate, but in each case a simple pattern rewrite can be performed. As an example, the case for the Hadamard gate is given by the rewrite:



with X gates being converted to Z gates when commuted past the Hadamard and Z gates being converted to X gates. The case for the CX gate is slightly more complicated; an XZ gadget on the control wire of the CX gate can be commuted as follows:



This case highlights a difficulty of this transformation: an individual commutation rewrite for a two qubit gate can increase the number of XZ gadgets, and a naive implementation of this compiler pass would have an exponential execution time (with respect to the length of the program).

To solve this our pass greedily applies `xz-fusion` rewrites before applying any commutation rewrites. Despite this, applying these rewrites to operations in a non-optimal order can still result in exponential runtime. We therefore write a custom pattern rewrite driver which walks the IR once, ensuring that operations are rewritten in the order they appear in the IR. This ensures a linear runtime of this pass.

***Lowering:* `lower-xzs-to-select`.** The last compiler pass we define for XZ gadgets effectively reverses the action of `convert-to-xzs` and `xzs-select` by converting each XZ(S) gadget to a selection between classical gates. This can be used as a first step to converting these gadgets to a more traditional representation when followed by compiler passes such as `lower-dyn-gate-to-scf`, which was introduced in the previous section.

5 Evaluation through Case Studies

Our work is primarily evaluated through its application to various representative areas of quantum computing. This section briefly introduces each case study, explains its representation in our IR, and discusses the utility of the transformations we have introduced. Each case study is self-contained, with separate domain-specific evaluations. These are select examples of the dynamic gates model, rather than an exhaustive list of applications, but we believe they cover the interaction of hybrid quantum-classical computing.

5.1 Randomised Compilation

Randomised compilation [30] is a procedure for tailoring the noise of a quantum circuit. As opposed to classical computation, quantum computation is probabilistic and hence it is often desirable to run a computation many times, taking the results of measurements as samples. Instead of running the same circuit each time, randomly-generated equivalent (under noiseless execution) circuits can be run, changing the noise characteristics of the computation when the results of successive runs are averaged.

The procedure acts on a circuit in the Clifford+T gate set, and begins by partitioning the gates into *hard* gates (T, H, and CX) and *easy* gates (Clifford $\setminus$ {H, CX}). Before each hard gate, a randomly chosen Pauli gate is applied to each of the input qubits. After the execution of the hard gate, corrective Pauli or Phase gates are applied to the output qubits, preserving the semantics of the circuit. We refer to the random gates added to the circuit (including the corrective operations) as *padding gates*. The padding gates are *easy* gates, which can be fused with other adjacent easy gates, limiting the increase in size of the circuit due to this procedure.

Existing compilers and optimisers for quantum circuits may have the rewrites needed to simplify Pauli and phase gates, but they are not ideal for fusing padding gates because they are not static. Two ways to apply this optimisation are known:

- Instantiate each iteration of the circuit with its randomly generated padding gates, and then optimise the resulting programs in existing compilers individually.
- Generate the random gates at runtime, using a special procedure on the hardware to perform the gate merging in real time. This procedure can possibly run on the same classical control hardware used to perform error correction.

Neither option is optimal. Instantiating early duplicates a lot of work, as each iteration has to go through the entire compilation pipeline. Instantiating at runtime asks more of the hardware, and requires knowing the format of the program once it is being executed.

The fundamental problem is that the random choice of padding gates cannot be represented as part of the quantum program, which prevents optimisation before instantiation of random variables. The dynamic gate IR solves this problem: the randomly generated padding gates can be encoded just like the phase flip channel in Section 2, and can be fused with XZS gadgets (see Section 4.1) long before any random variable has its value chosen.

The padding gates are introduced by the compiler pass `randomized-comp`. This pass matches on each hard gate, adding conditional Pauli X and Z gates before each input with a classical selection on a random variable and feeding the result to a dynamic gate. It then similarly inserts dynamic gates for the correction operations to each output. Crucially, it shares the same random variables for the padding gates before and after the operation. This encodes that the correction gates are deterministic once the choice has been made with which random Pauli gates to pad the hard operation.

Once the conditional padding gates have been added, they can be fused with the `xzs-simplify` pipeline introduced in Section 4.1. More precisely, this converts the padding and correction gates to XZS gadgets, simplifies the classical selection operations before fusing the gadgets, and then lowers back to constant gates and selections. Note that these applications need XZS gadgets; XZ gadgets are not expressive enough. Even though the gates generated before each hard operation are Pauli gates, placing an X gate before a T gate necessarily involves a Phase gate as part of the correction. It is important not to run the `xz-commute` pass in this scenario, as it would commute the padding gates past the hard gates (or at least past CX and H gates), nullifying the effect of the whole procedure.

We summarise two methods of performing randomised compilation in Figure 5:

- The naive pipeline represents the known pipeline for applying randomized compilation; n circuits are obtained by cloning the original circuit, instantiating the random variables with the `flip-coins` pass (which randomly replaces each probabilistic operation by a constant operation), before running a pass to fuse easy gates (which we have named `gate-fusion` in the figure) on each copy.
- The dynamic gate pipeline represents the method detailed in this section, where `xzs-simplify` is run before cloning the circuit and instantiating any random

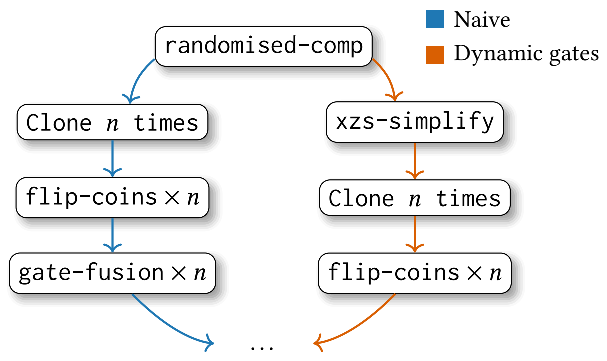


Figure 5. Two compiler pipelines for generating n circuits with the randomised compilation algorithm. The naive pipeline represents the state of the art, while the dynamic gate pipeline utilises dynamic gates to delay any cloning until after optimisation.

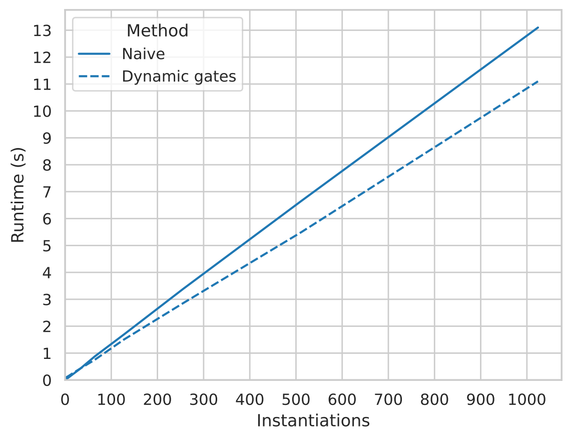


Figure 6. Time taken to generate n randomised circuits from a circuit preparing the 10-qubit GHZ state, with the naive and dynamic gate pipelines outlined in Figure 5. With greater values of n, the dynamic gate pipeline is up to 20% faster.

variables with `flip-coins`. In this pipeline, the fusion operation is only run once, rather than once per iteration.

We benchmark the naive pipeline against the dynamic gate pipeline on a circuit preparing a 10-qubit GHZ state, displaying the results in Figure 6. As expected, the simpler naive pipeline is faster for small numbers of iterations, but is quickly outperformed by the dynamic gate pipeline, which is up to 20% faster for greater instantiation counts.

A secondary benefit of the dynamic gate approach is the size of the generated representation. If restricted to static quantum circuits, each individual instantiation must be stored separately, causing the representation size to scale linearly with the interation count. Conversely, the dynamic gate representation (before instantiation) can be used to create an arbitrary number of samples. The overhead for this is small; the dynamic gate representation of applying randomised compilation to the 10-qubit GHZ state requires 237 operations, whereas the static representation requires (depending on random seeding) around 50 operations per iteration, and so is already less concise at representing 5 samples.

As discussed above, we believe that this approach works better when the random variables persist longer in the compilation pipeline. Evaluating this is out of scope, as this work considers higher-level representations of quantum programs only. The benchmark above is effectively the worst case scenario for the dynamic gate setup, where the circuit is cloned directly after fusion – and yet it still displays a moderate speedup.

5.2 Quantum Error Correction

Another way to combat noise in a computation is quantum error correction (QEC); see Roffe [25] for a summary. Broadly, QEC works by encoding a logical qubit as a state over multiple physical qubits, introducing redundancy. Errors can be detected by syndrome measurements, which preserve the code-space (the states of the physical qubits corresponding to states of the logical qubit). The results of these measurements are then passed into a decoder, a classical function that determines which error has most likely occurred. The physical state can then be corrected with Pauli gates, returning it to the code-space and thereby correcting the errors.

In stabiliser codes, each syndrome measurement consists of CZ and CX gates between qubits containing the state of the logical qubit and a freshly allocated auxiliary qubit, before measuring the auxiliary qubit to get a classical bit. The decoder could in principle be any classical computation. In the case we will discuss, it takes the form of a small boolean circuit, but for production scale QEC codes it could take the form of large matrix calculations or even neural network computations on a GPU or an FPGA. Finally, the output of the decoder is fed into conditional Pauli gates, which have a natural representation in the dynamic gate model.

To preserve a quantum state, these three phases are run in a loop. QEC protocols can only tolerate a certain number of errors while preserving the quantum state. Noise effects idle qubits, so the protocol works better when the time between successive cycles of syndrome measurements is shorter.

One method of reducing the time between syndrome measurements is to delay the corrective operations. The gates used in QEC are all part of the *Clifford group*, the gates which normalise the Pauli gates, allowing the correction operations to be propagated past the next round of syndrome measurements. The correction operations for successive cycles can be fused, reducing the number of quantum operations required. More importantly, they can be buffered: the next round of

Table 2. The number of operations needed to present a various number of cycles of the perfect 5-qubit QEC code in the dynamic gate IR, both before and after application of the `xz-propagation` pipeline. The operation counts are split into quantum operations (allocations, gates, and measurements) and any other operation.

	Operations (#)			
	Before `xz-propagation`		**After** `xz-propagation`	
Cycles (#)	**Quantum**	**Other**	**Quantum**	**Other**
1	42	40	37	32
10	420	400	325	500
100	4200	4000	3205	5180
1000	42000	40000	32005	51980

syndrome measurements can start before the decoder finishes execution. This optimisation is crucial in modern QEC workflows, including recent experiments such as those performed by Google [9], that can perform up to millions of QEC cycles a second.

This optimisation is performed using the `xz-propagation` pipeline. When applied to a number of consecutive QEC cycles, the pipeline first converts each corrective gate to its XZ gadget representation, then propagates these corrections to the end with a `xz-commute` pass, and finally merges the corrections. This method automatically modifies the results of later syndrome measurement. It also synthesises a function to combine the results of decoding cycles, without any input from the user.

As a prototypical case, consider the smallest error correction code capable of correcting a single qubit error: the perfect 5-qubit code [14]. The implementation in the dynamic gate IR encodes all three phases of the error correction cycle, including the classical decoder. Each phase is visually distinct within the IR, and when the pipeline is run on two consecutive cycles, it can be easily verified by inspecting the generated IR that only one correction phase remains.

Table 2 analyses the size of IR required to represent this QEC code within the dynamic gate IR. It demonstrates that the `xz-propagation` pass reduces the number of quantum operations, at the cost of a small increase in the number of classical operations.

We further ran our pipeline on IR representing larger numbers of consecutive QEC cycles, measuring the duration of each compiler pass in the `xz-propagation` pipeline. The results, in Figure 7, provide evidence that the `xz-propagation` pipeline runs in linear time with respect to the number of operations in the input.

Although we only consider correction cycles for a QEC code above, and not any logical gates, the `xz-propagation` pipeline will commute corrective gates past transversal (Clifford) logical gates. We therefore conjecture that this pipeline could form part of a compiler which converts a circuit to its fault tolerant implementation.

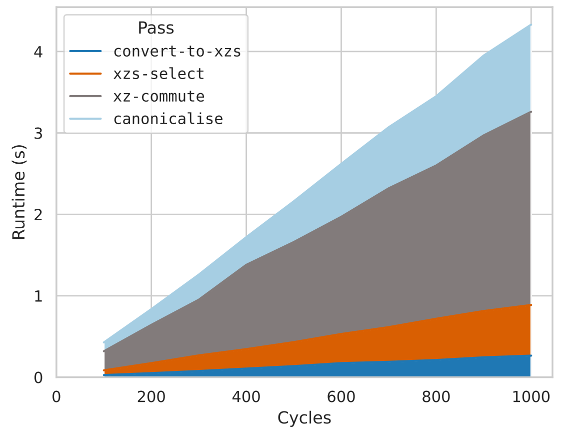


Figure 7. Duration of each pass of the `xz-propagation` pipeline when run on a varying number of cycles of the 5-qubit perfect QEC code. Each pass runs in linear time with respect to the input size.

5.3 Measurement-Based Quantum Computing

An alternative to the circuit model of quantum computing is given by measurement-based quantum computing (MBQC) [24]. This way of implementing unitary evolution works by entangling the input qubits with fresh auxiliary qubits; the entire computation is then driven by performing measurements. As quantum measurement is probabilistic, the computation is non-deterministic. The operation can be made deterministic (unitary) again by applying corrective Pauli gates to the output qubits, conditioned on the classical outputs of the measurements.

The measurement calculus [6] gives a syntax for MBQC programs. The normal form for MBQC programs is given by *measurement patterns*. These first allocate new qubits in the $|+\rangle = \mathrm{H}\,|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ state, entangle them (with controlled-Z gates), perform XY-plane measurements (measurements in the basis $\frac{1}{\sqrt{2}}(|0\rangle \pm e^{i\theta}\,|1\rangle)$ for some angle θ), and finally correct with Pauli gates conditioned by a linear combination of the measurement results. Consider a generic rotation, given by the unitary $\mathrm{R_X}(-\phi)\mathrm{R_Z}(-\theta)\mathrm{R_X}(-\lambda)$:

$$Z_5^{s_1+s_3} X_5^{s_2+s_4} [M_4^{\phi}]^{s_1+s_3} [M_3^{\theta}]^{s_2} [M_2^{\lambda}]^{s_1} M_1^0 E_{4,5} E_{3,4} E_{2,3} E_{1,2}$$

Each operation in this pattern acts on the qubits in its subscript (with qubit 1 being the input qubit and qubit 5 being the output qubit), and s_n corresponds to the classical outcome of measuring qubit n (which is unambiguous as each qubit is measured at most once). Each $E_{n,m}$ is a CZ gate on

Listing 2 An MBQC program that is equivalent to the generic rotation $R_X(-\phi)R_Z(-\theta)R_X(-\lambda)$ in our dynamic gate IR, with input qubit `%q1` and output `%q5_2` and angles ϕ, θ, and λ given by `%phi`, `%theta`, and `%lambda`. Both quantum and classical dependencies are represented explicitly as SSA-values.

```
%q2 = qubit.alloc<#qubit.plus>
%q3 = qubit.alloc<#qubit.plus>
%q4 = qubit.alloc<#qubit.plus>
%q5 = qubit.alloc<#qubit.plus>
%q1_1, %q2_1 = qssa.gate<#gate.cz> %q1, %q2
%q2_2, %q3_1 = qssa.gate<#gate.cz> %q2_1, %q3
%q3_2, %q4_1 = qssa.gate<#gate.cz> %q3_1, %q4
%q4_2, %q5_1 = qssa.gate<#gate.cz> %q4_1, %q5
%s1   = qssa.measure<#measurement.xy<0>> %q1_1
%a2   = angle.cond_negate %s_1, %lambda
%m2   = measurement.dyn_xy<%a2>
%s_2  = qssa.dyn_measure<%m2> %q2_2
%a3   = angle.cond_negate %s_2, %theta
%m3   = measurement.dyn_xy<%a3>
%s_3  = qssa.dyn_measure<%m3> %q3_2
%z    = arith.xori %s_1, %s_3 : i1
%a4   = angle.cond_negate %z, %phi
%m4   = measurement.dyn_xy<%a4>
%s_4  = qssa.dyn_measure<%m4> %q4_2
%x    = arith.xori %s_2, %s_4 : i1
%g    = gate.xz %x, %z
%q5_2 = qssa.dyn_gate<%g> %q5_1
```

qubits n and m, each $[M_n^\psi]^x$ is an XY-plane measurement of qubit n with angle $(-1)^x\psi$, and each X_n^x or Z_n^x is a Pauli X or Z gate, classically conditioned on the boolean x.

This pattern is in the standard CME (Correction-Measurement-Entanglement) form, where entanglement operations come before measurements, which in turn come before corrective gates. Any well-formed pattern can be converted to this form by a standardisation procedure [2].

The dynamic gate model can be regarded as an alternative syntax for MBQC programs based on SSA. The operation `measurement.dyn_xy` accommodates the dynamic angle on the measurement operations by taking an angle as operand and generating a measurement value representing the corresponding XY measurement. Listing 2 translates the pattern above into the IR. Let us stress that the conditional negation of the angles in each measurement operation is explicated into an operation, and that a single dynamic XZ gadget performs the conditional X and Z operators.

The IR is not only flexible enough to represent both circuit-based programs and MBQC programs, it can also convert from the former to the latter via local rewrites. The transformation takes as input a quantum circuit consisting of CZ and $J(\theta)$ gates, which form a universal gate set. The generic rotation unitary $R_X(-\phi)R_Z(-\theta)R_X(-\lambda)$ of our running example is given by the circuit:

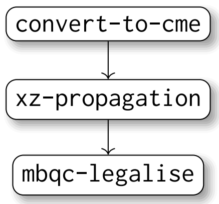
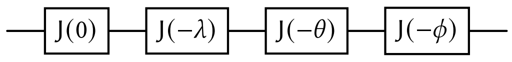
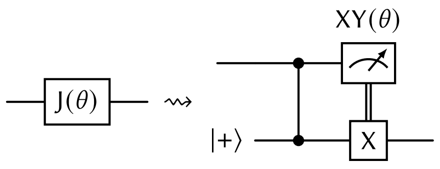


Figure 8. A compiler pipeline to convert CZ/J circuits to MBQC programs within the dynamic gate IR.



To allow the angles λ, θ, and ϕ to be parameters to the circuit, the operation `gate.dyn_j` encodes the J gates dynamically in the IR by taking a single angle operand.

As MBQC programs already permit CZ gates, the next step is to convert each J gate to a measurement pattern. The following rewrite does this:



where the measurement is given by a dynamic XY-plane measurement and the conditioned X gate is given by a dynamic selection between an X gate and an identity gate. The pass `convert-to-cme` performs this pattern rewrite.

After this transformation, we already have a valid measurement pattern, but it is not necessarily a CME pattern yet. After all, converting composed J gates adds a corrective Pauli gate in the middle of the program, before other CZ gates and XY-plane measurements. The `xz-propagation` pass standardises the pattern by pushing the corrective Pauli gates to the end of the program (and then fusing them). This process introduces corrective Z gates via the commutation rules for CZ gates, and conditionally negates angles via the commutation rules for XY-plane measurements.

Note that this optimisation does not need an extra "signal shifting" operation. Previous standardisation procedures for the measurement calculus needed signal shifting to track the negations of the classical measurement results. The explicit classical dataflow in our IR means this is no longer required.

The translation finishes with a pass `mbqc-legalise` that reorders the operations to result in a valid CME pattern. This pass does not change any dataflow between operations, and so leaves the semantics unchanged. Instead, it simply moves the allocations to the beginning of the program, leaves any CZ gates in the middle of the program, and moves any dynamic corrective gates to the end of the program.

Figure 8 summarises the full pipeline that converts CZ/J circuits to MBQC programs. If you apply this pipeline to the circuit J(θ) that represents the generic rotation, then you get the program in Listing 2, as desired.

6 Related and future work

In this section, we analyse some of the differences between the IR presented in this paper with other previously introduced quantum IRs. We then review other work that could form the basis for different gadgets, or that we otherwise believe could be represented well by the dynamic gate framework. We follow this discussion with some further suggestions of future work and potential additions to the IR.

6.1 Related quantum intermediate representations

There exist two dominant textual representations for quantum programs: OpenQASM and QIR. OpenQASM [5] closely corresponds to the representation of quantum programs in the Qiskit [11] quantum toolkit. Version 2 of OpenQASM was restricted to quantum circuits with a fixed number of qubits, though version 3 has added a limited ability to represent classical computations. Conversely, QIR [23] is built on top of LLVM [15], and inherits all its classical functionality, but quantum gates are applied via opaque function calls with little interaction between the classical and quantum components of the program. In both these formats, qubits have reference semantics, unlike the value semantics common in modern compilers.

Other quantum IRs have been proposed within the MLIR framework. McCaskey and Nguyen [18] introduce an MLIR dialect for representing programs, leveraging the tooling present in MLIR to define a translation down to QIR, while remaining expressive enough to provide lowerings from other toolkits to this dialect. Like QIR, qubits in this dialect are given by reference semantics. The first MLIR quantum dialect using value semantics was QSSA [20]. QIRO [10] introduces parallel quantum dialects; one with reference-semantics for qubits and one with value-semantics, leveraging that linearity analysis is easier within the non-SSA dialect, and optimisations are easier within the SSA dialect.

Quantinuum recently introduced tket 2, extending tket 1 [27] to hybrid quantum-classical programs. It is built on HUGR [13], a custom built library for creating quantum-classical representations. A possible avenue for future work is looking at the feasibility of introducing the framework, operations, and accompanying translations to this library.

6.2 Quantum Gadgets for Optimisation

A range of quantum gadgets already exist in the literature. Phase polynomials [1] are a gadget for representing combinations of CX and T gates, primarily used to reduce the T-count of a circuit.

The ZX calculus [3] is a graph-based method of representing linear maps. It introduces quantum gadgets called red and green spiders, and it has been shown these gadgets can be used to optimise quantum circuits [12]. As future work it would be interesting to explore adding ZX spiders to the dynamic gate model, which is nontrivial because the ZX calculus making no distinction between the inputs and outputs of spiders.

The tket [27] compiler makes use of two quantum gadgets inspired by the ZX calculus: phase gadgets, and their generalisation Pauli gadgets. These gadgets are used like the XZ gadgets in this paper: subcircuits can be represented by phase gadgets, which can be optimised before translating back to a traditional circuit representation.

6.3 Other Future Directions

The current work does not discuss subprocedures or subcircuits. While these can be represented with MLIR's `func` dialect, a more complete solution would be able to interact with conversions between reference and value-based semantics, Pauli propagation, and inverting unitaries. An interesting direction for further research is to explore implementating a `subcircuit` operation producing a gate value from a circuit, which could then be applied dynamically.

A different direction to extend the current work is to add quantum data types other than qubits. The addition of quantum registers has been explored in other quantum IRs, with QIRO and QSSA both introducing custom quantum register types with operations for manipulating them. Note that the MLIR `tensor` type is immutable, making it difficult to use on qubits with value-semantics. Other work [29, 32] has explored adding higher-level data types to quantum languages, which will be necessary to represent in future quantum IRs.

Finally, while quantum noise channels were used to motivate dynamic gates, their use within a noise-aware compilation pipeline remains relatively unexplored. Natively representing noise channels with dynamic gates should in particular work for the classical simulation of noisy circuits, where it should be possible to optimise the noisy circuit before instantiating the noise for a particular sample, much like in our randomised compilation case study.

7 Conclusion

This paper presents the dynamic gate model for hybrid quantum-classical computing (Section 2), along with an intermediate representation based on this model (Section 4). The key strength of the model is its flexibility; many seemingly unrelated techniques in quantum computing have a natural representation within this IR, which we explored through case studies in Section 5.

We further introduced the XZ and XZS gadgets as part of this framework (Section 4.1), equipped with a selection of optimisations for working with them, and demonstrated the effectiveness of these optimisations for performing various transformations within each case study.

Acknowledgments

The authors would like to thank Chris Vasiladiotis for feedback on earlier drafts of this paper. This work was funded by EPSRC Grant EP/W032635/1, Robust and Reliable Quantum Computing

References

[1] Matthew Amy, Dmitri Maslov, and Michele Mosca. 2014. Polynomial-Time T-Depth Optimization of Clifford+T Circuits Via Matroid Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 10 (Oct. 2014), 1476–1489. doi:10.1109/TCAD.2014.2341953

[2] Anne Broadbent and Elham Kashefi. 2009. Parallelizing Quantum Circuits. *Theoretical Computer Science* 410, 26 (June 2009), 2489–2510. doi:10.1016/j.tcs.2008.12.046

[3] Bob Coecke and Ross Duncan. 2011. Interacting Quantum Observables: Categorical Algebra and Diagrammatics. *New Journal of Physics* 13, 4 (April 2011), 043016. doi:10.1088/1367-2630/13/4/043016

[4] A. D. Corcoles, M. Takita, K. Inoue, S. Lekuch, Z. K. Minev, J. M. Chow, and J. M. Gambetta. 2021. Exploiting Dynamic Quantum Circuits in a Quantum Algorithm with Superconducting Qubits. *Physical Review Letters* 127 (2021), 100501.

[5] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. 2022. OpenQASM 3: A Broader and Deeper Quantum Assembly Language. *ACM Transactions on Quantum Computing* 3, 3 (Sept. 2022), 12:1–12:50. doi:10.1145/3505636

[6] Vincent Danos, Elham Kashefi, and Prakash Panangaden. 2007. The Measurement Calculus. *J. ACM* 54, 2 (April 2007), 8–es. doi:10.1145/1219092.1219096

[7] D. P. Divincenzo. 1997. Mesoscopic Electron Transport. Kluwer, Chapter Topics in quantum computers, 657–677.

[8] Mathieu Fehr, Michel Weber, Christian Ulmann, Alexandre Lopoukhine, Martin Paul Lücke, Théo Degioanni, Christos Vasiladiotis, Michel Steuwer, and Tobias Grosser. 2025. xDSL: Sidekick Compilation for SSA-Based Compilers. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25)*. Association for Computing Machinery, New York, NY, USA, 179–192. doi:10.1145/3696443.3708945

[9] Google Quantum AI and Collaborators. 2025. Quantum Error Correction below the Surface Code Threshold. *Nature* 638, 8052 (Feb. 2025), 920–926. doi:10.1038/s41586-024-08449-y

[10] David Ittah, Thomas Häner, Vadym Kliuchnikov, and Torsten Hoefler. 2022. QIRO: A Static Single Assignment-based Quantum Program Representation for Optimization. *ACM Transactions on Quantum Computing* 3, 3 (Sept. 2022), 1–32. doi:10.1145/3491247

[11] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. 2024. Quantum Computing with Qiskit. doi:10.48550/arXiv.2405.08810 arXiv:2405.08810 [quant-ph]

[12] Aleks Kissinger and John van de Wetering. 2020. Reducing the Number of Non-Clifford Gates in Quantum Circuits. *Physical Review A* 102, 2 (Aug. 2020), 022406. doi:10.1103/PhysRevA.102.022406

[13] Mark Koch, Agustín Borgna, Seyon Sivarajah, Alan Lawrence, Alec Edgington, Douglas Wilson, Ross Duncan, Craig Roy, Luca Mondada, and Lukas Heidemann. [n. d.]. HUGR: A Quantum-Classical Intermediate Representation. *Planqc 2025* ([n. d.]).

[14] Raymond Laflamme, Cesar Miquel, Juan Pablo Paz, and Wojciech Hubert Zurek. 1996. Perfect Quantum Error Correcting Code. *Physical Review Letters* 77, 1 (July 1996), 198–201. doi:10.1103/PhysRevLett.77.198

[15] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. doi:10.1109/CGO.2004.1281665

[16] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308

[17] Rikus R. le Roux, George van Schoor, and Pieter A. van Vuuren. 2014. A Survey on Reducing Reconfiguration Cost: Reconfigurable PID Control as a Special Case. *IFAC Proceedings Volumes* 47, 3 (Jan. 2014), 1320–1330. doi:10.3182/20140824-6-ZA-1003.01544

[18] A. McCaskey and T. Nguyen. 2021. A MLIR Dialect for Quantum Assembly Languages. In *IEEE International Conference on Quantum Computing and Engineering*. 255–264.

[19] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information* (10th anniversary edition ed.). Cambridge university press, Cambridge.

[20] Anurudh Peduri, Siddharth Bhat, and Tobias Grosser. 2022. QSSA: An SSA-based IR for Quantum Computing. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. ACM, Seoul South Korea, 2–14. doi:10.1145/3497776.3517772

[21] C. Piveteau and D. Sutter. 2023. Circuit Knitting with Classical Communication. *IEEE Transactions on Information Theory* (2023), 2734–2745.

[22] J. Preskill. 2018. Quantum Computing in the NISQ Era and Beyond. *Quantum* 2 (2018), 79.

[23] QIR Alliance. 2021. *QIR Specification.*

[24] Robert Raussendorf and Hans J. Briegel. 2001. A One-Way Quantum Computer. *Physical Review Letters* 86, 22 (May 2001), 5188–5191. doi:10.1103/PhysRevLett.86.5188

[25] Joschka Roffe. 2019. Quantum Error Correction: An Introductory Guide. *Contemporary Physics* (July 2019).

[26] B. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principle of Programming Languages*. 12–27.

[27] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. 2020. TKET: A Retargetable Compiler for NISQ Devices. *Quantum Science and Technology* 6 (Nov. 2020). doi:10.1088/2058-9565/ab8e92

[28] Shigeyuki Takano and Hideharu Amano. 2022. Reconfiguration Cost for Reconfigurable Computing Architectures. In *2022 23rd ACIS International Summer Virtual Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD-Summer)*. 62–67. doi:10.1109/SNPD-Summer57817.2022.00019

[29] Matan Vax, Peleg Emanuel, Eyal Cornfeld, Israel Reichental, Ori Opher, Ori Roth, Tal Michaeli, Lior Preminger, Lior Gazit, Amir Naveh, and Yehuda Naveh. 2025. Qmod: Expressive High-Level Quantum Modeling. doi:10.48550/arXiv.2502.19368 arXiv:2502.19368 [quant-ph]

[30] Joel J. Wallman and Joseph Emerson. 2016. Noise Tailoring for Scalable Quantum Computation via Randomized Compiling. *Physical Review A* 94, 5 (Nov. 2016), 052325. doi:10.1103/PhysRevA.94.052325

[31] N. Yanofsky and M. A. Mannucci. 2008. *Quantum Computing for Computer Scientists.* Cambridge University Press.

[32] Charles Yuan and Michael Carbin. 2022. Tower: Data Structures in Quantum Superposition. *Tower: Data Structures in Quantum Superposition* 6, OOPSLA2 (Oct. 2022), 134:259–134:288. doi:10.1145/3563297