Greedy Reliability Aware Qubit Routing

Kevin Volkel

Abstract

With the error rates of NISQ quantum machines being on the order of 10^{-2} to 10^{-3} for 2 qubit CNOT gates, along with high variability in these error rates across links on the same machine, software passes within the compiler that map logical qubits to compatible physical qubits in a reliability-aware manner have the potential of increasing the reliability of NISQ programs. Furthermore, the problem of routing logical qubits to physical qubits throughout the program in some optimal manner is NP-complete, so it is important to consider low complexity greedy approaches in order to have tractable compilers for programs with large numbers of qubits and gates.

This project studies several reliability aware qubit routing policies implemented on top of the *SabreSwap* algorithm within *Qiskit* in order to investigate the impact that being aware of the reliability variation for CNOT gates, single qubit gates, and readout errors can have on the success of a quantum program when accounted for within a routing pass. Because at its core *SabreSwap* is a $O(N^{2.5})$ complexity algorithm, the resulting reliability aware mapping policies are of identical complexity, making it a more scalable solution than approaches like A* and Satisfiability Modulo Theory (SMT) solvers for large programs. In terms of program success improvements, this project shows that reliability aware approaches for both mapping and routing routinely provide improvements over techniques that just aim to reduce the overall number of gates and circuit depth. By evaluating 14 compiler passes over 3 IBM quantum machines and 5 benchmarks, this project shows that the best compiler pass depends on both logical qubit interaction and machine dependent characteristics. Machines with lower relaxation times are more sensitive to decoherence, so passes that consider this perform best. While programs with complex circuit communication and large numbers of single qubit gates tend to prefer passes that consider SWAP path reliability and single qubit operation error rates.

1 Introduction

Due to the error rate of NISQ quantum computing devices, where CNOT gate error rates can be as high as 10^{-2}, methods to mitigate errors that occur during computation and measurements are needed. This can be managed by many different techinques, from quantum error correction codes that use a form of quantum redundancy to detect and correct errors, to passes within the compiler that accounts for reliability information when making decisions on what hardware qubits single qubit, multi qubit, and readout operations should be preformed on. With the scale of NISQ devices being limited to the order of 10 to 100 physical qubits, quantum error correction approaches like surface codes that need at least 9 total physical qubits and a nearest neighbor connectivity to sustain an error tolerant logical encoding are not likely to be tractable for current machines [5]. However, it has been shown that methods to *mitigate* rather than *correct* errors can be useful to increasing program success rate. For example, Tannu *et al.* have shown that they are able to increase success rates by biasing the output of a program to a more favorable state for measurement [10]. While other approaches have leveraged reliability information of gate error rates to adapt

the mapping and routing of qubits to increase the success of programs by making sure that the program does not perform operations or readouts on high error rate qubits [7, 8, 11].

A compiler for a quantum machine has several general phases in order to make sure that the quantum program is in a form compatible with the underlying hardware. First, it should select an initial allocation of physical qubits to the logical program qubits, this is known as a *mapping pass* in *Qiskit's* compiler framework [2]. Second, the initial mapping, along with the program, are checked against the connectivity map of the quantum machine. If there are any multi-qubit gate operations that are not compatible with the initial layout, then SWAP operations are necessary in order to move the quantum states of the gate inputs to appropriate locations so the operation can be performed. The problem of choosing SWAPs that lead to an optimized circuit under some metric has been shown to be NP-complete [9]. However, there have been approaches that try to efficiently explore the space of SWAP possibilities by using $A*$ state space searches [12, 10], or by using methods based Satisfiability Modulo Theory (SMT) solvers [8, 7]. Though, it has been shown that $A*$ searches still incur exponential time complexity as the algorithms searches for an exact solution over a large combinatorical space. Also, SMT solvers have been shown to be tractable in the regime of program sizes under 32 qubits, with an SMT solution for reliability aware mapping and routing taking up to 2.7 hours for a 32 qubit machine [7].

Li *et al.* acknowledge the lack of scalability of $A*$ state space explorations for large programs [6], and instead take a greedy approach to the mapping and routing problem in order to develop a scalable solution with computation complexity of just $O(N^{2.5})$ compared to the complexity of $O(exp(N))$ exhibited by the $A*$ approach of [12] and [11]. However, the cost functions associated with scoring a given SWAP configuration are only based on minimizing the number of gates and the depth of the resulting circuit after mapping and routing. Hence, the algorithm is completely oblivious to the underlying reliability of the paths used to transport states using SWAP gates, and the reliability of measurement operations that will be performed on logical qubits once they reach their final physical location. Since this algorithm is efficient in time, and and it is implemented within *Qiskit's* open source compiler infrastructure, it is a good candidate for studying extensions to SWAP scoring heuristics that account for different vectors of errors, e.g. CNOT error rate, readout error rate, single gate error rate, decoherence time. For this reason, this project explores several different policies for taking into account reliability information about a quantum machine back-end provided by IBM.

In this project, the error rates for different operations is explored for different quantum machines in order to identify the largest sources of errors and the variability of those error rates across a given machine. Using this information, several independent heuristics are developed in order to target one or a set of these sources for error. They are first evaluated separately to understand which source of error is most important to target for a given machine, and then they are evaluated together to evaluate whether a combined approach is better than an individual approach. For example, the impacts of noise adaptive initial mappings are separated from the impacts of noise adaptive routing, along with the impacts of considering different sources of error such as SWAP gates and measurements. While developing these heuristics, analysis is carried out to ensure that emitted scores for a given SWAP candidate will guarantee that the greedy algorithm will converge. That is, this project shows that existing techniques and data structures may not be readily usable in this specific greedy approach for routing, as the algorithm may never find a sequence of scores that allows the compiler to converge and agree on a final mapping to perform a CNOT gate. For example, it is shown that if the 2Q reliability matrix of Murali in [8] is

used to express the asymmetric nature of the reliability of a path between two qubits, then *SabreSwap* will never converge in certain settings. Likewise, when taking into account a mixture of single qubit error sources like single qubit gates and measurement errors, to mix these reliability scores with the reliability scores for paths of CNOT gates special care needs to be taken to ensure convergence.

Overall, this paper explores 14 different compiler passes (7 with newly added reliability-aware scoring heuristics) across 3 different quantum machines and 5 different programs consisting of 3 distinct program classes: adder, BV, and QFT. By studying different program classes that each have different program connectivity of logical qubits, we are able to determine whether or not that program characteristics influence the choice of the best reliability aware routing strategy. Results show that machines with lower relaxation times and thus are more sensitive to decoherence errors tend to prefer the compiler passes that take into account decoherence as a reliability metric. Furthermore, the evaluation shows that for quantum programs with simple connectivity patterns, passes that take into account not only the reliability of the current layer of CNOTs being routed, but also the following layer are preferred. Lastly, results show that on average, the highest ranking pass in terms of program success rate factors in the reliability of SWAP paths, single qubit, and measurement operations.

2 Background

2.1 Qubit Mapping and Routing

In this project it is assumed that the underlying technology used to run quantum programs is the IBM superconducting transmon technology. For this type of technology, the connectivity amongst the physical qubits is typically limited such that there is only nearest neighbor connections. This means that two qubit gates such as CNOTs are limited to physical qubits that are immediate neighbors of one another, and therefore it is possible that the hardware cannot support all possible logical qubit connections within the program. This issue requires solutions to two problems. One problem is picking an initial mapping for all logical qubits in the program such that each is assigned a physical qubit. The second problem is ensuring that all logical operations in the program are performed on physical qubits that support those operations. In this project it is assumed that SWAP gates are implemented with 3 CNOT gates as shown in Figure 1 since the primitive 2-qubit gates on the IBM machines are CNOT gates.

Take for example Figure 1, where a simple 4 logical qubit program is mapped to a 4 qubit machine. The 4 logical qubits Q_{Lx} are mapped initially to some physical qubits indicated by the change in the circuit from the first frame to the second frame. The original logical circuit is constructed on this new initial mapping, and during a routing pass it is seen that the logical CNOT operation that uses Q_{L0} as the control and Q_{L3} as the target are not in a compatible position since Q_{L0} resides at Q_{P3} and Q_{L3} resides at Q_{P0}. Therefore, a swap operation is needed. After swapping the states on Q_{P2} and Q_{P3}, Q_{L0} state is now in a compatible position for the CNOT operation to be performed by moving to physical qubit Q_{P2}.

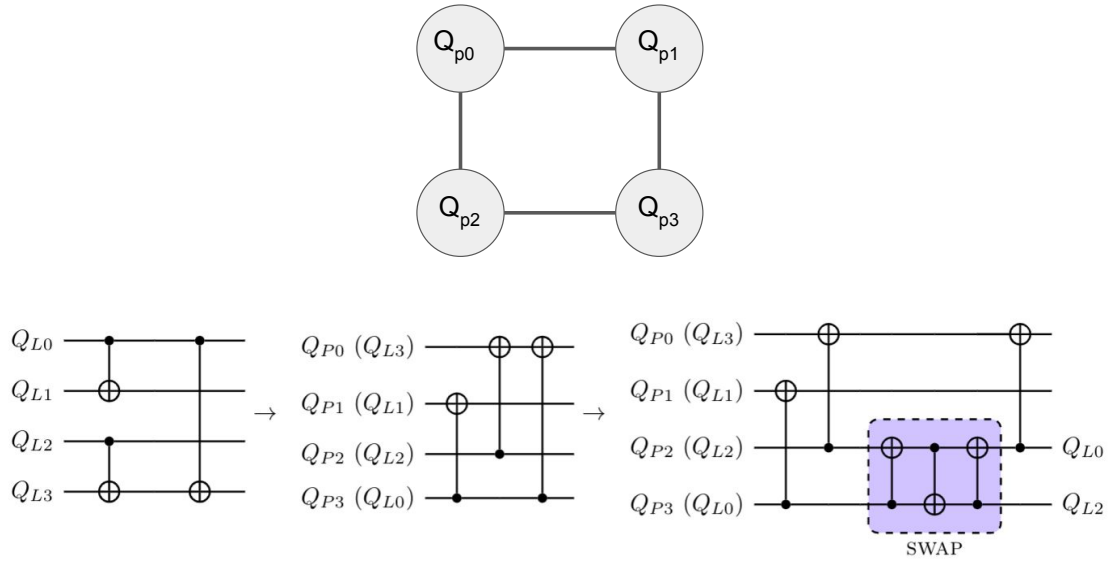

Fig. 1: Example of the difference between mapping and logical qubit routing. Mapping performs an initial association between the set of logical qubits Q_L and the set of physical qubits Q_P. This amounts to a wiring reorder in the circuit as shown as the transition from the first to second circuit. After mapping, routing is still needed to transform the second to the third circuit so that the quantum states are in a compatible physical position.

2.2 SabreSwap

Since *Sabre* is the base routing algorithm that this project builds upon, a description of its original implementation is given. The general algorithm described by Li *et al* consists of a strategy for both generating an initial map, and for generating the required SWAP gates needed for the physical realization of qubit interactions within the program. The mapping and routing are generated together by performing 3 back-to-back passes of the same routing algorithm over the circuit, with each building upon the previous. For example, the first pass assumes a random initial mapping and performs the routing needed for the program. Once the end of the circuit is reached, the original circuit is reversed and the end of the circuit is used as the starting point. The resulting positions of logical qubits at the end of the first pass are used as the initial map for the second pass. The second pass is performed starting from the end and going to the beginning and repeating the same process as the first pass, leading to an initial mapping for the beginning of the circuit. Finally, the third pass performs the final routing using the final initial mapping obtained after the second pass. In *Qiskit's* implementation, the initial map construction is done separately from the final routing pass and allows for different routing passes to take the place of *Sabre's* routing algorithm to allow for more flexibility.

Algorithm 1 below is *Sabre's* routing algorithm [6]. The algorithm has four main components: a *front layer* (F) data structure that holds gates that need to be routed, a *distance matrix* (D) containing the length of the shortest (in number of hops in coupling graph) path between all physical qubits in the machine, a layout data structure (π) that tracks current logical to physical qubit mapping, and a *cost heuristic* which is used to quantify

the impact of a given SWAP. The algorithm starts by filling the front layer with two-qubit gates that need to be routed because they are not nearest neighbors on the machine. While looking for two-qubit gates that need to be routed, all single qubit operations and two-qubit operations that are already satisfied are "executed". Executed in this context means that we do not need to consider these gates for routing, and they can be immediately executed on the current physical layout of the program. All executable gates are removed from consideration until no more can be found, and after this, the process of moving the quantum state associated with two logical qubits that need to be neighbors is carried out.

Instead of considering all possible SWAPs that can be applied to each physical qubit, *Sabre* only considers those SWAPs that can help solve the problem of routing the logical qubits in the front layer. Hence, only physical qubits that have a logical qubit in the process of being routed are considered. Once these eligible qubits are located, all possible swap operations that can be applied to them are added to a list called *SWAP_candidates*. For each SWAP candidate in the list, a temporary logical qubit to physical qubit mapping is made by applying the SWAP under consideration to the current layout.

With this temporary layout, a cost calculation is performed with a heuristic function. The cost heuristic involves calculating the total remaining minimum distances for all gates in the front layer using the distance matrix. *Sabre's* heuristic cost function (Equation 1) uses the distance matrix to make sure that the new layout is consistently being pushed towards a configuration that results in the shortest possible path between logical qubits in the front layer after the SWAP is applied. The heuristic also tries to balance gate parallelism versus the number of gates added to the circuit by using a *decay factor* that makes sure that the same logical qubit is not repetitively in a SWAP which leads to serialization and larger circuit depth. Lastly, the heuristic can also be configured to not just consider gates in the front layer F, but to also consider gates in an extended layer E. This helps choose a SWAP that may be useful when routing gates in E when the time comes later when they are added to F. Therefore, the total heuristic averages these two together, and the extended layer E is limited by a weight W so it does not overpower F. After the heuristic cost is calculated for all candidate SWAPs, the lowest cost candidate is chosen and is applied to the current layout to update the logical to physical qubit mapping. The process is repeated until all logical qubits have been satisfied in the program.

$$Heuristic\_Cost(F, SWAP, D, \pi, E) = max(decay(SWAP.q_1), decay(SWAP.q_2))\{ \\ \frac{1}{|F|} \sum_{gate \in F} D[\pi(gate.q_1)][\pi(gate.q_2)] + \frac{W}{|E|} \sum_{gate \in E} D[\pi(gate.q_1)][\pi(gate.q_2)]\} \quad (1)$$

Through simple complexity analysis it can be shown that *SabreSwap* has a computational complexity of $O(N^{2.5})$ with N being the number physical qubits. At worst, all qubits of the machine can have logical states in the process of being routed. Thus, the heuristic calculation takes $O(N)$ iterations to sum over the distance matrix. Furthermore, the SWAP list will grow proportionally with the number of qubits being routed, so the heuristic can run up to $O(N)$ times. For a grid layout typically used in transmon machines, the maximum number of SWAPS to move two qubits together is the diameter of the grid, which scales $O(\sqrt{N})$. Therefore, total worst-case complexity is $O(N^{2.5})$, per logical qubit in the program.

Algorithm 1: *SabreSwap* [6]

Input: Front Layer F, Current Mapping π, Distance Matrix D,
Circuit DAG,Circuit New_DAG, Chip Coupling Graph $G(V, E)$
Output: Inserted SWAPS, Final Map π_f

```
/* assume F filled with independent gates from the front of the DAG
   */
while F ≠ ∅ do
    execution_list = ∅;
    for gate g in F do
        if π satisfies g.qubits then
            execution_list.append(g);
        end
    end
    if execution_list ≠ ∅ then
        for gate e in execution_list do
            /* Get dependent gates                                  */
            successors = DAG.get_successors(e);
            F.remove(e);
            New_DAG.apply(e)
            for gate s in successors do
                if s.dependencies resolved then
                    /* Gate s is next to be routed                  */
                    F.append(s)
                end
            end
        end
        /* Go back and check if more can be executed                */
        Continue;
    else
        /* Noting to execute, need to route                         */
        score_list = [ ];
        /* Get viable SWAPs of F                                    */
        SWAP_candidates=Obtain_SWAPs(F,G);
        for SWAP in SWAP_candidates do
            π_temp = π.copy()
            π_temp.update(SWAP)
            score_list[SWAP] = Heuristic_Cost(F,SWAP,D,π_temp, E)
        end
        best_SWAP=score_list.index_of(min(score_list))
        New_DAG.apply(best_SWAP)
    end
end
```

3 Error Rates in NISQ Machines

When developing solutions for a reliability aware routing pass, it is important to observe the variability in error rates for different gates to understand where the most opportunity

lies. Figure 2 shows the error rate for all native operations on the *ibmq_16_melbourne* machine. This plot visualizes the calibration data that IBM makes publicly available for its quantum machines. Note that the vertical axis is in log scale, and the horizontal axis indicates a unique physical qubit on the machine. The first characteristic we are interested in is how much variation there is in error rates, since if there was not any, there would be no opportunity allowing the compiler to be aware of reliability information. However, this figure shows that there is considerable variation across all of the native operations for this machine. Readout error rates encompass a wide range of values, from a minimum value of 2.5% to a maximum value of 18.4%, a 7.36× increase. Likewise, for CNOT error rates, the minimum error rate for a link was measured to be 1.46% with the highest error rate being a 3.95× increase to 5.73%. Interestingly enough, the least error-prone link and the highest error prone link have a connection to the same physical qubit Q_{P1} which means that it cannot be assumed that CNOT error rates for all links of a given physical qubit are similar, and all links should be accounted for individually.

We can also see from this that CNOT error rates, along with readout errors are orders of magnitude larger than the error rates for single qubit U2 and U3 gates. This indicates that reliability aware routing should prioritize the reliability metrics that pertain to CNOT operations and measurements. Furthermore, it is interesting to note that there is not necessarily exact correlation in when readout errors are maximized compared to when CNOT error rates are maximized. For example, if the correlation is calculated between the average error across all CNOT links for a physical qubit and the readout error rate for that qubit, the value of 0.36 is obtained which indicates a weak absolute correlation for this instance of reliability data. This indicates that we need to take this readout error rate information into account separately from the CNOT information, and try to make the best decision at a given point in the routing process that balances the error rate of CNOT and measurement operations.

Prior works typically disregard single qubit errors [7, 11] when constructing a reliability aware routing and mapping. This seems to be a reasonable assumption, since even for the highest error rate single qubit U3 gates, the error rate is approximately 1 order of magnitude less than CNOT an measurement operations. Though, it would be reasonable to believe that

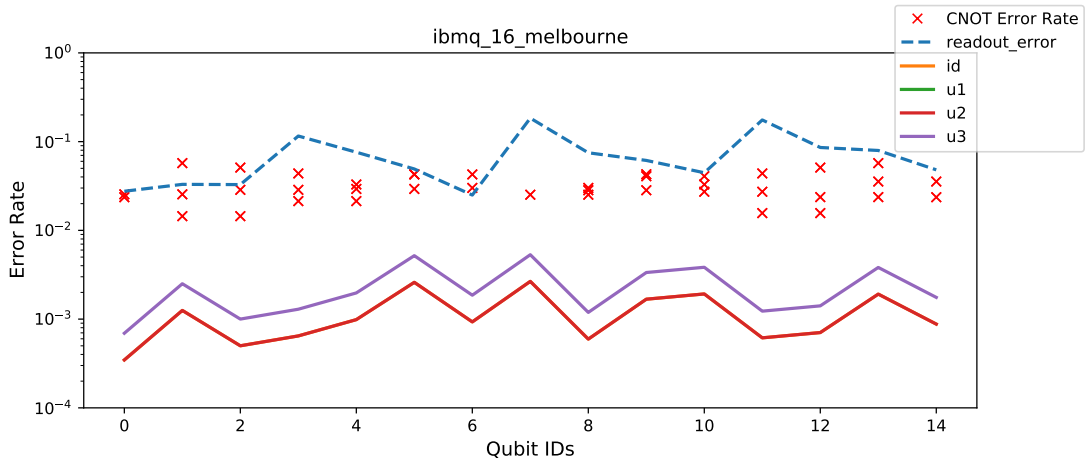

Fig. 2: Error rates as provided by IBM for all native gates and operations on the *ibmq_16_melbourne* machine.

the placement of single qubit gate execution could have impacts on the overall reliability since there is high variability in error rates and probability of success scales $O(\epsilon^t)$. ϵ is a probability of success for a single qubit gate, and t is the number of times that gate type is executed. Allow us assume the largest U3 error rate is 0.004, and the smallest to be $7 \cdot 10^{-4}$ (estimates from Figure 2). For the 8-qubit QFT program used as a benchmark, it has 100 single qubit operations, and so the probability of success for all these in the high error rate scenario will be $(1 - 0.004)^{100} = 66.97\%$ while for the lower error rate the success rate is $(1 - 7 \cdot 10^{-4})^{100} = 93.23\%$. Hence taking into account single qubit errors in this case could theoretically lead to a 1.4× more reliable program.

4 Reliability Aware Sabre Swap

Figure 3 shows an example of why *SabreSwap* is not sufficient on its own when crafting a reliability aware routing pass. There are two different paths that can be taken to move the state of qubit Q_{P0} to neighbor that of Q_{P2}. One is shorter geometrically, while the other is more reliable. The shorter path swaps the state of Q_{P1} and Q_{P2} and has probability of success $0.8^3 \cdot 0.80 = 41\%$. Recall a SWAP gate is 3 CNOT gates, so all have to succeed. On the other hand, the green path shows a path with larger success rate $0.95^3 \cdot 0.98^3 \cdot 0.95^3 \cdot 0.9 = 62\%$. Notice that a reliability aware router should also ensure the least reliable link is the one time use CNOT application. Thus, for reliability purposes we need to modify the scoring mechanism for *SabreSwap* in order to push the resulting solution towards more reliable paths rather than just shorter paths for the sake of minimizing gate count. The next sections go over the general transformation that needs to be done to the cost heuristic of Equation 1, and then goes into the details how framing the problem of reliable paths in this greedy approach needs to be done in order to assure convergence of the algorithm.

4.1 Reliability Aware Distance Matrix

The first step in transforming the *SabreSwap* algorithm to be aware of error rates is to change the information stored in the *distance matrix*, D. Instead of storing the shortest path length between two physical qubits, it should store a metric that reflects the probability of success along that path. One thing should be noted when considering probabilities and distances. Probabilities cannot be directly summed, which is what is required from Equation 1 and shortest path algorithms like Dijkstra. For example, given the set of probabilities $\{0.5, 0.5\}$ and another set of probabilities $\{0.3, 0.7\}$, clearly both sum to 1 but have different products, and the product is what we are interested in since we want a way to rank the probability of multiple independent events (gate applications) to have certain outcomes (success). However, if we take the *log* of a product of numbers, $log(ab) = log(a) + log(b)$, we can break the product of probabilities down into a sum of log-probabilities. Furthermore, $log(x)$ is a monotonic function, so the relative ordering of for a group of probabilities is preserved after applying the log function to each probability. However, Equation 1 wants to minimize the cost, and algorithms like Dijkstra and Floyd-Warshall want to look for the smallest path between nodes. So, one last transformation on the probability of success of a path needs to be done. This can be done by adding a negative sign $(-log(x))$ such that the function is always decreasing, and of course this is always positive since the probability of success can not be larger than 1. More formally, this process is described in Algorithm 2 below, which constructs a matrix R where each entry is the smallest negative log-probability of success between two nodes. This matrix is simply constructed by leveraging the negative-

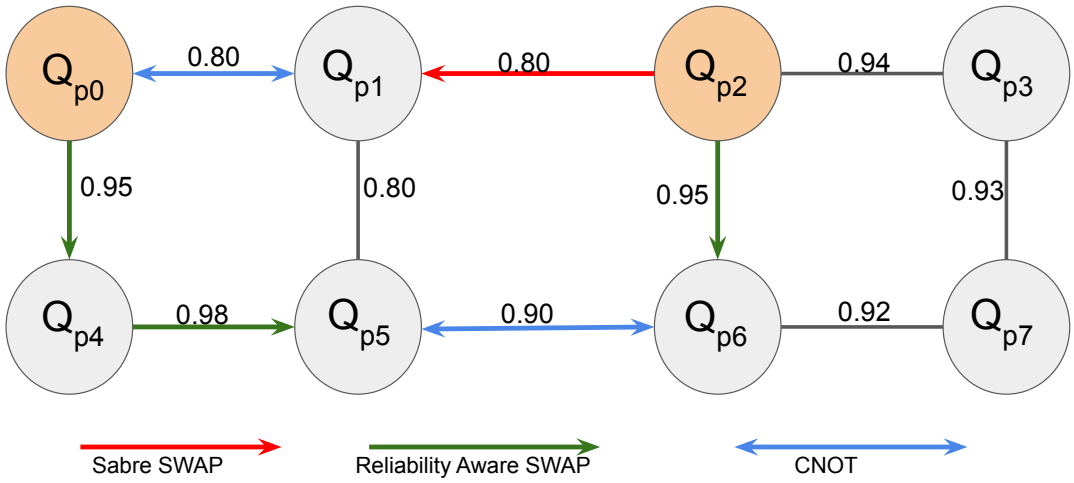

Fig. 3: Example of two different possible paths that can be taken when moving the logical state starting at Q_{P0} and Q_{P2}. *SabreSwap* favors taking the shorter path (red), while a path 2 hops larger can be more reliable (green). Labels indicate ***success rates*** for CNOT gates on those links between physical qubits.

log-probability of performing a SWAP on each CNOT link. By using this measure, the total success probability along a path is equivalent to just a summation of edge weights. Since the highest probability of success relates to minimizing negative-log-probabilities, shortest path algorithms like Dijkstra can be used to fill in R. The base reliability algorithm directly substitutes R for D in Equation 1 with no other changes. The matrix P in Algorithm 2 holds the set of physical qubits traversed along this shortest log path, and is needed for Section 4.2.

One should note that this matrix is different than the 2Q matrix explained by Murali in [8]. The matrix used in this work is the negative log probability of success for performing SWAPS on *all* edges between two nodes on the highest success probability path. It does not take into account that one of these links will however be used for the target CNOT gate we are trying to route. Murali *et al.* address this by instead of finding the highest success path of SWAPs between two qubits q_0 and q_1, it instead finds the highest success path to a neighbor of q_1 say, q_1' and then models the remaining link between q_1' and q_1 of having the success rate of a single CNOT gate. For example, using their methodology on the graph in Figure 3 ,the entry for qubits Q_{P2} and Q_{P5} will be to $R[2][5] = -\log\left(0.95^3 \cdot 0.9\right) = 0.11$. Also, note that their 2Q matrix is also not symmetric: $R[2][5] \neq R[5][2]$. On the other hand, this work will have a symmetric matrix, with $R[2][5] = -\log\left(0.9^3 \cdot 0.95^3\right) = 0.20$. The reason why Murali's 2Q matrix is not used directly in this algorithm is due to convergence issues and is expanded on in the next section.

Algorithm 2: Constructing Reliability Matrix R and Path Matrix P

Input: Chip Coupling Graph $G(V, E)$, Chip Noise Information N
Output: Reliability Matrix R, Shortest Path Matrix P
reliability_graph=∅
`/* Let any vertex in` G `and reliability_graph be equivalent to a physical qubit. */`
for *v in G* **do**
  if *v not in reliability_graph* **then**
    reliability_graph.add_vertex(v);
  end
  for v_n *in v.neighbors* **do**
    if *edge (v,*v_n*) in reliability_graph* **then**
      Continue;
    end
    CNOT_error_rate= $N.gate(v, v_n)$;
    neg_log_prob_SWAP = $-3 \cdot \log(1 - CNOT\_error\_rate)$;
    `/* Add edge with weight = neq_log_prob_SWAP */`
    reliability_graph.add_edge(v,v_n,neg_log_prob_SWAP);
  end
end
`/* Have graph with edges representing -log success rates of SWAPs, now create` R `and` P `*/`
$R = [\,][\,]$
for v_s *in reliability_graph* **do**
  for $v_d \neq v_s$ *in reliability_graph* **do**
    R[v_s][v_d] = reliability_graph.dijkstra(v_s,v_d).length();
    P[v_s][v_d] = reliability_graph.dijkstra(v_s,v_d).path();
  end
end

4.2 Ensuring Convergence

Convergence needs to be proven when employing a greedy algorithm so that at least some answer can be found. By replacing D with R in Equation 1 and always choosing the minimal sum of paths, eventually all logical qubits needing routing will reach neighboring physical qubits. This is because choosing SWAPs that increase distance between nodes also increases the number of links in the coupling graph that need to be traversed, and hence the increases the probability of error. That is, there is no case in which moving nodes further apart is more favorable. Another way to look at it is that SWAP reliability is encoded simply as an edge weight, and minimizing the sum of edge weights no matter what it represents, will bring logical qubits into neighboring proximity.

However, replacing D with R may lead to unwanted SWAP's being taken from a reliability perspective. For example, consider Figure 3, and let the logical qubit states to be routed be positioned at Q_{P4} and Q_{P2}. Now, when evaluating $SWAP(Q_{P4}, Q_{P5})$ and $SWAP(Q_{P2}, Q_{P6})$ using R, we get remaining distances $-\log\big(0.9^3 \cdot 0.95^3\big) = 0.20$ and $-\log\big(0.9^3 \cdot 0.98^3\big) = 0.16$ respectively. Thus we will take the latter SWAP as it minimizes the overall score. However, this is a lower success rate SWAP (0.95 success rate) versus

the other option which has a 0.98 success rate. So, we would like to take into account this asymmetry in the path so we can push the solution to choose the right combinations of movements.

Murali's 2Q matrix takes this into account in a way that lowers the cost of the links adjacent to physical qubits where the logical state resides down to individual CNOT error rates rather than SWAP error rates. This allows us to consider the path in both directions and choose the direction that leads to the highest probability SWAP being chosen. However, from Algorithm 1, we see that *Sabre* only considers cost on the new layout *after* applying the swap. This means that the SWAP to get us to a given layout is left out in costs calculations. Though, we can change Equation 1 to be (leaving out the next layer, and decay terms):

$$Heuristic\_Cost(F, SWAP, R, \pi) = -\log(P_{success}(SWAP)) + \sum_{gate \in F} R[\pi(gate.q_1)][\pi(gate.q_2)] \quad (2)$$

Finally, the problem with a 2Q matrix being used as R is that it downgrades links in a way that leads to an asymmetric matrix. This could lead to a situation where a layout has minimal cost but does not in fact resolve a routing requirement, leading *Sabre* to oscillate indefinitely. This is shown as an example in Figure 4. In this example, nodes marked x_0, x_1 indicate two states for a gate x that must be routed, likewise for the qubits labeled y_0, y_1. All labels on the links are negative log-probabilities for the CNOT gate at that link, and assume that $\Theta >> a > b > c > d$. When one of the x and y qubits both occupy ends of the Θ link, using the 2Q matrix and Equation 2 leads to this links length being set to just Θ. Therefore, when comparing against all possible qubit movements around each x and y

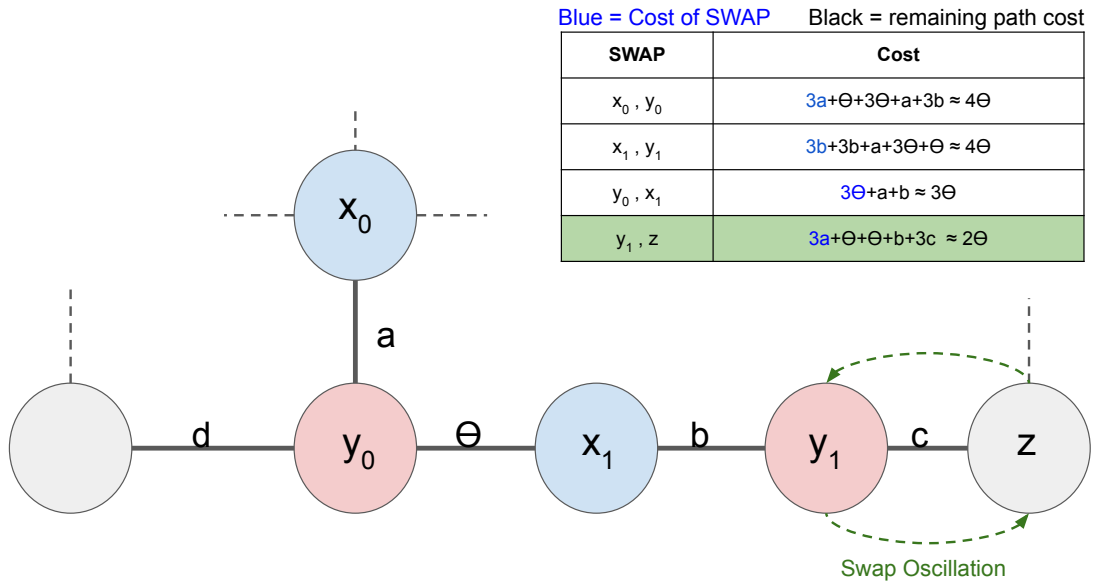

Fig. 4: Swap oscillation caused be an asymmetric R matrix. A possible scenario is that the states of y_1 and z continually swap with each other to ensure that one x and one y qubit occupies the end of the link Θ.

Algorithm 3: Cost function algorithm to adjust for the cost of the SWAP applied.

Input: Front Layer F, Mapping after SWAP π,
Mapping Before SWAP π_i, Reliability Matrix R, Path Matrix P
Output: Cost of SWAP C
`/* NOTE: negative index indicates python-type reverse indexing */`
$C = 0$;
for *gate g in F* **do**
  $C = C + R[\pi(g.q1)][\pi(g.q2)]$;
  for *logical qubit q in g* **do**
    if *q not in SWAP.logical_qubits* **then**
      `/* Search for gate g that SWAP works on */`
      Continue;
    end
    source = q;
    destination = logical qubit in g not q;
    original_cost = $R[\pi_i(source)][\pi_i(destination)]$;
    current_cost = $R[\pi(source)][\pi(destination)]$;
    if *current_cost* > *original_cost* **then**
      `/* Only consider SWAPs that move logical qubits closer */`
      Continue;
    end
    `/* get physical qubit path between qubits in g */`
    path = $P[\pi(source)][\pi(dest)]$;
    next_to_last_qubit = path[-2];
    last_qubit = path[-1];
    unused_swap_cost = $R[next\_to\_last\_qubit][last\_qubit]$;
    adjustment = $R[\pi(SWAP.q1)][\pi(SWAP.q2)]$ - unused_swap_cost;
    $C = C$ + adjustment;
  end
end

as shown in the figure, the most favorable layout is one that keeps one of the x and one of the y qubits at each end of Θ. Therefore, the x and y qubits will never come close enough to interact.

This project addresses this issue by using the path matrix P calculated in Algorithm 2. The algorithm for picking the SWAP that leads to the best move along the shortest negative-log-path is based on the idea that *Sabre* checks *all* possible swaps around each logical qubit. Therefore, give two logical qubits q_0 and q_1 there will be some movement that proceeds q_0 down the shortest negative-log-path to q_1 and vice versa. So, we have the ability to consider both movements on the same path before making a decision. Under this assumption, when it is seen that a SWAP contributes to a logical qubit's movement which makes it closer to its target, the path matrix is used to obtain the SWAP distance between the target node and its neighbor in the path (last 2 nodes in the path). This is then subtracted from the value of $R[q_0][q_1]$ in order to cancel the distance between the target and the neighbor, and the cost of the applied SWAP is then added. This really just flips the score to be that of if we looked at the reliability matrix score after applying the swap

to target. The reason for only doing this when progress is made between two logical qubits is to ensure no subtraction is introduced when the physical distance is increased between qubits, ensuring forward progress is always made and convergence is achieved. Formally, the algorithm for this new cost is given in Algorithm 3.

4.3 Single Qubit Operations

Taking into account single qubit reliability measures requires understanding *when* a certain single qubit measure/gate is performed at a given physical qubit. We can make a few assumptions when it is certain a single qubit operation is executed at a physical location based on how *Sabre* handles "execution" in the beginning of Algorithm 1. Recall, once a 2-qubit gate is finished routing, all following satisfied gates are also finished. Since single qubit gates are automatically satisfied, they are guaranteed to run on the same physical qubit that the preceding CNOT was routed to. So, it is easy to take into account the reliability of all qubit operations in place by checking if they are satisfied. However, once a unsatisfied 2-qubit gate is found, it no longer becomes a guarantee that all succeeding single qubit operations thereafter are executed in place where the original CNOT was routed to. Therefore, when considering single qubit operations, only those up to the next un-satisfied CNOT gates are considered.

Lastly, as shown in Figure 2, measurement error rates can be on the same order of magnitude as CNOT error rates or much higher. So, just adding this error rate to the cost calculation can possibly lead to the case where measurement errors are driving the decisions of a SWAP. However, measurement errors have no information of physical locality embedded in them like the information of path success that is embedded in the edges of the *reliability_graph* in Algorithm 2. Therefore, we may have a situation where the algorithm keeps preferring a low readout error qubit without making progress. To avoid this issue, single qubit operations are taken into account in a second pass that collects all SWAPs that finish the routing of a pair of logical qubits. Thereafter, the score for each of these SWAPs is summed with the negative log probabilities of the success rates for all executable gates on the physical location given by the SWAP. This is described in Algorithm 4. Algorithm 4 assumes a reliability table T that returns the reliability of every gate on the physical machine. This is provided by IBM's backend facilities.

4.4 Decoherence Time

Decoherence time is taken into account in a way that tries to mimic the decay factor of *Sabre*, limiting the number of sequential SWAP gates. This is factored in by keeping a list of depths for each logical qubit of the program (like maximum qubit operations in [3]). To ensure fair scoring for all swaps, a term is added to every swap cost that reflects the estimated latency of the circuit using the qubit with deepest depth. This approximation to depth is used to avoid calculating a true critical path depth on the circuit DAG every time a change is made to the circuit. If there is a SWAP on the "critical qubit" (one with most operations), then an additional latency term describing swap latency is added. Probability of success due to decoherence is estimated to be $e^{\frac{-t}{min(T_1,T_2)}}$, where T_1 and T_2 are the system minimum relaxation and dephasing times to be conservative. The negative log probability

Algorithm 4: Pass on SWAPs after initial scoring to take into account single qubit reliability

Input: Mapping Before SWAP π, List of SWAPs L, Front Layer F
Circuit DAG, Chip Coupling Graph G, Gate Reliability Table T
Output: SWAPs with adjust costs

```
filtered_swaps = ∅
for swap s in L do
    /* copy layout and apply s, put new layout in π_s                    */
    π_s= π.apply(s);
    for gate g in F do
        phys_distance=G.distance(π_s(g.q1),π(g.q2))
        if phys_distance is 1 then
            /* collect swaps result in g's qubits being physical
               neighbors                                                 */
            if s not in filtered_swaps then
                filtered_swaps.append(s);
            end
            /* collect all gates that execute on this swap              */
            filtered_swaps[s].executable_gates.append(g);
        end
    end
end
if filtered_swaps == ∅ then
    /* return swaps w/o changing their scores                           */
    return L;
end
for swap s in filtered_swaps do
    executed_list = ∅;
    π_s= π.apply(s);
    for gate e in s.executable_gates do
        if e in execute_list then
            Continue;
        end
        executed_list.append(e)
        s.cost = s.cost + T[π_s(e.qubits)];
        successors = DAG.get_successors(e);
        for gate n in successors do
            if π_s satisfies n.qubits and n not in executed_list and
            n.predecessors in executed_list then
                s.cost = s.cost + T[π_s(n.qubits)];
                executed_list.append(n);
            end
        end
    end
    return filtered_swaps;
end
```

of this estimated success rate is thus $\frac{t}{min(T_1,T_2)}$. The adjusted cost for a swap is then:

$$Cost(SWAP) = HeuristicCost(SWAP) + \begin{cases} \frac{t_c}{min(T_1,T_2)} \\ \frac{t_c+3\cdot t_{CNOT}}{min(T_1,T_2)} \end{cases} \quad (3)$$

The top term in the bracket is added if the SWAP is applied to the qubits that don't have the largest depth, and otherwise the bottom is added. t_c represents the latency of all gates on the deepest qubits, $3 \cdot t_{CNOT}$ is the estimated time cost of extending the critical path by a SWAP.

4.5 Complexity of Reliability Heuristics

The complexity of *Sabre* with the new reliability heuristics stays largely unchanged, as most of the changes are just to the data structures to score reliability. However, the addition of the second pass to take into account single-qubit operation reliability may add another term to base $O(N^{2.5})$ complexity. This is because all successors are searched to see if they can be executed for a given swap. So if this search is proportional to the depth of the circuit (L), the complexity can be estimated as worst case $O(N^{2.5} \cdot L)$. However it was observed that the number of back-to-back single qubit gates for the benchmarks studied was at most 2, so the complexity realistically would be still be $O(N^{2.5})$.

5 Methodology

5.1 Benchmarks

The benchmarks chosen for this project were picked to give a range in program sizes and logical qubit connectivity. These benchmarks are listed Table 1, and they also represent benchmarks that have been used in studies for reliability aware routing and mapping schemes [11]. The *adder* benchmark is based on the ripple carry adder algorithm as described by Cuccaro in [4], and has short range logical qubit interactions. The QFT programs were chosen to include examples of algorithms that have all distances of qubit communication, and should stress the importance of routing algorithms as it is likely that all communication cannot be determined at mapping time. Lastly, the Bernstein-Vazirani algorithm was chosen in order to represent an algorithm that although has short, medium and long range communication, has more of a central tendency of communication, e.g all CNOT gates target the ancillary qubit of the algorithm. Lastly, the benchmark sizes in terms of qubits (N in Table 1) were chosen based on trial and error with larger versions. It was observed that the large instances mostly resulted in noise, and since it is not logical to compare slight variations in noise across different compiler passes, they have been avoided. Note that the QFT algorithms have an input of state $|0\rangle_n$ and a constant output of state $|0\rangle_n$ with no superposition. This was ensured by adding a Hadamard gate at the end of each quantum line because the QFT is equivalent to the Hadamard gate when the input is $|0\rangle_n$. Ensuring non-superposition allows us to understand when a trial was successful, all other benchmarks have non-superposition outputs by definition.

5.2 Pass Configurations

The following enumerated list gives an overview of the pass-components (*Mapping Passes*, *Routing Passes*) and how the components are combined to create the evaluated passes

Tab. 1: Quantum benchmarks used for transpiler evaluation.

Benchmarks

Benchmark	Input	Output	N	Communication	Pattern
Adder	$\|0001\rangle : \|1111\rangle$	$\|10000\rangle$	9	Short range	Nearest Neighbor
qft_n8	$\|0\rangle_8$	$\|0\rangle_8$	8	All distances	Multi-target Hubs
qft_n6	$\|0\rangle_6$	$\|0\rangle_6$	6	All distances	Multi-target Hubs
bv_n8	String = $\{1\}^7$	$\|1\rangle_7$	8	All distances	Single-Target Hub
bv_n6	String = $\{1\}^5$	$\|1\rangle_5$	6	All distances	Single-Target Hub

(*Evaluated Combinations*), with their descriptions. The names under *Evaluated Combinations* will be referenced when discussing results in Section 6. The naming convention for the evaluated passes is set up to convey how individual components are coupled with each other. The first component in the name is always the mapping pass used. For example *base+noise* uses naive mapping *plus* a base-level noise aware routing. Each different pass is designed to understand a different characteristic of reliability aware routing, and how it impacts success. For example, the pass *base+noise* uses a naive mapping algorithm that just places logical qubits on the first set of corresponding physical qubits, while using *Sabre* configured to have the heuristic cost function of Equation 2. This allows us to see how much of an impact noise aware routing has on its own. For similar reasons there exists a $noise + base$ pass that uses *Qiskit's* noise-adaptive-mapping based on [11], along with *Qiskit's* naive routing that simply moves each pair of logical qubits on multi-qubit gates together without rigorously considering optimality from either an overall distance or reliability perspective. Of course, we would like to observe how more advanced mappings geared towards reliability, like noise adaptive mapping, interact with routing passes that are noise aware as well. So passes like $noise + noise$ are also included in the evaluation.

1. Mapping Passes
   (a) ***base*** - Basic *Qiskit* mapper, for n qubit program, chooses the first n physical qubits.
   (b) ***noise*** - *Qiskit*'s built in noise-adaptive router, *NoiseAdaptiveLayout*. Prioritizes the heaviest edges in the program qubit interaction graph, and maps each logical qubit on the high priority edges to the physical qubits with the highest readout and CNOT/SWAP reliability. Based on GreedyE from Murali [7].
   (c) ***sabre*** - mapping phase of *Sabre* [6]. Leverages the router pass to pick out the best initial mapping.
2. Routing Passes
   (a) ***base*** - *Qiskit* base routing. Picks the shortest path between physical qubits when routing is needed and simply takes it without considering any scores.
   (b) ***sabre*** - *Sabre* routing as shown in Algorithm 1.
   (c) ***noise*** - most basic noise cost heuriestic based on the reliability matrix only, e.g. Equation 2. Effectively just swap D with R.
   (d) ***exec*** - Additional single qubit gate second pass (Algorithm 4).

(e) ***noise_la*** - Reliability based lookahead swapping that leverages *Sabre's* already existing lookahead swap as shown in Equation 1.

(f) ***adjust*** - Reliability heuristic that takes into account path asymmetry, based on Algorithm 3.

(g) ***adj+dec*** - combined adjustment with decoherence awareness from Equation 3.

3. Evaluated Passes - x.y indicates pass x option y

(a) ***base+base*** - 1.a + 2.a

(b) ***base+noise*** - 1.a + 2.c

(c) ***base+sabre*** - 1.a + 2.b

(d) ***noise+adj+dec*** - 1.b + 2.g

(e) ***noise+adj+dec+exec*** - 1.b + 2.g + 2.d

(f) ***noise+adj+exec*** - 1.b + 2.f + 2.d

(g) ***noise+adjust*** - 1.b + 2.f

(h) ***noise+base*** - 1.b + 2.a

(i) ***noise+noise*** - 1.b + 2.c

(j) ***noise+noise+exec*** - 1.b + 2.c + 2.d

(k) ***noise+noise_la*** - 1.b + 2.e

(l) ***noise+sabre*** - 1.b + 2.b

(m) ***sabre+noise*** - 1.c + 2.c

(n) ***sabre+sabre*** - 1.c + 2.b

5.3 Quantum Machines and Setup

Each benchmark is compiled with all the passes from the previous section, and each program is run on three different machines. The three machines used were *ibmq_16_melbourne*, *ibmq_toronoto*, and *ibmq_manhattan*. Manhattan has 65 qubits, Toronto has 27 qubits, and Melbourne has 16 qubits. The main reason for choosing these three machines is that it allows us to look at error variability that may exist across machines, and connectivity variability across machines. Melbourne has a higher average qubit connectivity compared to the other two, with 3 edges per qubit, while Toronto and Manhattan have mostly only 2 edges per qubit besides qubits at intesections that have 3 edges.

For all benchmark runs and passes, they were all run within the same error rate calibration window (usually a day for IBM). This allows us to factor out time variation in compilation outputs due to different reported error rates that tend to vary from day to day. Each benchmark is run for 8k shots, and all benchmarks are compiled using *Qiskit's* compiler/transpiler infrastructure. All passes are managed using *Qiskit's* pass manager, and all other supporting passes like scheduling are held constant for each combination of mapping and routing. The *Sabre* routing algorithm is considered with all its features active, such as decay rate controls and considering the following layer of gates to be routed as shown in Equation 1.

6 Results and Discussion

When comparing the reliability of the different passes, the probability of successful trial (PST) is used as the main metric. This score provides the probability that any given trial will return the desired result. Figure 5 shows the relative PST for all passes and benchmarks for the Toronoto machine. All PST results are normalized to *noise+sabre* since this is the most advanced configuration for mapping and routing that *Qiskit* provides from a reliability standpoint. Another important metric also considered is the rank of correct answer, as with both PST and rank, we can get an understanding of the overall impact of the different passes. This is shown in Figure 7, where each bar represents the inverse of the rank of the correct answer for each program on the Toronto machine. So, any bar at the dashed red line indicates that the given pass provides the correct answer as the strongest answer.

The first thing to notice from Figure 5 is that all pass combinations with the highest PST take into account reliability information in some form. This indicates that it is worth while to take into account the reliability data provided by IBM instead of ignoring it. Furthermore, all of the best PST combinations include a reliability routing heuristic not available in *Qiskit's* base implementation (except *bv_n6*). However, there is no clear pass that performs the best for every single benchmark.

Table 2 and Figure 6 give perspective on the characteristics of the circuits generated for each pass on Toronto. Table 2 gives the raw gate and depth counts, and Figure 6 normalizes all the depths to *sabre+noise*. It is interesting to consider the relationship between the depths of the resulting circuits and their success rates. The passes that have the lowest normalized depths tend to also have the highest PSTs. For example, *sabre+sabre* has a normalized depth of 0.88× for *bv_n8*, and a normalized PST of 2.93× making it the pass with the 2nd highest success rate. Furthermore, for this program, there are only 4 passes that have normalized depths $> 1\times$ that also have normalized PST's of $> 1\times$. Following the same trend, all passes with $> 1\times$ depth have a $< 1\times$ PST for the *bv_n6* program. For programs with larger depths, like *adder* and *qft_n8*, as shown by Table 2, it seems to be that the depth is not as good as an indicator for how well a mapping/routing combination will perform in terms of PST. For example, for *qft_n8*, the four best passes that surpass the PST of *noise+sabre* have up to 1.37× the depth of *noise+sabre*. Also, although having a

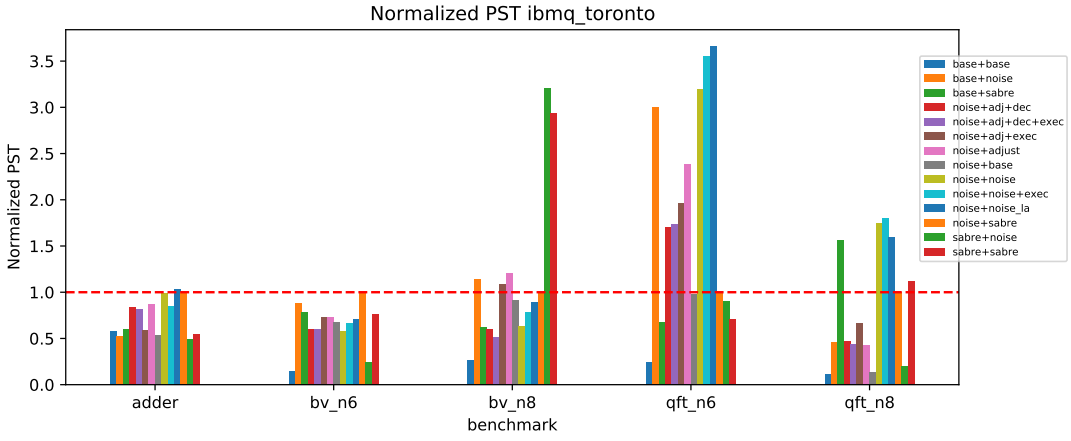

Fig. 5: PST normalized to *noise+sabre* for the *ibmq_toronto* machine. Each color in the key from top to bottom goes left to right for each benchmark.

Tab. 2: Raw gate and depth counts after each path for Toronto. G is the number of gates, D is the depth of the circuit.

passes	Adder		bv-n6		bv-n8		qft-n6		qft-n8	
	G	D	G	D	G	D	G	D	G	D
base+base	376	255	62	41	68	45	132	106	279	224
base+noise	343	249	35	24	47	27	120	78	270	144
base+sabre	256	195	35	24	47	30	117	83	237	139
noise+adj+dec	295	168	29	20	44	31	141	79	282	156
noise+adj+dec+exec	298	171	29	20	44	31	147	87	288	162
noise+adj+exec	343	222	23	18	38	27	135	79	261	126
noise+adjust	313	200	23	18	38	27	135	79	267	139
noise+base	433	220	29	20	56	32	147	88	321	186
noise+noise	292	203	29	20	56	32	129	70	243	119
noise+noise+la	283	196	23	18	56	32	126	65	243	115
noise+noise+exec	307	228	29	20	56	32	129	73	243	122
noise+sabre	253	168	23	18	35	26	129	79	225	101
sabre+noise	382	291	53	48	56	32	174	124	336	219
sabre+sabre	274	203	26	20	38	23	111	74	207	104

smaller PST than *noise+sabre* (0.98×), *noise+noise* is the only pass that can result with the *adder*'s correct output being the most prominent while also having 1.2× the depth of *noise+sabre*. In summary, this data indicates that circuits with larger logical depths are able to trade off more depth for more reliability.

The relationship between circuit depth and PST performance seemed to be stronger for the Manhattan machine, where even for the deep circuits like *qft_n8*, the highest PST was attained by a pass with $< 1\times$ circuit depth. To get a better look into this, Table 3 summarizes the average rank of a given pass to see if there is any preference across machines for certain passes. This shows the average rank given to each pass across all benchmarks for

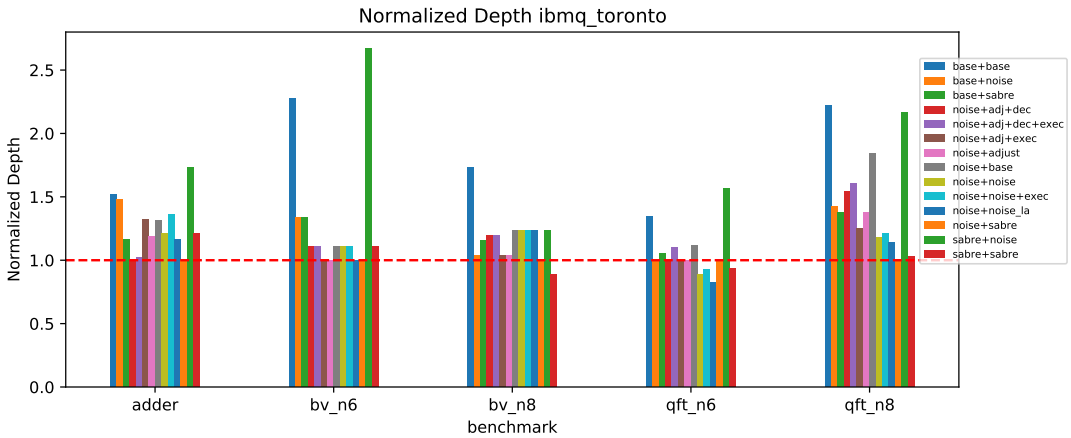

Fig. 6: Circuit depths normalized to *noise+sabre* for the *ibmq_toronto* machine. Each color in the key from top to bottom goes left to right for each benchmark.

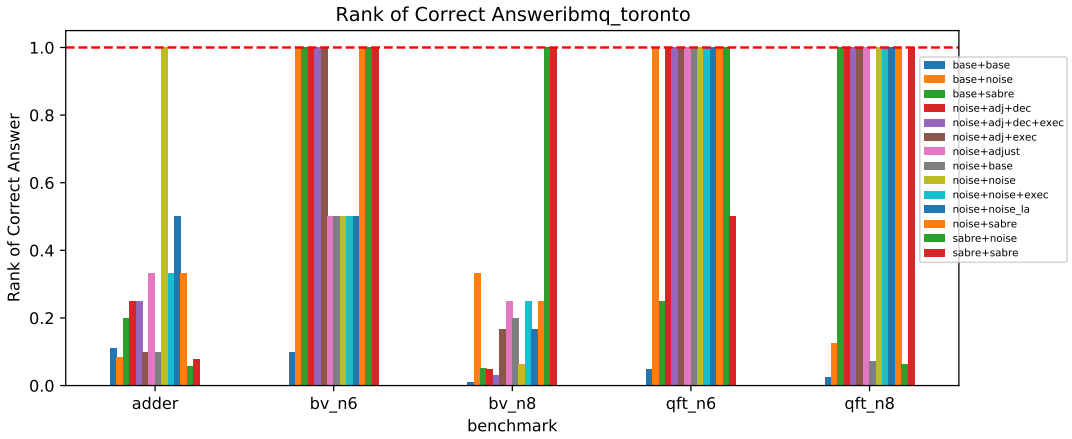

Fig. 7: Inverse of rank for each benchmark and pass on the *ibmq_toronto* machine. Bars at the red line indicate that the correct answer is the output with highest PST.

each machine, and overall (average across all machines). The rank is a weighted calculation that assigns a weight of 14 to the first ranked pass, 13 to the second, and so on. The interesting point is that Melbourne and Manhattan both prefer *noise+adj+dec*. While Toronto prefers *noise+noise_la*, and *noise+adj+dec* is Toronto's 10th ranked pass. Looking into this, it was found that Melbourne and Manhattan had average relaxation times (T_1) of 56 μs and 58 μs respectively, while Toronto's average T_1 was 94μs. Due to the lower T_1 times, it would be expected that the best passes be those that keep the serialized gates from growing too large, like *noise+adj+dec*. It is also worth noting that in the general case the highest ranking passes are *noise+adj+dec* and *noise+noise+exec*, indicating that all types of reliability information should be accounted for in a reliability pass, not just SWAP reliability.

Table 4 gives average ranks for each pass for each benchmark type, averaged across the three machines and the different instances of *qft* and *bv* to see if there is a program-specific preference of mapping/routing passes. The passes are organized with respect to the largest average ranks for *qft*. An interesting difference between *bv* and *qft* is the change in ranking for the *noise+base* pass. For *bv* it is the second highest ranked pass, while for *qft* it is the second *worst* pass. This is likely due to *qft*'s more complex routing requirements than *bv*. *bv* only has one central qubit used as the target. This type of connectivity interacts with the mapper by placing the central target logical qubit on one of the most reliable physical qubits. Afterwards, the remaining control qubits are filled in around the most reliable paths. Afterwards, the basic router just chooses the lowest distance path between target and control, but always with the control as the qubit being moved. Thus, the initial mapping allows the basic router to avoid low reliability connections. This is similar to assumptions made by Murali *et al.* in [7]. However, with more advanced qubit communication like *qft*, which has qubits that act as both target and source in certain phases, this assumption does not hold as the initial mapping gets perturbed by all the routing needed. Furthermore, *qft* programs have much more single qubit gates, with *qft_8* having 100 single qubit operations, and *bv_n8* having 16 single qubit ops. So, this combination would logically lead to the preference to a pass that takes into account both routing and single qubit operations. Lastly, *adder* ranks *noise+noise_la* the highest on average. This makes sense since this pass looks ahead to the next operations that need to be routed, and *adder* has very localized

Tab. 3: Table of average ranks based on PST for each pass and machine. Entries are sorted by highest total average rank. This is a larger is better score.

Average Pass PST Rank (Machines)					
passes	Man.	Mel.	Avg Mel-Man	Tor.	Total Avg
noise+adj+dec	10.2	10	10.10	6.2	9.13
noise+noise+exec	10.4	7.4	8.90	9.8	9.13
noise+noise_la	8	8.4	8.20	11	8.9
sabre+sabre	10.8	7.4	9.10	8.2	8.88
noise+sabre	7.4	8	7.70	10.2	8.33
base+noise	5	10.6	7.80	8.6	8
noise+adjust	10.6	4.2	7.40	9.2	7.85
base+sabre	7	8.6	7.80	7.2	7.65
noise+adj+dec+exec	7	9.2	8.10	5.4	7.43
noise+adj+exec	8.2	5.2	6.70	8.6	7.18
sabre+noise	5.6	9.2	7.40	4.8	6.75
noise+noise	5.8	5.8	5.80	9	6.6
noise+base	6.8	6	6.40	5	6.05
base+base	2.2	5	3.60	1.8	3.15

Tab. 4: Table of average ranks based on PST for each pass and benchmark. Entries are sorted by highest qft average rank. This is a larger is better score.

Average Pass PST Rank (Benchmarks)			
passes	adder	bv	qft
noise+noise+exec	6.67	7.50	12.17
noise+noise	7.33	4.17	9.33
sabre+sabre	6.00	9.83	9.17
base+noise	6.00	8.50	8.67
noise+noise_la	10.33	9.17	8.50
noise+adj+dec	9.33	9.00	8.33
noise+adj+exec	8.00	6.00	8.33
base+sabre	7.33	7.33	8.00
noise+sabre	9.33	8.83	7.83
noise+adj+dec+exec	6.33	7.17	7.67
noise+adjust	8.33	8.67	7.17
sabre+noise	9.00	6.50	5.33
noise+base	4.33	9.50	3.17
base+base	6.67	2.83	1.33

qubit interactions such that only groups of a few qubits in a small number of layers need to be shuffled to fulfill routing requirements. This allows looking ahead to be able to accurately assess which shuffling leads to the most reliable routing.

7 Related Work

Zulehner *et al* [12] describes an algorithm that targets IBM QX architectures, and the main aim of the algorithm is to reduce the total number of gates present in the resultant circuit after mapping. Their approach is based on the A* algorithm, which is just a tree-based state space exploration method that makes informed choices based a heuristic scoring mechanisms to guide which SWAP gates to consider. The A* algorithm is used as a mechanism to cope with the exponential space of possible SWAPs by using a heuristic that scores a given set of SWAPs based on how many gates those SWAPs will inject into the circuit, and how many gates will be needed after this set of SWAPs to ultimately complete the mapping for a CNOT gate. As the algorithm evolves, only SWAPs with the lowest scores are pursued until a mapping is reached which admits the ability for 2 qubits to perform their CNOT gates. Their heuristic is admissible, that is, the best possible choice of SWAPs is never filtered out and will eventually be converged upon.

Li *et al* [6] aim at the same problem that Zulehner tries to address and also targets IBM quantum chips, but argues that Zulehner's approach is scales $O(exp(N))$ with the number of qubits N since it is still an exhaustive search given that Zulehner's heuristic is admissible. Instead of doing exhaustive searches, Li *et al* present an algorithm that has complexity $O(N^{2.5})$ and is based on a greedy approach of evaluating all possible 1 hops around each qubit that needs to be routed, and selects the 1 hop that results in the lowest sum of distances between all logical qubits that must be mapped at a given layer in the dependency graph. Building on this they add decay factors to their heuristic in order to avoid serializing SWAPs to increase parallelism and shorten run time, and they also factor in the distances among logical qubits in the next layer of the dependency graph such that they can find a SWAP mapping that minimizes the amount of additional work needed for the next layer. However they do not consider the reliability of a mapping, nor do they add in any information about single qubit gates to the mapping process.

Zulehner *et al.* and Li *et al.* aimed at lowering the overall gate count and depth of the resulting circuit, but other approaches have been developed in order to directly target reliability as a key optimization metric. Murali *et al.* use SMT solvers in [8, 7] in order to find an optimal mapping that maximizes an objective function that reflects maximizing the success rate of a program based on reliability data for both CNOT and readout error rates. The CNOT error rates are used to score not only the reliability of a given CNOT operation between physical qubits, but also to score the SWAP paths that would need to be taken in order to move 2 quantum states to be adjacent to one another. However, they are only concerned with optimizing the initial allocation process, and are under the assumption that a good mapping will also lead to a good routing design. Although they do use SMT solvers to design a qubit mapping, they also develop two different greedy approaches that use profiling from the connectivity of logical qubits in the program to determine the placement of logical qubits on hardware [7]. One of the greedy approaches orders the priority of logical qubits to be mapped based on their degree of connectivity in the program (GreedyV), while the other approach orders the priorities of edges based on the weight of the edges (GreedyE). The weight of an edge in the program graph is the number of times a CNOT is executed between two logical qubits in the program. The latter greedy approach (GreedyE) is actually implemented within *Qiskit's* mapping pass, as *NoiseAdaptiveLayout* [1].

Tannu *et al* in [11] also tries to address the issue of reliability aware mapping and routing, and base their approach also on the state-space exploration algorithm $A*$. This approach optimizes reliability by choosing SWAP operations between qubits that have the lowest

CNOT error rates. However, it seems that this work does not consider the asymmetric nature of program success based on the path of highest reliability. For example, Murali in [8] does this by using their 2Q reliability matrix which has a value for each pair of physical qubits, and calculates the reliability between the pair of qubits by taking into account that one would like to use the lowest error rate links for SWAPs due to a SWAP having 3 CNOT operations while reserving the higher error rate links for individual 1 time CNOT gates. Furthermore, a state space exploration based on $A*$ has been shown by Li in [6] due to its large memory demands. Tannu also provides a heuristic that attempts a reliability aware mapping based on program profiling that helps place high interaction qubit pairs on high reliability links, which is very similar to Murali's [7] GreedyE approach previously discussed.

8 Conclusion

This project evaluated the impact of program success for 14 individual compiler passes across 5 programs and 3 different quantum machines, along with developing reliability heuristics that can be ensured to converge for greedy routing algorithms. It was found that a pass that combines the reliability of a SWAP path, single qubit gates, and measurements leads to the best general solution for reliability compared relatively to the other 13 passes. Furthermore, it was shown that the choice of a pass can be machine and program dependent. For example, machines with lower relaxation times preferring passes that include terms that account for decoherence errors. Lastly, program qubit communication patterns seem to play a role in determining the most favorable pass. With simple localized communication patterns preferring passes that consider the reliability of the next layer of CNOTs for a given current routing layer.

References

[1] NoiseAdaptiveLayout — qiskit 0.23.0 documentation, . URL `https://qiskit.org/documentation/stubs/qiskit.transpiler.passes.NoiseAdaptiveLayout.html`.

[2] Transpiler passes (qiskit.transpiler.passes) — qiskit 0.23.0 documentation, . URL `https://qiskit.org/documentation/apidoc/transpiler_passes.html`.

[3] M. Alam, A. Ash-Saki, and S. Ghosh. Circuit compilation methodologies for quantum approximate optimization algorithm. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 215–228. IEEE. ISBN 978-1-72817-383-2. doi: 10.1109/MICRO50266.2020.00029. URL `https://ieeexplore.ieee.org/document/9251960/`.

[4] S. A. Cuccaro, T. G. Draper, S. A. Kutin, and D. P. Moulton. A new quantum ripple-carry addition circuit. URL `http://arxiv.org/abs/quant-ph/0410184`.

[5] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland. Surface codes: Towards practical large-scale quantum computation. 86(3):032324. ISSN 1050-2947, 1094-1622. doi: 10.1103/PhysRevA.86.032324. URL `http://arxiv.org/abs/1208.0928`.

[6] G. Li, Y. Ding, and Y. Xie. Tackling the qubit mapping problem for NISQ-era quantum devices. In *Proceedings of the Twenty-Fourth International Conference on Architectural*

Support for Programming Languages and Operating Systems, ASPLOS '19, pages 1001–1014. Association for Computing Machinery. ISBN 978-1-4503-6240-5. doi: 10.1145/3297858.3304023. URL http://doi.org/10.1145/3297858.3304023.

[7] P. Murali, J. M. Baker, A. Javadi-Abhari, F. T. Chong, and M. Martonosi. Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 1015–1029. Association for Computing Machinery, . ISBN 978-1-4503-6240-5. doi: 10.1145/3297858.3304075. URL http://doi.org/10.1145/3297858.3304075.

[8] P. Murali, N. M. Linke, M. Martonosi, A. J. Abhari, N. H. Nguyen, and C. H. Alderete. Full-stack, real-system quantum computer studies: architectural comparisons and design insights. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 527–540. ACM, . ISBN 978-1-4503-6669-4. doi: 10.1145/3307650.3322273. URL https://dl.acm.org/doi/10.1145/3307650.3322273.

[9] M. Y. Siraichi, V. F. d. Santos, S. Collange, and F. M. Q. Pereira. Qubit allocation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 113–125. Association for Computing Machinery. ISBN 978-1-4503-5617-6. doi: 10.1145/3168822. URL https://doi.org/10.1145/3168822.

[10] S. S. Tannu and M. K. Qureshi. Mitigating measurement errors in quantum computers by exploiting state-dependent bias. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pages 279–290. Association for Computing Machinery, . ISBN 978-1-4503-6938-1. doi: 10.1145/3352460.3358265. URL http://doi.org/10.1145/3352460.3358265.

[11] S. S. Tannu and M. K. Qureshi. Not all qubits are created equal: A case for variability-aware policies for NISQ-era quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 987–999. Association for Computing Machinery, . ISBN 978-1-4503-6240-5. doi: 10.1145/3297858.3304007. URL http://doi.org/10.1145/3297858.3304007.

[12] A. Zulehner, A. Paler, and R. Wille. An efficient methodology for mapping quantum circuits to the IBM QX architectures. 38(7):1226–1236. ISSN 1937-4151. doi: 10.1109/TCAD.2018.2846658.