11-3-2022

Compilation Optimizations to Enhance Resilience of Big Data Programs and Quantum Processors

Travis D. LeCompte
Louisiana State University and Agricultural and Mechanical College

COMPILATION OPTIMIZATIONS TO ENHANCE RESILIENCE OF BIG DATA PROGRAMS AND QUANTUM PROCESSORS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Division of Electrical and Computer Engineering

by
Travis LeCompte
B.S., Louisiana State University, Baton Rouge, LA 2017
December 2022

Acknowledgments

Completing this dissertation would not have been possible without the help of many friends and associates that I have made along the way. First, I would like to thank my advisor, Dr. Lu Peng, for the guidance, assistance, and advice he has provided with all of my research and development through these years. I also thank each of my committee members: Dr. Anas (Nash) Mahmoud, for his guidance on my professional development and his advice on snacks; Dr. Jianhua Chen, for acting as my honors college advisor in my undergraduate years and continuing to provide feedback on the machine learning portions of my research; Dr. Ramachandran (Vaidy) Vaidyanathan for his questions and feedback regarding all my work, and his willingness to help in times of need; and Dr. Pramod Achar, from the Department of Mathematics, for his time while acting on my committee and his feedback from an alternate point of view. I would also like to thank Ms. Adrienne Steele with LSU's Society of Peer Mentors for allowing me to participate as a member and be involved in their community outreach activities. I would like to thank all of the ECE faculty, namely Ms. Beth Cochran, Ms. Stacey Vessel, Ms. Stacie Oliver, and Ms. Stacia Moses, for all of their assistance over the years. Lastly, I would like to thank Dr. Tzeng from ULL for his consistent editing assistance.

I am thankful both to the Division of Electrical and Computer Engineering and the State of Louisiana for funding my research throughout this journey.

Additionally, I would like to thank my family for supporting me through the process. First and foremost, I must thank Jesslyn for her support and patience. I would also like to thank Adam, Christian, Dakota, Collin, and Edwardo for sticking with me through many late nights of work. Finally, I would like to thank my fellow Ph.D. students, Tao Lu and Fang Qi, for many discussions of our work.

Table of Contents

List of Figures

Abstract

Modern computers can experience a variety of transient errors due to the surrounding environment, known as soft faults. Although the frequency of these faults is low enough to not be noticeable on personal computers, they become a considerable concern during large-scale distributed computations or systems in more vulnerable environments like satellites. These faults occur as a bit flip of some value in a register, operation, or memory during execution. They surface as either program crashes, hangs, or silent data corruption (SDC), each of which can waste time, money, and resources. Hardware methods, such as shielding or error correcting memory (ECM), exist, though they can be difficult to implement, expensive, and may be limited to only protecting against errors in specific locations. Researchers have been exploring software detection and correction methods as an alternative, commonly trading either overhead in execution time or memory usage to protect against faults.

Quantum computers, a relatively recent advancement in computing technology, experience similar errors on a much more severe scale. The errors are more frequent, costly, and difficult to detect and correct. Error correction algorithms like Shor's code promise to completely remove errors, but they cannot be implemented on current noisy intermediate-scale quantum (NISQ) systems due to the low number of available qubits. Until the physical systems become large enough to support error correction, researchers instead have been studying other methods to reduce and compensate for errors.

In this work, we present two methods for improving the resilience of classical processes, both single- and multi-threaded. We then introduce quantum computing and compare the nature of errors and correction methods to previous classical methods. We further discuss two designs for improving compilation of quantum circuits. One method, focused on quantum neural

networks (QNNs), takes advantage of partial compilation to avoid recompiling the entire circuit each time. The other method is a new approach to compiling quantum circuits using graph neural networks (GNNs) to improve the resilience of quantum circuits and increase fidelity. By using GNNs with reinforcement learning, we can train a compiler to provide improved qubit allocation that improves the success rate of quantum circuits.

1. Introduction

With the ubiquity of the Internet and increasing accessibility to technology, more and more individuals are interacting online for every reason imaginable. With each online action comes data that is generated, stored, and analyzed in some way. This overwhelming and ever-increasing volume of data, commonly referred to as Big Data, poses serious difficulties in storage, transmission, and analysis. A frequently overlooked aspect is that of resilience to errors that occur during execution. As the volume calculations increases, so too does the frequency of errors. A recent study of the Tesla K20 GPU demonstrated that the mean time between failure (MTBF) of double bit errors can be as low as 160 hours, or roughly one error per week [93]. These errors can cause a process to fail, or worse, provide incorrect results with no notification of an error. This problem can be exacerbated by computations in extreme environments like space or high temperatures. Additionally, the desire for lower power consumption in devices, either for increased battery life or reduced heat generation, drives voltage levels lower, thus increasing sensitivity to environmental interference. Some solutions like error correcting memory (ECM) can help protect a system's memory, but this cannot protect all components like registers or gates and may not be feasible depending on the application. Many software resilience methods have been studied as an alternative, commonly using some form of redundancy and checkpointing to detect and correct errors.

Quantum computers, a promising new computing technology, experience a similar error problem as classical systems, though much more severe. Built using various physical technologies like photons instead of electronics, quantum computers offer the potential to solve certain problems that are intractable on classical systems, like factorization, in polynomial time. The currently available systems are classified as noisy intermediate-scale quantum (NISQ)

devices. These NISQ devices experience a much higher error rate than classical systems, and the severity of individual errors is typically more impactful on the output. Error correction algorithms exist to completely remove most errors, but these algorithms cannot be implemented on current NISQ systems due to their limited size. Instead, developers have chosen to deal with the errors and minimize their impact through various means. The most common approach is simply to run a circuit many times to help remove the effects of random errors. Other approaches, however involve adjusting the compilation of a circuit to make it more resilient to errors, similar in concept to classical methods.

In this work, we discuss two methods to increase the resilience of classical Big Data algorithms. Specifically, we introduce the following two approaches:

- Improving the reliability of single-threaded Big Data kernels using algorithm invariants.
- Protecting synchronization mechanisms of multi-threaded Big Data kernels algorithms using fine-grained logging mechanisms.

Additionally, we will provide a short quantum computing overview and introduce three quantum computing improvements. Specifically, we will discuss the following methods:

- Improving quantum execution reliability using prioritized cache qubits.
- Utilizing graph neural networks (GNNs) to aid in compiling quantum circuits to achieve greater reliability.
- Accelerating compilation of quantum neural networks (QNNs) using partial compilation.

The remainder of this work will be structured as follows. First, we will discuss both classical approaches for Big Data algorithms in Chapters 2 and 3. Additional background regarding quantum computing is given in Chapter 4. The addition of prioritized cache qubits is discussed in Chapter 5. Partial compilation improvements for QNNs are discussed in Chapter 6, while GNN

aided compilation of quantum circuits is presented in Chapter 7. Lastly, we conclude this work in Chapter 8.

2. Soft Error Resilience of Big Data Kernels through Algorithmic Approaches

2.1. Introduction

In today's world, Big Data processing has become progressively more prevalent. A large percentage of the world's population spends hours every day connected to the Internet in some way. This continuous usage by such a large population generates an immense volume of data to process. Recent measurements by Cisco estimate that total annual network traffic has reached the zettabyte threshold as of 2016 and continues to grow [8]. To handle this processing, most companies utilize high-performance computers such as supercomputers or computing clusters. As these high-performance computers become more and more advanced, one primary focus is to minimize the amount of power required by a processor to perform operations. Traditionally, error resilience is focused on natural events that can affect the contents within computer clusters [6], such as charged alpha particles or cosmic rays. With new circuitry technologies and low-power operations for energy savings, errors can arise form more varied usage conditions (such as high temperature/altitude zones/vehicles), 3D interconnect and chip structures, etc. [56].

A recent study reveals the mean time between failure (MTBF) of double bit errors in the Tesla K20 GPUs used in the Titan supercomputer [93] is as low as 160 hours (about one error per week.) This is 3 magnitudes smaller than the manufacturer-rated MTBF of 219,282 h under a controlled environment [73]. As such, Big Data practitioners who seek to build clusters using commodity hardware (which may not have ECC like the K20 does) may not be able to consider

soft faults to be an impossibility. The result is errors on less protected systems may go unnoticed. Thus, it has become increasingly important for programs to detect and prevent these faults on their own

There exists a large range of Big Data algorithms for many specific applications, ranging over regression and classification to simple statistical reporting. One cannot hope to examine each program individually to study its fault-tolerant potential. However, it is common for programs to share features and reuse basic algorithms. Researchers have come up with proposals with the goal of characterizing Big Data programs with simpler benchmarks, including HiBench [47], BigBench [39], AMP Benchmarks [2], YCSB [22], LinkBench [3], CloudSuite [32], and BigDataBench [38]. The latest one of the collections, BigDataBench, identifies eight Big Data kernels or dwarves that are used by a significant number of these programs: linear algebra, sampling, transform operations, graph operations, logic operations, set operations, sort, and statistic operations. Thus, we believe that studying the fault tolerance of one representative from each of these kernels will provide insight into potential fault-tolerant mechanisms for Big Data overall. In this paper, we selected the following eight algorithms to represent each Big Data kernel, respectively: matrix multiplication, Markov chain Monte Carlo, fast Fourier transform, breadth-first search, MD5, union set operation, quicksort, and GREP. We observe the fault-tolerant potential of each algorithm by identifying algorithm-specific invariants that, when violated, indicate the occurrence of a soft fault. These invariant checkers are implemented into each algorithm along with a recovery system. Faults are then injected during the execution of the algorithm with fault injection tools such as KULFI, and the resulting behavior is observed. We show that these fault-tolerant systems reduce the impact of these faults by lowering both incorrect answers and execution failures. This provides information into the effectiveness of this

method of fault tolerance on Big Data applications in general, and the value of fault resilience in Big Data algorithms. Our experiments demonstrate that the soft error resilience will be significantly improved with the proposed methods.

2.2. Background

2.2.1. Soft Errors

Soft errors are transient bit flip errors that can occur during program execution. Soft errors can be caused by decay of electronic components or environmental conditions like radiation or temperature changes. They can occur at any point during program execution and in any physical location on the host machine, including RAM, registers, cache, or even ALUs. A key feature of soft errors is their transient nature. Unlike hard errors that are caused by permanently faulty hardware, soft errors occur randomly due to environmental conditions and cannot be predicted or fixed by replacing hardware. The effects of soft errors can be broadly classified into three types: crashes, where the program abnormally quits execution; hangs, where the program enters an infinite loop; and incorrect output, where the program completes execution but provides a wrong result as a result of silent data corruption (SDC).

Soft errors are typically addressed using a two-stage solution. First, a solution must detect the presence of a fault. This is easy for crashes as a program exiting early is rather obvious. Hang detection is somewhat more difficult but can be achieved by tracking execution progress using separate observation threads or processes. In most cases, SDC is the most difficult of the three outcomes to identify and correct, as a user may not know the expected correct output to compare their results against. Instead, SDC is commonly identified using redundancy – execute the same program two or three times in parallel or sequence and identify if the results match.

Once a fault is identified, it then needs to be addressed and corrected. Crashes and hangs are both commonly addressed using some form of checkpointing that periodically saves execution progress and can roll back to a correct state in the event of failure. SDC on the other hand requires re-execution when comparing two separate executions. When using triple redundancy instead, one can use a "majority rules" policy to select the result that occurs most frequently.

2.2.2. Invariants

Invariants are characteristics of an algorithm, or sections of an algorithm, that must hold true if the algorithm is executing correctly. In other words, invariants are "rules" that must be kept to ensure correctness of the algorithm. Some invariants are low-level and programmatic. For example, incrementing a variable should only ever increase the value by one, or the *else* branch of an *if-else* statement should only be taken if the condition is false. Other invariants are more high-level and conceptual. A good example here are hashing algorithms – if you hash a value twice with the same input and secret keys, then you should get the same hash as output. Additionally, it is much easier to run the hashing algorithm twice than to reverse the hashing process and recreate the input value from the hash.

These invariants provide a simpler way to monitor the correctness of an algorithm during execution. If the invariants of an algorithm are violated at any point during execution, it is likely that a fault has occurred any may cause SDC. With this in mind, we can design a detection method and associated correction methods to increase program resilience to faults by taking advantage of algorithmic invariants.

2.3. Design

For each of the Big Data kernels included, we follow a general approach to analyze the fault tolerance of the kernel. This process involves identifying a specific implementation of an algorithm to test and represent the kernel, which must in turn be compatible with KULFI and LLVM; identifying one or more invariants within the algorithm; implementing the said invariant(s) to check for errors, along with recovery in the event any invariant is violated; identifying an error criteria, to allow for detecting improper program output; and lastly, injecting faults into program execution during tests to observe the effects of the invariant implementation and recovery system.

To identify implementations for testing, we searched for published implementations of algorithms that we consider exemplified the kernel in question. This search typically began with the BigDataBench benchmark suite itself, though some compatible implementations were difficult to find and are taken from public GitHub repositories.

Next, we attempt to identify invariants within the algorithm for use in identifying errors during program execution. Some implementations are relatively simple, such as grep, and do not exhibit high-level invariants. For these implementations, we choose to use redundancy in critical operations to eliminate errors. We refer to these as programmatic invariants. For those that do contain invariants, we then implement the check for the invariant along with a recovery system. Thus, if faults are injected into the program, the invariants potentially fail and the program recovers from the fault, instead of allowing the fault to propagate to output error or program failure.

However, we must be able to detect whether a program is creating proper output or not. These criteria are algorithm specific, typically involving a comparison of outputs or of statistics

for the outputs. For some programs such as union, this is very straightforward, while others such as breadth-first search or Markov chain Monte Carlo (MCMC), which relies on random sampling, are more complex. In general, however, we collect this information by executing the algorithm with no modifications or faults injected to collect the golden, error-free output. During experiments, all results are compared to these golden outputs to identify success or failure.

Lastly, we need to test the effectiveness of the fault-tolerant additions. This testing includes a minimum of 5000 trials for both fault-tolerant and non-fault-tolerant algorithms each, along with varying-sized data sets for some algorithms. Faults are injected dynamically into program execution using KULFI, and outputs for both the fault-tolerant and non-fault-tolerant versions are compared with the "correct" program output, as determined by a non-fault-tolerant execution with no injected faults. This comparison gives a metric to determine whether a trial is incorrect, and how incorrect it is.

First, we will provide an overview of the types of invariants used for each kernel. The invariants are shown in the following table, while implementation details can be found in the associated paper [64].

Table 2.1. Table of kernels, selected benchmark, and implemented invariant.

Kernel	**Benchmark**	**Invariant**
Linear Algebra	Matrix Multiplication	Algorithmic (Matrix-Vector Multiplication)
Sampling	MCMC	Programmatic
Transform Ops.	FFT	Algorithmic (Parseval's Thm)
Graph Ops.	BFS	Programmatic
Logic Ops.	MD5	Programmatic
Set Ops.	Set Union	Programmatic
Sorting	Quicksort	Programmatic
Statistics Ops.	GREP	Programmatic

As shown, the invariants can be classified into two classes: strict algorithmic invariants and program invariants. Algorithmic invariants are highly specific and may not exist for every

benchmark. They take advantage of features of an algorithm that make the results easy to verify with simple methods. By comparison, program invariants rely on redundancy mechanisms within the code to protect important regions. Some examples include loops and control structures, counters for incremental operations, or accumulating variables.

2.4. Results

Here we will present a summary of results from testing each of the eight Big Data kernels. First, we show the execution breakdown for the baseline algorithms and their fault tolerant versions. These results are shown in Figure 2.1. As shown, most of the benchmarks experience a large increase in correct output when adding the fault tolerant methods. We observe that we can group the algorithms together by their results. Specifically, we identify three classes of algorithms.

The first class of algorithms, Type 1 algorithms, exhibit algorithmic invariants that allow for less complex error detection mechanisms. Type 1 algorithms include both matrix multiplication and FFT. These algorithms are the most vulnerable to errors without fault tolerance methods, likely due to their heavy reliance on raw numerical data and the potential for error propagation. However, these high-level algorithmic invariants show the greatest improvements in reducing both incorrect outputs and abnormal termination.

The second class of algorithms, Type 2 algorithms, are those that rely on program invariants and show reasonable improvements. These algorithms include grep, set union, MD5 and quicksort. These kernels show moderate vulnerability without fault tolerant methods and moderate improvement with them.

The final class of algorithms, Type 3 algorithms, are those that do not show much improvement with the added fault tolerant methods, but were hardly vulnerable to begin with.

These algorithms include BFS and MCMC. We believe this behavior to be a result of the algorithms themselves. For BFS, there may be many fault sites that do not end up affecting the behavior of the algorithm itself, as most of the algorithm is traversing pointers. MCMC on the other hand is probabilistic in nature. If an error affects one of the randomly generated values, this simply looks like more noise in the distribution, which is heavily averaged out by taking many samples.

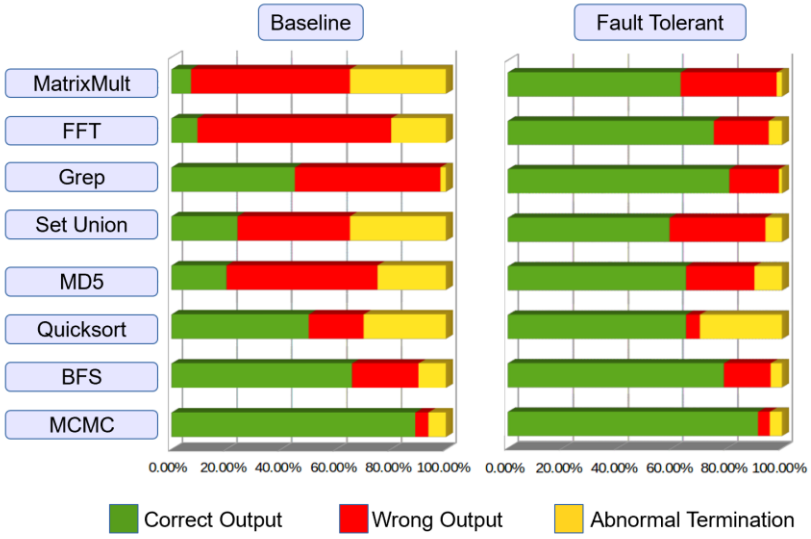


Figure 2.1. Results summary for all 8 benchmarks.

In order to provide more detail into the vulnerability of the Big Data programs, we inject a variety of fault sites, which are variables used by the program, using KULFI during execution. We then execute the program with faults injected many times and observe the frequency with which the injected faults cause errors. Using these observations, we calculate the vulnerability of a fault site as the likelihood that a fault injected into the site results in an error. This is equal to the number of occurrences of an error when injecting faults at the given fault site divided by the total number of injections performed on that fault site. These results of the fault site lifetimes are shown in Figure 2.2.

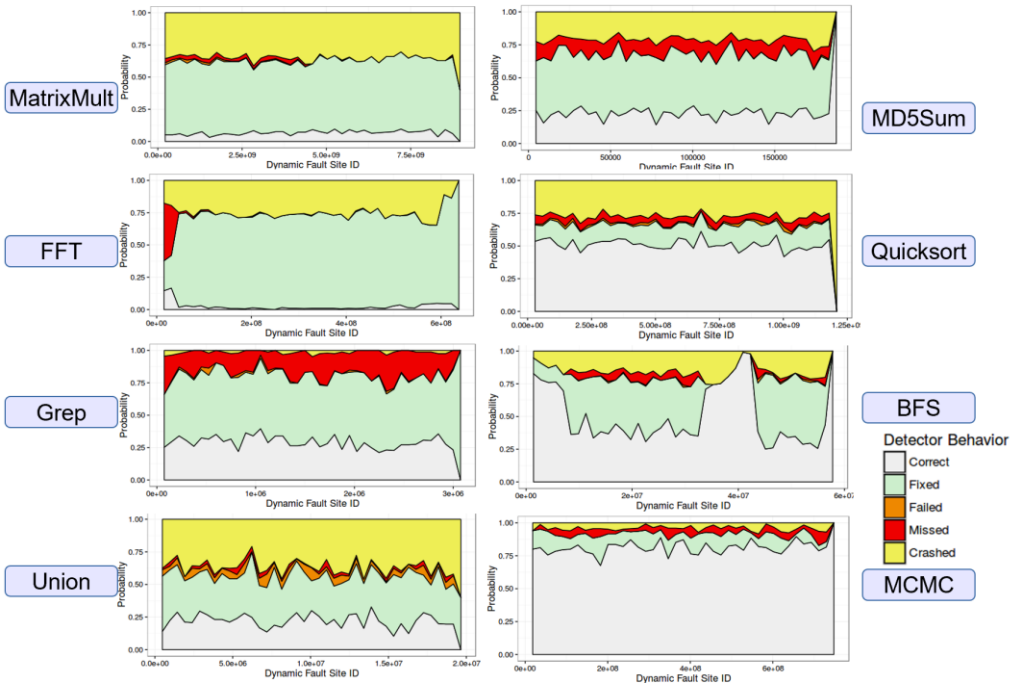


Figure 2.2. Detailed results summary for each benchmark by fault site.

Here we can see a more detailed breakdown of the kernels' execution behavior. Execution outcomes are classified into five types. Correct outcomes are simply outcomes that match the golden outcome. Fixed outcomes are outcomes where an error was detected and corrected, resulting in a correct outcome. Failed outcomes are situations where an error was detected, but correction failed, resulting in incorrect outcome. Missed outcomes are incorrect outcomes where an error was not detected. Crashed outcomes are simply cases where the program terminated due to the error.

As shown, the Type 1 algorithms, have an extremely large proportion of fixed outcomes, a small proportion of missed outcomes, and a fairly consistent proportion of crashed outcomes. This matches our expectations – the invariants do a good job of detecting and correcting most errors, unless the error simply causes the program to crash before it can be corrected. The Type 3 algorithms also show behavior consistent with the previous results – most of the outcomes never

detected any error at all, and were still correct. This is particularly true for MCMC. The remaining algorithms show a variety of behaviors, though they all indicate that the program invariants can provide a substantial increase in reliability.

To close our discussion on the invariants, we also investigated the execution time overhead for each of the kernels when introducing the fault tolerant additions. These results are shown in Figure 2.3 and Figure 2.4. The first shows the actual overhead percentage. The Type 1 algorithms show the lowest overheads by far. By taking advantage of special features of the algorithms to implement the algorithmic invariants, we can achieve the most fault tolerant improvements with the least overhead. By comparison, the programs relying on program invariants show considerably higher overhead. This is likely due to their nature as redundancy methods, and they are implementation dependent.

Figure 2.4 instead shows the expected runtime of an algorithm given relative success rate. This is calculated by assuming we would run an algorithm again after identifying a faulty run after execution. Effectively, this demonstrates how much time we save when using the fault tolerant methods compared to running the algorithm multiple times. As expected, the Type 1 algorithms again show the lowest relative execution time, indicating the highest speedup using the fault tolerant additions, while the Type 3 algorithms show that the improvements gained are not really worth the overhead costs, and in the case of MCMC, are actually worse than simply running the algorithm without the fault tolerant additions.

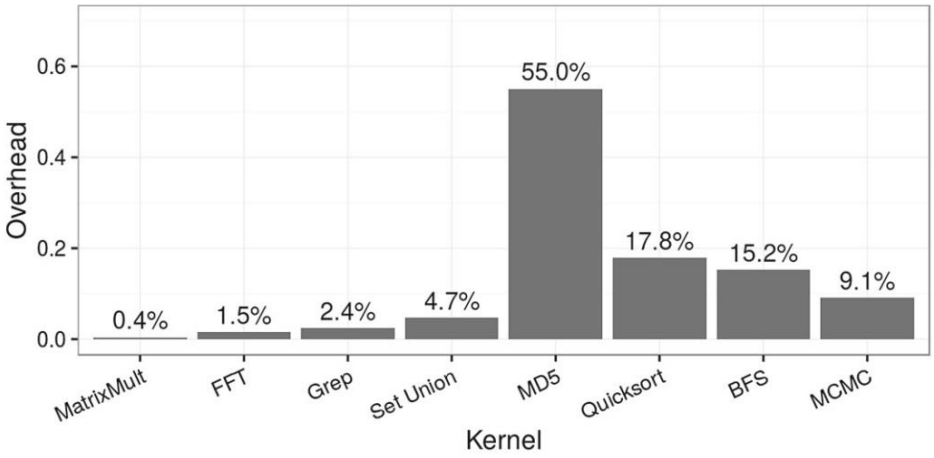


Figure 2.3. Execution time overhead for each benchmark.

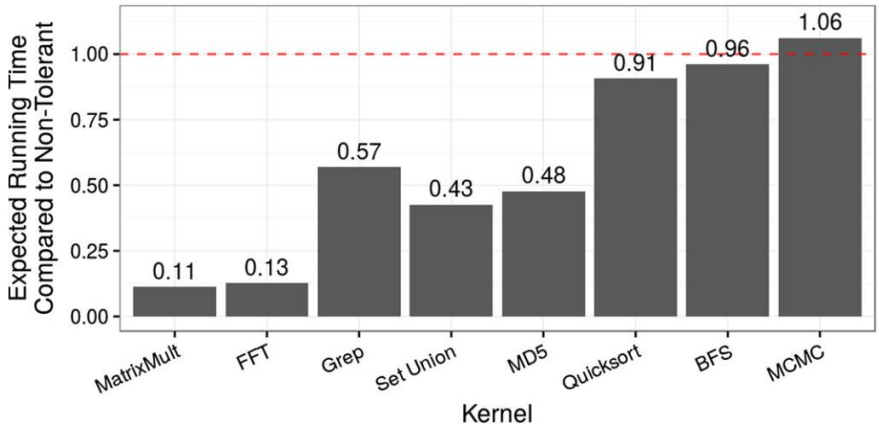


Figure 2.4. Expected runtime improvement for each benchmark when using fault tolerant additions.

2.5. Related Works

The study of error resilience and related fields such as uncertainty quantification has been mostly focused on scientific computing so far [14]. Error resilience is a must for highly unreliable environments such as on an unmanned aerial vehicle [97], especially with the increasing processing power of onboard computers.

Numerical errors are more complicated than their integer counterparts as floating-point operations are not exact and dependent on order of operation. This can be seen in parallel reduction [20] and the linear algebra routines in BLAS [48]. Arithmetic-heavy applications

including physics-based simulations are designed to cope with round-off errors [106]. With finite bit precision, floating-point operations and their results can be seen as approximations. Recently, NVIDIA began to provide half-precision (FP16) floating-point arithmetic [44] with the aim to boost performance at negligible cost of accuracy, particularly in deep-learning applications which are closely related to big data applications. From this point of view, approximation and tolerance to soft faults are very similar in nature.

The fault resilience workflow encompassing fault injection and resilience can be done on multiple levels of the hardware–software stack. Existing works have utilized actual proton sources on the physical level [10], embedded hardware sensors on the circuit level [68], FPGAs on the digital logic level [84], full-system simulators and virtual machines on the architectural level [43], and debugging utilities on the high application level [97].

Fault resilience study can be costly on the experiments side as well as on the engineering side. Several works have proposed remedies: to reduce the huge size of the fault injection experiment space, Relyzer [62] exploits control flow and value equality to prune the fault injection space; to save engineering cost on large codebases, a programming model called containment domains [18] provides developers a hierarchical view of fault resilience. Modular analysis [15] provides a first step toward lowering the cost in fault injection and the understanding of numerical error propagation. We expect to see fault resilience get increased support in the future in toolchain and modeling just like profiling and debugging do.

2.6. Conclusion

We have shown the effects of fault-tolerant code added into Big Data algorithms by experimenting with eight Big Data Dwarves as defined by the Big Data Benchmark suite. For each of the eight dwarves, we have implemented algorithm-specific invariants where applicable

to identify and correct errors in program execution. We have discussed the effects of the fault tolerant additions to each algorithm individually by evaluating error magnitudes, error probabilities, and program output behavior. Additionally, we have compared the eight algorithms to each other, resulting in three classes of Big Data dwarves: those with high-level invariants that are extremely vulnerable to faults, but show the most improvement with fault-tolerant code additions; those without high-level invariants but with moderate natural resilience, which show lower improvement rates than the first group; and those that are naturally extremely resilient, which show minor execution improvement with the fault-tolerant code additions. We have also analyzed the overhead introduced by the fault-tolerant mechanisms and quantified the expected running times for the algorithms to evaluate the benefit of the mechanisms in light of their overhead introduced, which supports the fault-tolerant performance of algorithms being grouped by level of invariant.

Together, these analyses create a portfolio displaying the resilience that fault-tolerant additions can lend to common Big Data algorithms, reducing the chances of program failure and wrong output. This additionally reduces the time and energy wasted to rerun faulty algorithms and helps avoid the danger of not detecting a wrong output which, if undetected, could cause unprecedented damage.

3. Protecting Synchronization Mechanisms of Big Data Programs via Logging

3.1. Introduction

Program reliability is a major concern throughout many fields of computing. Applications that cannot reliably produce correct solutions are hardly useful. As chip designers drive to reduce power consumption, the voltage levels separating logic 0 and logic 1 become closer and provide less of an error margin. This increases the probability of bit flips during execution, where a 0 becomes a 1 or vice versa. Depending on the locations of these bit flips, they can directly interfere with program behavior and produce unexpected results.

Errors during execution can reveal themselves in different ways, including hangs, crashes and silent data corruptions (SDC). Crashes are the most obvious, where the process simply exits suddenly. Hangs can be more deceptive as the application may still seem to be doing work while actually making no progress. Most difficult to detect during execution are SDC, where a value used by the program is modified without causing a crash or hang. For example, one of the operands for an addition is corrupted, resulting in an incorrect output value. This can become particularly dangerous when errors propagate from one variable to another as the program continues to execute [100]. These errors waste time and resources as they go undetected.

While hardware solutions like error correcting memory (ECM) exist [37], they can be expensive to apply and carry overhead. This has led to an interest in software methods for fault tolerance. These methods typically achieve good error coverage with varying overhead costs

depending on their implementation details. Software fault tolerance methods [64] for computing involve both a detection stage and a correction stage respectively for identifying and recovering from errors. In order to trigger correction mechanisms, the detection mechanism must first identify an error.

Most methods exploit features specific to the algorithms in question to identify these irregularities during execution [67, 100], or require replication and comparison of the process periodically during execution to ensure correct behavior [27]. Newer approaches relying on machine learning to identify program deviation have also been introduced [92].

Correction mechanisms typically utilize a form of checkpointing for error recovery [53, 87]. Checkpointing involves taking snapshots of the process during execution and restoring to a previous correct snapshot upon detecting an error [87]. This can be done at varying granularity and frequency based on the application. Checkpointing is a relatively simple method that works well for crashes and hangs, but can be insufficient for SDC as the error can go unnoticed and result in erroneous checkpoints. In order to alleviate this, some systems require saving multiple checkpoints and more frequent checkpointing, involving undesirably higher overhead.

These detection and correction methods are extremely important for large-scale computing, where processes may run for many hours or even days on multiple nodes. If an SDC occurs early during execution, the algorithm could run for a long duration before the error is noticed, wasting substantial time, resources and energy. These programs are typically parallel in nature, employing fundamentally different techniques to solve problems. They commonly rely in part on synchronization mechanisms such as locks and barriers for sharing information among threads and ensuring coherence. However, these mechanisms themselves can be vulnerable to errors, leading to error behavior that occurs only within parallel programs. To our knowledge,

there is little previous work aimed at protecting these synchronization mechanisms from transient faults. Application checkpointing systems can solve crashes and hangs resulting from errors in synchronization mechanisms but require additional detection for SDC. Transactional memory has been proposed as a method to protect code executions from concurrency bugs [94], though its focus is on programmer errors not transient faults.

In this work we present a method for identifying and correcting violations of these synchronization mechanisms caused by transient faults via local logging systems. Tracking thread locations during execution reveals violations of the synchronization mechanisms. We implement a local checkpointing and recovery mechanism for the threads through Intel Pin [51] by exploiting the conceptual properties of these mechanisms. We include an investigation into the results of faults within these synchronization components to demonstrate the effectiveness of such methods, and a measurement of the overhead costs for implementation. Finally, we provide a comparison with transactional memory, another form of local logging and rollback for parallel systems that can act as an alternative for lock-based mechanisms. Our system implements similar logging mechanisms to an eager transactional memory system, but it benefits from simplified conflict detection when fewer conflicts can occur. Note that our mechanism also differs from conventional checkpointing in that it conducts logging at each parallel control structure (locks and barriers) in preparation for rolling back, if required, as opposed to collecting system execution states periodically or adaptively [87] under conventional checkpointing.

The contributions of this work can be summarized as follows:

- We examine vulnerability of BigDataBench kernels [12] to soft faults within concurrency control mechanisms during execution.

- We design and develop a logging mechanism based on transactional memory to detect and correct the resulting concurrency bugs by enforcing the control mechanisms.
- We demonstrate a mean 93.6% error coverage from the resulting concurrency bugs caused by these soft faults with a mean 6.55% overhead in the execution time at 64 threads.
- We compare the overhead of our developed logging mechanism against a full transactional memory system and find up to a mean 57.5% reduction in execution time overhead relative to transactional memory.

3.2. Background

3.2.1. Concurrency Control

As previously mentioned, many fault-tolerant methods exploit program features to increase coverage and reduce overhead. We focus specifically on locks and barriers as our synchronization structures. These fundamental mechanisms provide building blocks for more complex parallel data structures. However, these locks and barriers perform different functions and present different vulnerabilities. Locks protect critical sections of code, where only one thread should enter at any given time. Violations can cause race conditions where multiple threads access values at the same time. Failing to unlock locks, or poorly coordinating the order with which a thread claims multiple locks, can also lead to deadlocks, halting execution progress. Locks can be employed in either a fine-grained or coarse-grained manner. Coarse-grained locking protects a large amount of code that may not all be needed by the thread. It is easier to implement at the cost of performance, as more threads compete for the critical section. Fine-grained locking by comparison enables more parallelism by locking small sections of code that

are specifically necessary for the thread, but it is difficult to implement and may be more prone to deadlocking.

Barriers by comparison act as a trap, where no thread is allowed to pass until all involved threads have reached the barrier. Barriers are commonly used with an alternating computation and communication paradigm. When a thread finishes computation and needs to share information, each thread waits until every thread has completed computation and is ready for sharing. This prevents threads from overwriting values that are still needed by other threads, or reading old values that are no longer valid. Violations of the barrier would cause threads to sneak past, potentially causing these problems. Both locks and barriers are typically implemented using atomic operations that allow a thread to perform a combination of reads and writes as one single operation, ensuring coherence among multiple threads operating on a shared value. The most common example is the compare-and-swap (CAS), which compares a value *m* in memory to a given value *v*, and writes a third value to memory if *m* and *v* are equal.

3.2.2. Transactional Memory

There are other methods to ensure thread coherence besides directly using locks and barriers. The most relevant to note here is transactional memory. By automatically fine-grained locking individual memory locations, developers do not need to manually implement locking mechanisms. Instead, a thread simply marks the beginning and the end of a transaction, wherein all operations will be executed as if they were atomic. If there are conflicts due to multiple threads modifying the same memory locations, one thread is chosen to commit its transaction while others are forced to reattempt. These transactional memory-based systems provide an alternative for developers, potentially allowing for greater parallelism in their fine-grained nature. By being aware of how threads operate on a memory location (read vs write), such a

system can also allow multiple reads simultaneously as memory is then not modified. Transactional memory has previously been adopted to address concurrency bugs which result from developer errors but are not transient faults [94].

Transactional memory systems have been implemented both in hardware [58] and in software [31]. Both implementation methods have their respective benefits, with hardware systems typically having better performance in exchange for flexibility and simplicity. Transactional memory systems have also been proposed for accelerators like GPUs [17, 34]. As shown in the following sections, we utilize methods similar to eager transactional memory to protect coarse-grained locks and barriers. While similar to transactional memory, it is considerably simpler due to a limitation in the types of conflicts that may arise.

3.3. Motivation

It is important to note that transient errors in these parallel programs may differ considerably from those found in sequential programs. In sequential programs, faults may cause crashes, hangs or incorrect output by modifying pointers, loop control structures or variables holding important data. In parallel programs, crashes, hangs and SDC can all result from faults targeting parallel control structures like locks and barriers. For example, a fault that occurs in data used within or leading up to “xchg" or CAS instructions may cause the synchronization mechanism to fail. These failures can result in crashes, hangs, or SDC when threads violate concurrency control, either through race conditions in critical sections or accessing improperly synchronized data. SDC caused by these failures can further propagate into different errors, which may be difficult to identify, locate and recover from. Due to the different nature of these errors, we make the first attempt to detect and correct them in non-traditional manners, as discussed in Chapter 3.2.

We aim demonstrate the importance of protecting concurrency control mechanisms in parallel applications by examining the vulnerability of three representative benchmarks of the BigDataBench kernels under 2, 4, and 8 threads. The full list representative benchmarks for the BigDataBench kernels is shown in Table 2.1. We simulate transient faults using the Intel Pin Fault Injector (PINFI) [52]. As the errors can occur in different locations in each trial, we simulate transient faults rather than hardware faults. This automated fault injection tool targets instructions within select functions and modifies the bits to simulate soft errors during execution. For these experiments, we limit injection to the synchronization mechanisms contained within the programs. We inject a single fault into every lock encountered during execution. Hence, the number of faults injected for one trial equals the number of locks encountered over the course of execution. The results from these trials are shown in Figure 3.1.

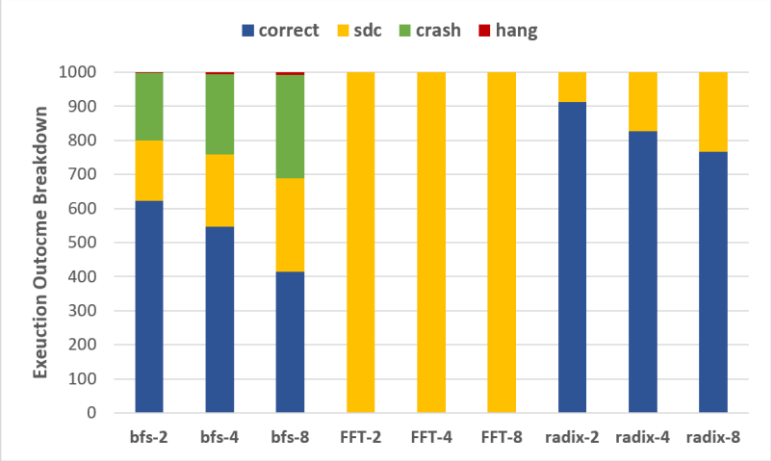


Figure 3.1. Execution outcome breakdown under 2, 4, 8 threads.

It is evident that as the number of threads increases, error frequency also increases. Having more threads in contention for the control structures results in more CPU time spent waiting at these structures during execution. As more instructions are executed involving these wait loops, it becomes more likely for errors to break these loops and thus break the control

structures. It is worth noting the variety of error profiles among benchmarks. FFT is completely vulnerable to SDC, while radix is only mildly vulnerable. In comparison, BFS contains vulnerabilities not present in the other two, including a sizable number of crashes and some hanging executions. These outcomes are caused by soft fault injection into the control structures which can cause race conditions, deadlock, or direct crashes as shown.

The observed outcomes depend heavily on the algorithm itself. Programs like BFS that rely heavily on pointers experience more crashes as errors due to failed synchronization may corrupt these pointers and result in erroneous memory access. Conversely, programs that include many logical or arithmetic instructions like FFT are more vulnerable to SDC as errors due to failed synchronization are more likely to simply modify data and not cause crashes. This observation supports other works in that the use of algorithm-specific methods for detection and correction throughout the entire execution, may be more efficient than generic methods, confirming the importance of protecting locks and barriers.

3.4. Design

3.4.1. Detection

In order to detect these violations, we implement a logging mechanism through Intel Pin [51] similar to an eager transactional memory system. The tool identifies marked barriers and locks in the binary and tracks their program locations during execution. This allows us to identify at what points which threads reside in critical sections or beyond barriers. Whenever multiple threads are detected within a critical section simultaneously, we know that its associated lock has been broken by faults. Similarly, if a thread ever passes a barrier before other threads are able to reach it, we know that the barrier had broken. Figure 3.2 provides an example of both successful and recovered execution paths.

Algorithm 3.1. Lock Error Detection and Correction

Input: The instruction pointer p for the entry point of the critical section, the current thread id tid, and a dictionary of lock ownership status S

1: **Atomically** attempt to set $S[p] = tid$
2: **if** $S[p] = tid$ **then**
3: Current thread successfully marked as owner, proceed to the critical section
4: **else**
5: Send rollback signal to $S[p]$
6: Clear $S[p]$
7: Roll back to p to reattempt lock acquisition
8: **end if**

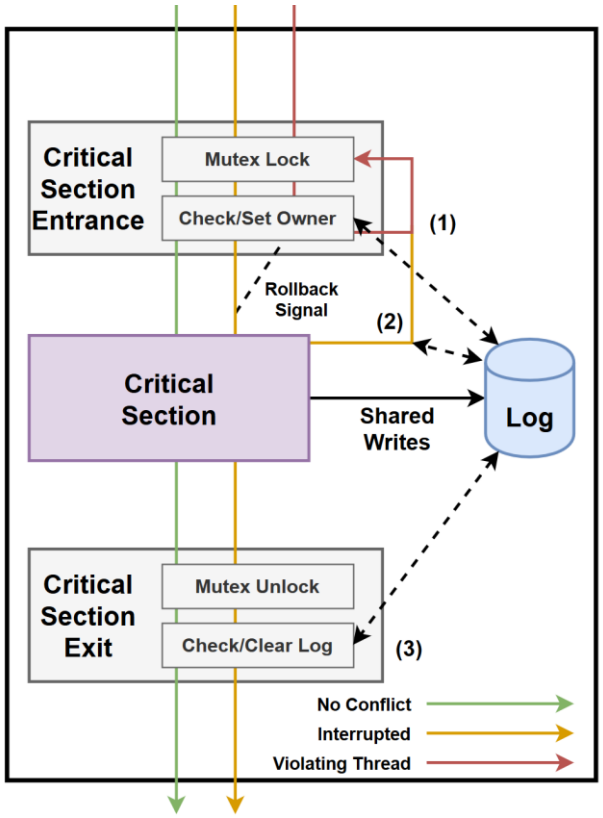


Figure 3.2. Execution flowchart with and without conflict.

Before entering a critical section, a thread must pass through both the original lock and the following protection functions, marked as **1** in Figure 3.2. The log tracks a thread's entrance to the critical section and executes Algorithm 3.1 to detect if the thread violates the exclusivity of

the critical section. The green line shows a thread which executes without interruption. The orange line (thread 1) shows a thread that is interrupted whereas the red thread (thread 2) erroneously breaks the lock and enters the critical section. When thread 1 enters the critical section first, it is marked as the owner of the lock within the log and can progress into the critical section. When thread 2 enters the critical section before thread 1 has exited and released ownership, the logging system is aware that the lock has broken for thread 2 or thread 1 and thus correction is attempted.

Additional work is necessary to ensure locks are correctly unlocked when leaving a critical section. This is different from atomicity violations where multiple threads enter the critical section, possibly leading to deadlocks and program hangs instead since no threads are then able to enter the critical section. To address this form of fault, we track which locks are owned by which threads. Upon exiting a critical section, if a lock is still owned by any thread no longer within the associated critical section, we can correctly identify the occurrence of an error, resulting in failure to release the lock. This occurs in stage **3** of Figure 3.2.

A similar method is used for detecting errors within barriers as illustrated in Algorithm 2. Note that all updates to counters are performed atomically to avoid race conditions. By inserting functions directly before and after a barrier, we can identify when a thread enters and exits the barrier. If any thread attempts to exit the barrier before all involved threads have reached the barrier, it has violated the expected behavior of the barrier and is identified as an error. This requires knowledge of the number of threads that are involved in the barrier, which can be given as a parameter, collected from the initialization of the barrier object, or assumed by default to be the number of threads in use by the process. If the barrier in question can be encountered

multiple times, additional checks are performed before Line 2 to ensure re-entering threads do not interfere with exiting threads.

Algorithm 3.2. Barrier Error Detection and Correction

Input: The instruction pointer p for the barrier and the number of threads involved in the barrier n

1: Initialize *entrance_counter* to n, exit counter to 0
When a thread attempts to enter the barrier:
2: Decrement *entrance_counter*
When a thread attempts to exit the barrier:
3: **if** *entrance_counter* is not 0 **then**
4: Roll thread back to the barrier and wait
5: **else if** *exit_counter* is not $n - 1$ **then**
6: Increment *exit_counter*
7: **else**
8: Reset counters to initial values
9: **end if**

3.4.2. Correction

Having detected the presence of errors, a thread can attempt local recovery via a rollback. This local recovery is similar to an aborted transaction in transactional memory systems. For barriers, rolling back is simple as errors are detected before threads can modify shared memory. When a thread is found to be exiting the barrier before all other threads have arrived, it is rolled back and forced to wait. When all threads arrive, the offending thread can then exit correctly together with all others. An example of both correct and erroneous executions is shown in Figure 3.3, where **1** and **2** mark the entry and the exit stages respectively. This reinforces the conceptual behavior of the barrier to ensure proper synchronization.

The process is somewhat more complex for locks as threads may modify shared variables. Additionally, we cannot be certain which thread within the critical section entered erroneously. Thus, both are rolled back and the logged writes are cleared, marked as **2** in Figure 3.2. Only one thread has reached the point of modifying shared variables, so the rollback is

relatively simple. This means that the log does not have to store backup values for a memory location for each thread. Since only one thread has been modifying shared variables, it can simply restore previous values while other threads reattempt the lock.

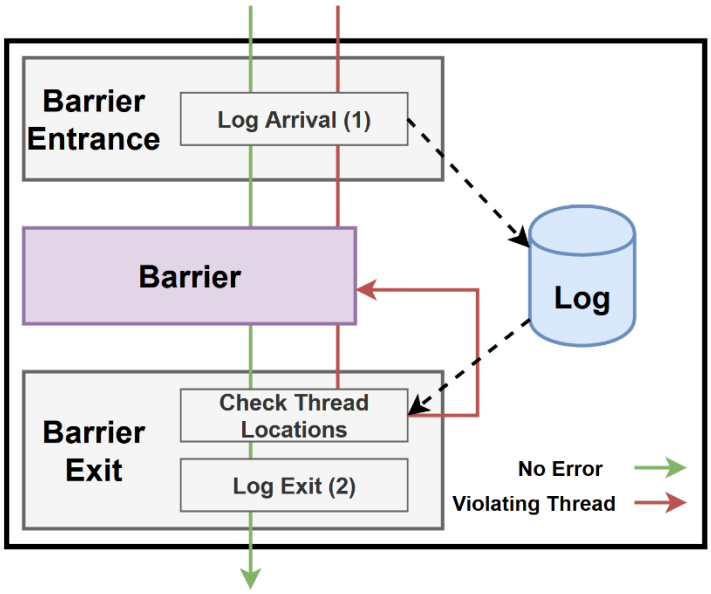


Figure 3.3. Execution flowchart showing correct and erroneous threads.

Due to our method of conflict detection, we do not encounter situations where two threads can both modify a shared variable before the conflict is detected, which simplifies the logging and rollback processes. We recommend using re-entrant locks in conjunction with our system, as the correct owner will then be able to re-enter the critical section after a rollback. Upon successfully exiting the critical section, thread information is cleared from the log, allowing other threads to enter. Ultimately, this mimics the conceptual function of the lock to ensure the correctness of the critical section. In summary, each of these correcting methods enforces the associated control structure behavior, thereby preventing the propagation of SDC.

3.4.3. Examples

For clarity, we provide two following examples to cover both thread and barrier encounters. Both examples will utilize two threads, thread **A** and **B**, to showcase the protection and correction mechanisms.

Locks: Suppose thread **A** encounters a lock at instruction *p*. The thread attempts to lock the lock, and our system attempts to claim ownership of the lock for thread **A** in line **1** of Algorithm 3.1. If thread **A** successfully claims ownership (line **2**), we assume thread **A** has appropriately locked the lock and can continue with the critical section (line **3**) while logging the usage of shared variables. Assume thread **B** encounters the lock while thread **A** is in the critical section. It is possible that thread **B** passes the lock entrance due to a fault in its or thread **A**'s locking process. Either way, the fault is detected in line **2** when thread **B** fails to take ownership. Both threads are then forced to reattempt acquisition of the lock in lines **5-7**, rolling back any changes made by thread **A**.

Barriers: Suppose thread **A** encounters a barrier expecting two threads at instruction *p* before thread **B**. When thread **A** enters the barrier to wait, marked as **1** in Figure 3.3, *entrance_counter* is decremented from *n=2* to *n=1* as shown in line **2** of Algorithm 3.2. Thread **A** should not pass this barrier until thread **B** arrives to ensure proper synchronization. If an error occurs to cause thread **A** to pass the barrier early, the check in line **3** succeeds and line **4** is executed, forcing thread **A** back to waiting. When thread **B** arrives at the barrier, we again execute line **2**. Both threads attempt to exit the barrier, this time failing the check on line **3**. Both threads can thus exit the barrier, one executing line **5-6** to increment the *exit_counter* and the other simply exiting and resetting the initial values.

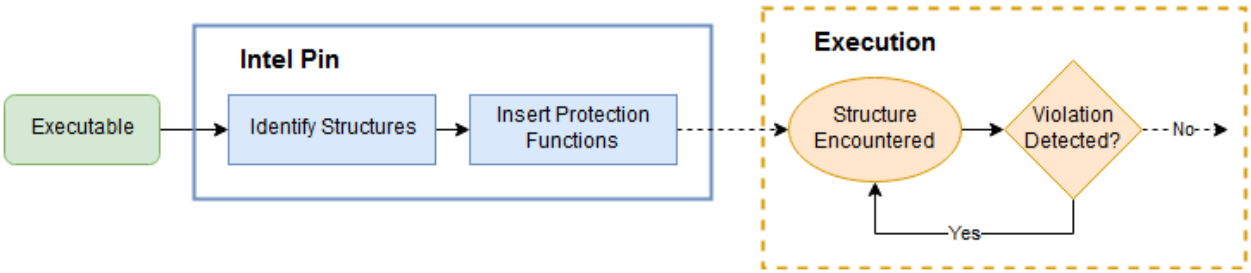


Figure 3.4. Design workflow with Intel Pin.

3.4.4. Implementation

For testing, we implement this method via Intel Pin as a Pin tool, which makes it flexible to work with any binary compiled for Intel processors. Intel Pin allows for both static and dynamic analyses and modifications of a program. As such, not only can we implement the transaction begin and end functions statically before execution, we are also able to track program locations and the control structure status dynamically during execution. Figure 3.4 shows the overall workflow of the tool. Upon loading the binary, Pin applies the tools to the binary in two steps, called the instrumentation and the inspection passes. Instrumentation traverses the program statically to identify any instructions of interest, namely those related to the locks and barriers we aim to protect. It then inserts inspection functions into the binary that will be executed during run-time. At run-time, these functions intervene in the program execution, carrying out the logging methods as necessary to track the program status. Specifically, we locate every lock and barrier used by the program and add protection functions to each of them. These protection functions initialize the logging system with the program counter and thread information. This allows the logging system to detect the violations of the associated synchronization mechanisms.

Although our implementation is purely software, it could be augmented with hardware support. Our synchronization protection mechanisms need the additions of (1) an on-chip lock ownership directory, whose entries, say *S[p]*, record the ID of the thread entering Lock *p*; see

Algorithm 3.1, (2) an on-chip SRAM partitioned statically into zones, with Zone p for holding the log associated with Thread *S[p]*, and (3) control logic for generating appropriate control signals and maintaining ownership directory entries. Both (1) and (2) are in the form of on-chip SRAM to improve performance. Additional instructions, similar to previous atomic instructions, could also be included to manage the lock ownership and logging operations involved with these added on-chip SRAM zones.

3.5. Evaluation

3.5.1. Vulnerability and Resilience

In order to test the effectiveness of our system, we execute the benchmark programs both with and without our protection mechanisms. All experiments have been run on a workstation with two Intel Xeon Platinum 8260 processors which support up to 48 threads each when enabling Hyperthreading, resulting in 96 total available threads. We test only up to 64 threads as some benchmarks require the thread count to be a power of two. As we have displayed in Figure 3.1, higher thread counts result in greater vulnerability, so unless otherwise noted, our experiments use the full 64 threads possible on our test machine. Once again, we use PINFI [52] to simulate transient faults by injecting one fault into each lock encountered during execution. For the following experiments, we restrict fault injection to both the synchronization code regions and our added protection code regions where applicable. We must inject into our added protection mechanisms to properly evaluate the vulnerability of the final system. This prevents full error coverage as the added code itself is vulnerable, although to a lesser extent.

The BigDataBench benchmarks and their categories [12] are listed in Table 2.1. Specifically, we test the Fast Fourier Transform (FFT), LU matrix decomposition (LU), radix sorting (RADIX), graph operations (BFS), sampling operations (MH, Metropolis-Hastings

implementation of the Markov chain Monte Carlo method), WordCount (WC), and set union (UNION). We have chosen not to test the Logical Operations category of the BigDataBench suite as its many samples are intrinsically sequential. Therefore, we have covered 7 out of 8 categories of parallel BigDataBench kernels. Both the baseline and protected versions of each benchmark are run through Intel Pin to provide a proper comparison between the two test cases with the maximum number of threads. The protection mechanisms are simply disabled in the baseline case. All benchmarks are executed with 64 threads. Our results are displayed in Figure 3.5 and Table 3.1.

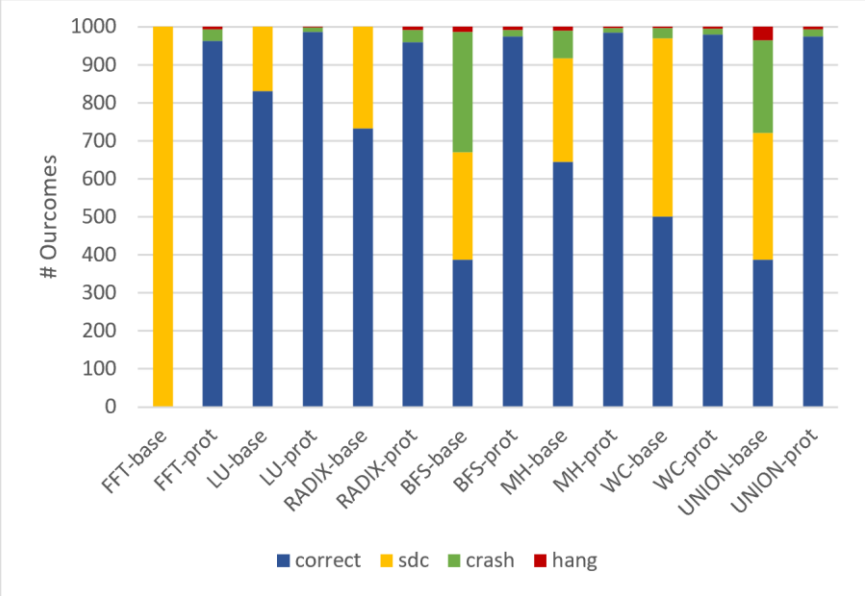


Figure 3.5. Execution outcome breakdown under 64 threads.

Table 3.1. Execution outcomes for each benchmark with and without protection. A total of 1000 trials have been executed for each benchmark under 64 threads. ER % represents the percentage of reduced error compared across both versions of the benchmark.

Category	Benchmark	Outcome				ER %
		Correct	SDC	Hang	Crash	
Transform Operations	FFT-base	0	1000	0	0	
	FFT-prot	962	0	31	7	96.2
Linear Algebra	LU-base	831	169	0	0	
	LU-prot	986	0	12	2	91.7
Sorting	radix-base	733	267	0	0	
	radix-prot	959	0	32	9	84.6
Graph Operations	bfs-base	387	283	316	14	
	bfs-prot	974	0	17	9	95.8
Sampling	mh-base	645	272	73	10	
	mh-prot	984	0	13	3	95.5
Statistics Operations	wc-base	501	469	27	3	
	wc-prot	980	0	13	5	96.4
Set Operations	union-base	387	334	244	35	
	union-prot	975	0	18	7	95.9

As shown previously in Figure 3.1, different benchmarks have different vulnerability profiles. FFT, LU and radix sort show varying degrees of SDC vulnerability, while the remaining four display considerable numbers of SDC and crashes. However, it is evident that the protection system removes almost all occurrences of crashes, hangs and SDC during the execution of these programs by correcting the soft faults within the control structures. It achieves a mean 93.6% error coverage across all kernels, with RADIX having the lowest coverage of 84.6% and WC having the highest coverage at 96.4%. We believe RADIX and LU show lower error coverage as they are already more resilient to errors and therefore there are fewer to correct. Interestingly, the protected benchmarks only contain crash and hang errors without any SDC occurrences. These crash and hang errors are preferable over SDC as they are more easily detectable and correctable during execution. Note that we do not claim that this method will address all possible errors that can occur in the program in general. Rather, the method focuses only on errors within the synchronization mechanisms, with errors beyond these synchronization structures deemed

outside the scope of this work. By reducing these errors, we prevent error propagation into other forms that may be more difficult or costly to detect with other methods.

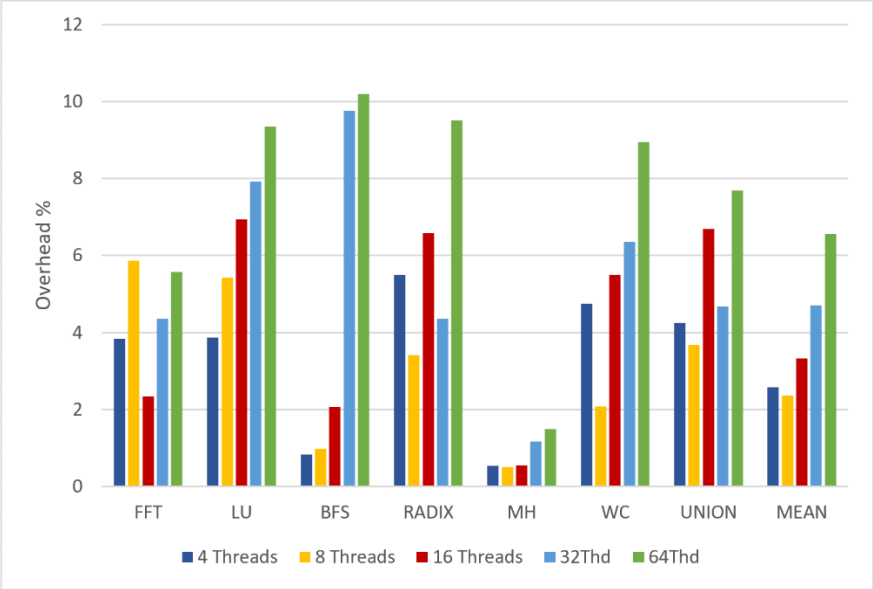


Figure 3.6. Execution time overhead.

3.5.2. Overhead

To properly compare the cost of our system, we also investigate execution time overhead incurred by the implemented protection mechanisms on each benchmark under 4 to 64 threads. Both the baseline and the protected version are again executed through Intel Pin to provide an accurate comparison of the overhead caused by the system itself. We calculate the execution time overhead as $\frac{(T_{prot}-T_{base})}{T_{base}}$ where T_{prot} is the average execution time of 1000 trials of the protected benchmarks, and T_{base} is the average execution time of 1000 trials of the unprotected benchmarks. These results are shown in Figure 3.6, where most benchmarks are found to have an overhead of less than 10% at all thread counts, with a mean overhead of at most 6.55% under 64 threads. These overhead levels are acceptable considering the improvement in error occurrences and the complete removal of SDC errors. It is clear that both BFS and LU have somewhat higher

overhead than the others at many thread counts. BFS is a larger application with greater memory requirements, leading to relatively higher overhead when experiencing context switching. Both BFS and LU also contain more complex and frequent concurrency control use, resulting in a higher accumulated overhead. As a result, it may not be wise to apply the protection mechanisms to every program; instead, they should be applied on an algorithm-to-algorithm basis.

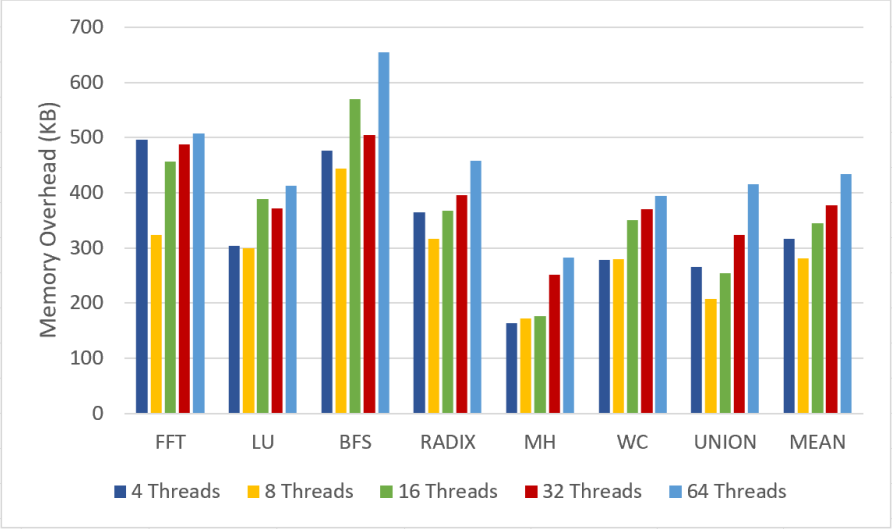


Figure 3.7. Memory consumption overhead.

For further evaluation of the system, we measure the memory overhead of our system on all benchmarks under 4 to 64 threads, as shown in Figure 3.7. To calculate this, we record the memory high-water mark measured from within the program during execution. Note that the memory overhead is very small relative to the total amount of memory used by the programs. Most benchmarks use a maximum of 0.6-2.5GB memory during execution. As such, the overhead of 300-700KB is relatively negligible. As expected, we see larger overheads at higher thread counts.

3.5.3. Comparison with Transactional Memory

We additionally provide a comparison against complete software transactional memory systems, since our protection mechanism relies on similar checkpointing and rollback operations. Specifically, we test against the C++ atomic library and its included transactional memory interface. We do not compare with hardware transactional memory systems, which are incomparable to our software-based protection mechanism. Specifically, since we enforce the high-level behavior of locks, our system handles entire critical sections in addition to logging individual memory locations. As such, we can frequently detect conflicts when threads first enter a critical section rather than at every individual memory access. Given that our logging mechanisms are less intrusive than full transactional memory, they should therefore demonstrate lower overhead. To test this, we modified kernels for evaluation, with the results for LU and FFT shown in Figure 3.8 and Figure 3.9 respectively.

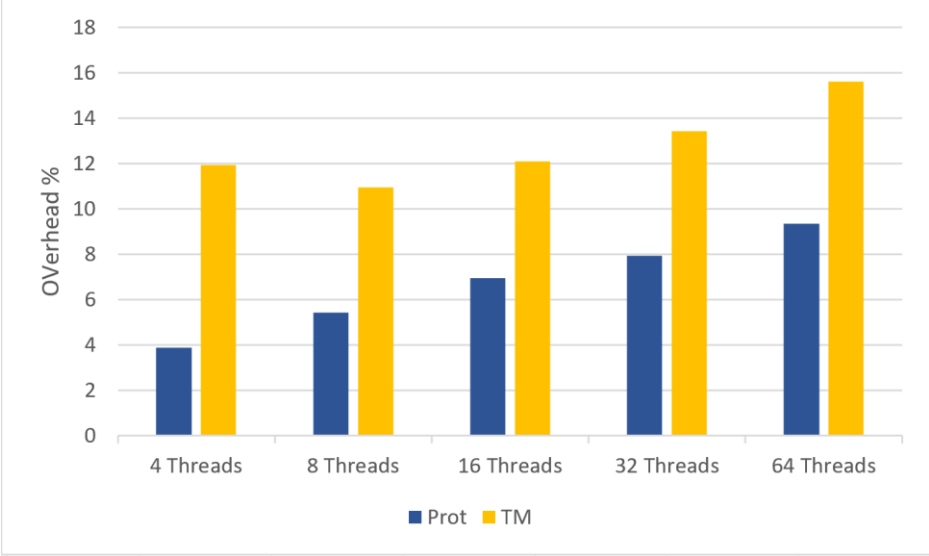


Figure 3.8. Comparative time overhead of our protection mechanism (prot) and transactional memory (TM) for the LU benchmark.

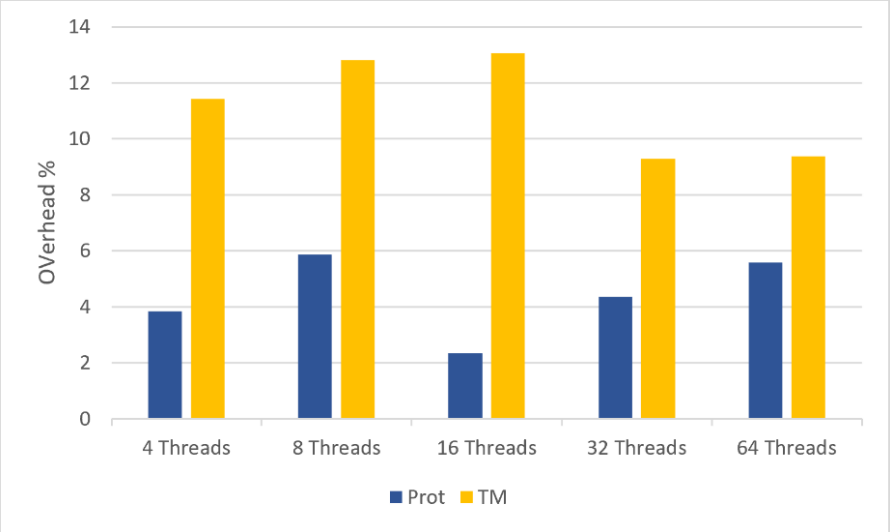


Figure 3.9. Comparative time overhead of our protection mechanism (prot) and transactional memory (TM) for FFT.

Our baseline protects the shared variables using standard *pthread* locks and barriers for comparison against both our protection mechanism and the transactional memory implementation. We test the programs under 4 to 64 threads to gain insights into how the systems handle varying numbers of parallel agents. According to Figure 3.8 and Figure 3.9,the largest gap in execution time overhead percentages occurs for FFT under 16 threads, with a difference of ~10%. In total, we observe a geometric mean reduction of 47.3% and 57.5% in overhead for LU and FFT respectively.

As we can see, the log-based protection mechanism consistently outperforms its transactional memory system counterpart at each thread count. While not displayed here, comparative overhead results for the remaining tested benchmarks exhibit similar performance gaps. This supports our previous claim that the protection mechanism is more lightweight than full transactional memory systems, resulting directly from simplifying many of the conflicts it must handle.

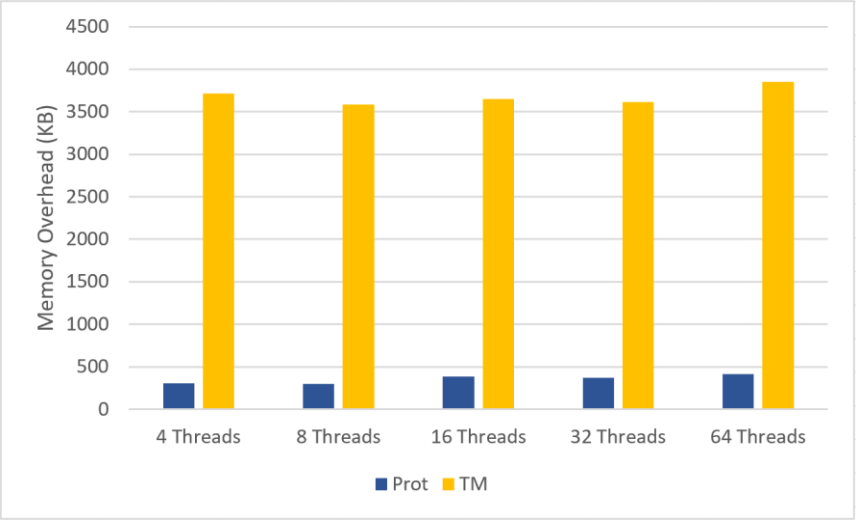


Figure 3.10. Comparative absolute memory overhead for LU.

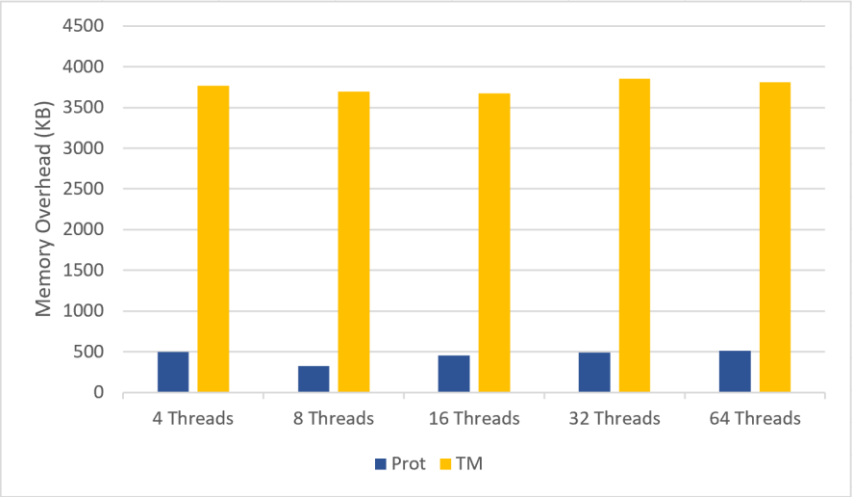


Figure 3.11. Comparative absolute memory overhead for FFT.

We also compare the memory usage for both our system and the transactional memory implementation, shown in Figure 3.10 and Figure 3.11. At all thread counts, our system consistently uses considerably less memory than the transactional memory implementation. Since our system can resolve conflicts sooner due to detecting the higher-level concurrency failures, it does not have to log as many values at one time for potential rollbacks, reducing the

total memory used. Again, note that these values are still small relative to the total memory consumption of these benchmarks.

3.6. Conclusion

In this work we have presented a method for ensuring the correct and reliable operation of synchronization structures within parallel programs, specifically focusing on locks and barriers. By utilizing a logging system that tracks program locations, we can identify violations of these structures and recover from them locally, rather than requiring system wide checkpointing and recovery methods. Through our experiments, we demonstrate that this method can achieve a reduction in error of up to 93.6% for the representative BigDataBench kernels while maintaining acceptably low overhead, averaging 6.55% above the baseline. When compared with transactional memory, we find up to a 57.5% reduction in execution time overhead.

4. Background on Quantum Computing

4.1. Introduction

Quantum computing is a relatively new computing technology that aims to solve problems that are traditionally intractable on classical computers, such as factoring large numbers, in polynomial time. There are many misconceptions about the operations of quantum computers and the problems to which they can be applied. Although quantum computers offer a unique opportunity for performance improvements, they are not without their limitations. They cannot simply solve every difficult problem for free. This chapter provides an overview of quantum information theory and the quantum computers available today, including both the benefits and weakness that ongoing research aims to address.

4.2. Information Theory

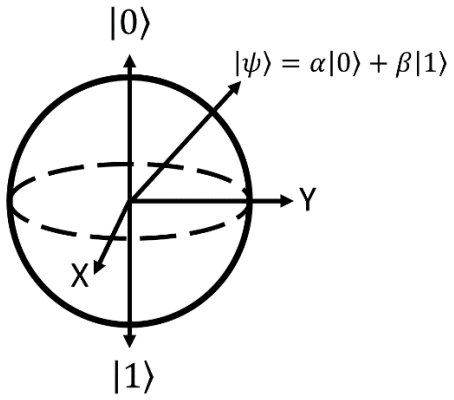
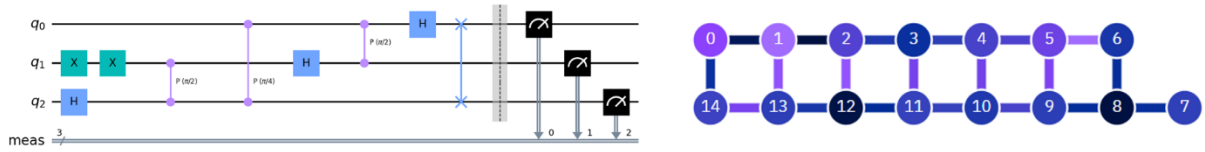


Figure 4.1. Bloch sphere representation of a qubit.

Quantum computers are composed of a number of qubits, or quantum bits. Similar to bits, qubits contain both a 0 and 1 state, commonly written as $|0\rangle$ and $|1\rangle$ to represent the high and low energy states, respectively. These states are known as the computational basis states. A qubit that is limited to $|0\rangle$ and $|1\rangle$ states is very similar to a classical bit. However, unlike classical bits, qubits can also take on any linear combination of these two states, represented as $\alpha|0\rangle$ +

$\beta|1\rangle$. Here, alpha and beta come from the set of complex numbers. In total, this gives four parameters to define a quantum state. As only the difference between the phase components of alpha and beta is relevant, we can visualize the state of a qubit as a three-dimensional sphere. This representation, shown in Figure 4.1, is called the Bloch Sphere. It is commonly used to visualize qubit states and operations on a qubit. The poles of the sphere represent the two basis states, while any other vector on the surface of the sphere represents a superpositioned state.



(A) Quantum Circuit **(B) IBM Quantum Backend**

Figure 4.2. (A) An example quantum circuit. (B) A sample SQC qubit mesh.

If the state of a qubit can be represented as any vector on the surface of the Bloch sphere, then quantum gates are represented as angular rotations of said vector. Every quantum computer requires a set of basis gates, commonly a selection of single- and two-qubit gates, from which all other gates can be composed. The simplest gate, an X gate, analogous to the classical NOT gate, simply switches the alpha and beta coefficients, effectively flipping the vector around the x-axis. There are similar Z and Y gates for flipping around their respective axes. There are also compound rotation gates, like U3, which rotates the state with an arbitrary rotation around all three axes, similar to a rotation matrix in computer graphics. Another very important gate, the Hadamard (H) gate, maps a qubit from a basis state to a perfect superposition $|0\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. There are also multi-qubit gates, most notably two-qubit gates like the controlled not (CNOT) gate. The CNOT conditionally applies an X gate to the target qubit if the control qubit is

a $|1\rangle$. This can be used for computation or for creating entanglement between qubits. Gates involving more than two qubits do exist, but are commonly decomposed into combinations of single- and two-qubit gates due to the complexity of implementation. Lastly, measurement gates are used to read the value of a qubit out into classical registers. Sets of operations on qubits are combined together to form the quantum analog of a program, commonly referred to as a quantum circuit. An example quantum circuit is shown in Figure 4.2A with individual gates labeled. Each horizontal line represents the lifetime of the associated qubit throughout execution.

4.3. Quantum Advantage

Quantum technology takes advantage of three important features that allow for improvements over classical technology: superposition, uncertainty, and non-locality (entanglement). Superposition of a single qubit has already been discussed above. However, the true advantage of superposition comes from having multiple qubits that are each in superposition. When considering a multi-qubit environment, the global state is represented as the tensor of individual qubit states. When each of these qubits is in perfect superposition, the global state captures all possible combinations of the basis states for all qubits. This lays the groundwork for algorithms such as QFT that allows for processing on multiple states simultaneously.

Hand-in-hand with superposition is uncertainty. A qubit that is in a superpositioned state simultaneously represents multiple classical states. In order to extract information from a qubit, one must measure the qubit into a classical bit. This measurement collapses the superposition into one of the basis states. The probability of measuring a $|0\rangle$ ($|1\rangle$) given a state $\alpha|0\rangle + \beta|1\rangle$ = $\alpha^2(\beta^2)$. This is important to note - we cannot use the extra power of a qubit for free, nor can we know in advance the result of a measurement on a qubit. This creates an interesting situation

where, even without errors involved, it is possible to measure a set of qubits and receive incorrect results due to probability. However, this is not purely a detriment. Due to the true random nature of qubits, we can use quantum uncertainty to power random number generators. The only limitation on the randomness of the resulting distributions is the precision with which one can prepare the quantum states before measurement. Thus, quantum uncertainty should be regarded as a tool that one must consider, compensate for, and leverage depending on the application.

The last mechanism is quantum non-locality, also known as quantum entanglement. Entanglement requires two or more qubits to be prepared into certain states (superpositions) and then made to interact with each other. The result is that the pair of qubits enter a state where measurements to one qubit can affect the other qubit instantaneously regardless of physical distance, connections, or barriers. The simplest entanglement can be demonstrated using a $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ qubit, q_1, and a $|0\rangle$ qubit, q_2. The multi-qubit state can be represented as $\frac{1}{\sqrt{2}}(|00\rangle + |01\rangle)$. If we perform a CNOT operation with q_1 as the control qubit and q_2 as the target qubit, our resulting state is $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. We can see now that the uncertainty present in q_1 is shared with q_2, and a correlation has formed between their states. If we were to measure either qubit and receive a 0 (1), we are guaranteed with 100% probability to collapse the other qubit into the same state. Entanglement forms the basis of most quantum encryption and communication methods that have been proposed. Additionally, these correlations allow for a completely new method of representing and processing data that is still being explored. However, entanglement is still extremely new and not fully understood.

4.4. Physical Realizations

Quantum computers can be realized with many different physical quantum systems, including superconducting circuits, photon polarization, electron spin, or trapped ions. Each system requires different physical components to represent the qubits themselves, to operate on the qubits, and to measure information back from the qubits into classical data. Currently, the most popular quantum computing technology is superconducting quantum computers (SQC), as these devices can take advantage of the mature semiconductor industry. Additionally, they generally have shorter gate times, leading to lower overall execution times. As they are superconducting circuits, they require being cooled down to near 0 Kelvin temperatures, making them difficult to produce and access. However, due to their greater accessibility and popularity, we will focus on SQC for this work. Unless otherwise noted, any following information pertains directly to SQCs and may not apply to other implementations of quantum computers.

SQCs are built using Josephson junctions to represent the qubits. The qubits are operated upon using microwave pulses of specific frequencies and durations to achieve desired rotations. The most important impact of using SQCs is that current SQCs cannot support an any-to-any connection between qubits when performing two-qubit operations. Instead, qubits can only interact with other qubits that are adjacent to them in the physical mesh. A sample qubit mesh is shown in Figure 4.2B. The edges in the graph denote which pairs of qubits can be used together in two-qubit gates. This becomes one of the major steps of quantum circuit compilation, discussed in section the following section.

4.5. Compilation

In order to execute a logical circuit on physical quantum hardware, the circuit must undergo a compilation process similar to that of compiling a classical program into an

executable. This involves two major steps - qubit allocation and qubit routing - along with a variety of other optimizations, such as merging or decomposing gates. Only qubit allocation and routing are relevant to our discussion, and both will be discussed in detail below.

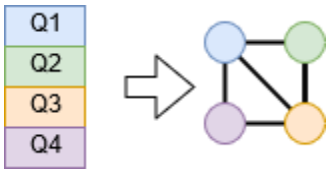


Figure 4.3. Four qubit initial layout.

Qubit allocation is the process of assigning each logical qubit used within a logical circuit to a physical qubit within the quantum computer, as seen in Figure 4.3. This is similar to register mapping or memory allocation in classical computers, where each variable must eventually map back to some physical storage. It is obvious that the physical machine must have at least as many physical qubits available as are used by the logical circuit. In cases where the logical circuit uses fewer qubits than what are available, the additional qubits are considered ancilla qubits that are largely unused.

Because of the importance of topology and positioning within the SQC qubit mesh, there are a variety of methods used for qubit allocation. The most trivial is simply to map logical qubits to physical qubits one-to-one in order. A more advanced method is to identify dense layouts with the greatest connections to reduce the number of SWAPS necessary during routing. Another alternative is to focus on physical qubits that have lower error rates to increase the success rate of the circuit. There are also compound methods that incorporate both of these aspects to identify the most promising layout.

Once we have an initial layout provided by the qubit allocation stage, we then need to satisfy the adjacency requirements dictated by the qubit mesh of the SQC. Since qubits cannot

interact in an any-to-any fashion, they must be moved to be adjacent to one another within the mesh to perform two-qubit operations as shown in Figure 4.4. The SWAP gate, a combination of three CNOT gates, performs a pairwise swap between the two involved qubits. This leads us to qubit routing, process of inserting SWAP gates during compilation to satisfy adjacency requirements. Generally, qubit routing involves not only satisfying the requirements but also minimizing the number of SWAPs to do so. Since SWAPs are CNOT gates, which generally have the longest gate times and highest error rates, excessive SWAPs can severely reduce the success rate of a circuit.

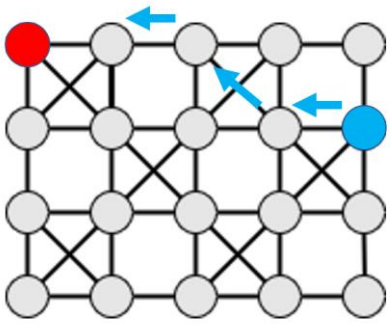


Figure 4.4. Routing example.

Qubit routing is generally more difficult to perform and more impactful on the final success rate of the circuit than qubit allocation. Similarly, there are many qubit routing methods that have been studied. The trivial method is simply to route qubits to one another in a greedy fashion to insert as few SWAPs as possible. This may have the direct result of minimizing the number of SWAPs for a particular two-qubit operation, but this can also move other uninvolved qubits further apart, potentially increasing the total number of SWAPs later in compilation. Some methods incorporate lookahead mechanisms to base SWAP decisions on the next N two-qubit operations in an attempt to avoid this addition of unnecessary SWAPs. Other methods are based on stochastic decisions, where the choice of a SWAP target is probabilistic.

Recent work [82] has demonstrated that, due to the variance of error rates throughout a SQC mesh, even routes with the same number of SWAPs could have different success rates as the individual CNOT connections have different success rates. This has led to an increase in noise-aware compilation methods that make decisions based on the current error environment. This can be as simple as simply weighing the error rates when faced with a choice of SWAP targets. Other methods, however, have combined the reversibility property of quantum circuits to optimize a circuit forwards and backwards. This effectively allows the method to view where the circuit should end then work its way backwards to the initial state. Multiple iterations of this process can be used to improve the final success rate.

4.6. Quantum Errors

Current quantum computers are still in a very early stage of development, with only a small number of vulnerable qubits. These noisy intermediate-scale quantum (NISQ) devices are primarily intended to be a research tool, though they have already been claimed to have achieved quantum supremacy, the ability to solve problems that would take so long as to be impossible on classical machines. However, NISQ devices still bear two major weakness that make them difficult to apply to many problems. First, NISQ devices commonly have on the order of ~10-100 qubits, while many interesting problems like Shor's algorithm may require many thousands. Second, NISQ devices are extremely vulnerable to errors. While quantum error correction methods exist, these methods require many redundant qubits to protect a single qubit, similar to classical redundancy methods. The limitation on the number of qubits available in a system directly impacts the availability of error correction methods that can be used for the computer.

Qubits experience a variety of errors that can generally be classified into two major categories - operational (or gate-based) errors and decoherence errors. Operational errors occur

when a qubit is involved with some single- or multi-qubit gate. Recent studies have also shown the existence of quantum crosstalk errors, where qubits adjacent to - but not directly involved in - operations can also be affected with errors. Decoherence errors by comparison are simply a result of the decay of the state of a qubit. This decay can either involve the decay of a qubit from a high-energy state ($|0\rangle$) into a ground state ($|1\rangle$) or a superpositioned state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ back to a basis state ($|0\rangle$ or $|1\rangle$). These errors occur over time following exponential distributions with time coefficients T_1 and T_2 . As such, these errors are commonly referred to as T1 and T2 errors.

The form of quantum errors can be considerably more complex than classical errors. In classical systems, errors are fundamentally bitflips, where a 1 becomes a 0 or vice versa. Qubits contain both phase and magnitude components within their state, and both can be affected by an error independently or simultaneously. Additionally, as there are infinitely many states a qubit can be in, there are also infinitely many states a qubit can change to as a result of an error. Similar to operations on a qubit, we can represent an error as a combination of rotations around each axis. This allows us to restrict this range of possible errors down to linear combinations of X, Y, and Z rotations. Error correction methods rely on this representation of errors to handle each type of error that may occur and correct them in turn.

It is important to note that the error rates of a given quantum computer are not static. In fact, these error rates are constantly changing due to changes in the environment. Qubits are extremely sensitive to external interaction, and by nature the superconducting systems are sensitive to the environmental temperature. These conditions may change by the minute, though many systems are only calibrated a few times per day to capture the current error information. These calibrations can provide important information for compilers, but the temporal difference between the calibration time and the compilation time can reduce the compiled circuit's accuracy.

5. Robust Cache-Aware Quantum Processor Layout

5.1. Introduction

Interest in quantum computing and information technology has grown considerably in recent decades. Various companies including IBM, Google, and Microsoft, along with governments around the world, have been working to advance quantum technology. Currently, these quantum chips largely act as specialized hardware for efficiently executing quantum algorithms and physical simulations, while a general quantum computer is far in the distance. However, this does not diminish the importance of quantum technologies which already see use in quantum random number generators and magnetic imaging devices. Google and NASA recently claimed highly anticipated quantum supremacy [4], the realization of a chip that can do in minutes what would take classical computers thousands of years.

The state-of-the-art quantum processors are classified as noisy intermediate-scale quantum (NISQ) machines due to their relatively small number of error-prone qubits. While these NISQ devices are beneficial research tools, their practical applications are severely limited by their scale and unreliability. Traditional error correction methods used in classical computers via replication cannot apply to qubits due to the quantum no-cloning theorem, which does not allow for copying unknown qubit states. Quantum error correction schemes do exist, but carry an overhead that is not practical on NISQ machines. This directly connects both the scale and reliability problems – at an arbitrarily large scale, we can protect qubits to ensure reliable

computation. However, just as Moore's Law is reaching its end, one can assume that an infinitely large quantum computer as predicted by the Dowling-Neven Law [26] is equally unrealistic. As such, there will always be a restriction on the number of high-fidelity qubits we can achieve in a quantum computer, and we would like to waste as few as possible on error correction.

We investigate the application of quantum caches to modern superconducting quantum computers in order to achieve functional error correction for robust quantum computing at increasingly smaller scales. Unlike classical caches that reduce execution time by reducing memory latency, these caches reduce the error correction overhead for protecting cache qubits by acting as a dedicated memory. By reducing the number of operations that take use these cache qubits, error probabilities are decreased and performance requirements are lowered, thus allowing for lighter error correction schemes. This results in more usable qubits as fewer are allocated for error correction, which in turn allows smaller scale devices to be robust to error while continuing to meet qubit and performance requirements. However, by restricting operational regions, we incur a cost as operations must avoid these cache regions. Through our experimentation we aim to minimize this overhead while maintaining the benefits of the caches.

To simulate these caches, we modify multiple parts of IBM's Qiskit quantum simulator [24]. Specifically, we target the virtual to physical qubit layout, the physical qubit coupling map, and the swap passes during compilation. This allows us to generate different cache layouts within the physical qubit map and ensure that the simulator can work with these caches to complete execution. For testing, we examine four cache layouts and two swap algorithms on five quantum algorithms and measure the overhead incurred. From our observations we provide a policy that performs well for all algorithms.

To ensure the validity of our results on large-scale systems that will implement error correction methods, we extend our simulations to larger mesh networks. Qiskit is unable to simulate large meshes due to memory constraints during the computation of the quantum algorithm. As we are focused on minimizing the movement of qubits during an arbitrary program, we can simulate large scale algorithms by removing the computation component. This enables us to bypass memory limitations at the cost of algorithm execution. Therefore, combined with our small-scale observations, we can display both correct execution and scalability.

The contributions of this work can be listed as follows:

- Our design is the first work, to the best of our knowledge, to apply memory architecture to superconducting quantum technology.
- Our design explores the design space of possible quantum computer cache layout using five advanced quantum algorithms.
- Our design achieves a possible maximum performance increase at 2.15 times compare to the worst cases while keeping a robust system.
- Our design is the first work to explore cache architecture design space at large-scale quantum chip level using mixed scale-out algorithm.

5.2. Motivation

In the previously mentioned quantum caches, the authors implement 8:1 and 1:2 encodings in their compute and cache regions, providing a large improvement in the number of qubits necessary for error correction. The main source of this improvement is taking advantage of ion trap quantum computers' long coherence times, which can be on the order of multiple seconds and possibly even minutes in certain configurations [99]. As the previous proposed cache is designed for ion trap systems, they do not face the mesh connectivity problems that

superconducting systems experience. This becomes the major design question we must address when porting this concept to superconducting technology.

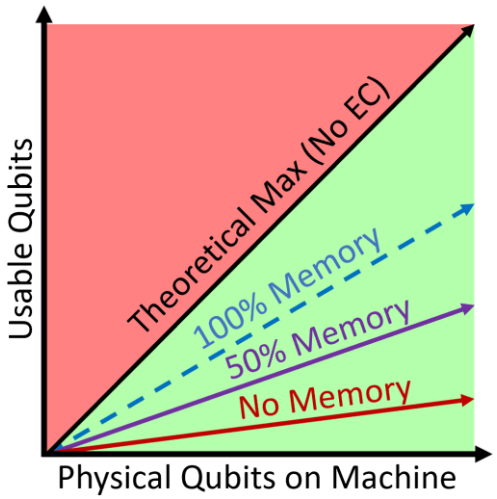


Figure 5.1. Scaling of available qubits with and without cache.

Superconducting technology cannot take advantage of coherence times when considering multiple error correction codes. Current superconducting coherence times are typically on the order of milliseconds. However, we can still take advantage of varying levels of encoding to save qubits where possible. Although we do not have the large coherence times, superconducting gate times, the time necessary to perform a gate operation on a qubit, are substantially shorter (order of nanoseconds) than ion trap machines (order of microseconds). This difference in gate time makes up for the difference in coherence times and results in comparable number of gate operations per coherence time. This allows us to follow a similar multi-level encoding structure. By having both "compute" and "cache" regions, we can also deploy faster but more costly error correction in compute regions, and a more qubit-efficient encoding in the cache region. Two simple codes to choose to employ are the Shor code for compute regions (1:8) and the Steane code for cache regions (1:6). In total this is a reduction of 2 qubits per data qubit, or a $\frac{9-7}{9} = 22\%$ reduction for the cache regions.

Figure 5.1 shows the concept of saving qubits as we increase cache size. Here, usable qubits are defined as the sum of both computation and cache qubits (basically all non-ancilla qubits reserved for error correction). Naturally, the line y=x acts as our absolute boundary. If we did not need to implement error correction at all, our device would lie along this line, though such a perfect machine does not currently exist. All other lines can be drawn following $N = \frac{pn}{c_1} - \frac{(1-p)n}{c_2}$ where *N* is the number of usable qubits, *n* is the number of physical qubits in the machine, *p* is the cache percentage, and c_1 and c_2 are the number of qubits necessary for the error correction codes for the cache and compute regions respectively. The dashed line represents marking every qubit as a cache qubit -- this is impractical as we do not have any qubits for computation, but it provides an upper bound on our design as it would apply the less costly error correction code to the entire system. Similarly, the red line denotes having no memory, and thus having the costliest error correction code apply to all qubits. This results in the most qubits allocated for error correction. Therefore, when choosing a cache size, we fall somewhere between these two lines -- shown here is an even split between cache and computation qubits. We can guide our choice of cache size by the number of qubits required for an algorithm. By maximizing the cache percentage *p* while maintaining enough usable qubits to execute the desired algorithms, we waste fewer qubits on error correction while meeting functional requirements.

An alternative perspective is to consider that we are enabling error correction on a system that cannot otherwise support error correction codes while meeting the performance and qubit requirements for a given algorithm. For example, if we wish to use Shor's code for an algorithm that requires *n* logical qubits, we would effectively need *9n* physical qubits when adding the ancilla qubits for error correction. By comparison, using a 50% cache size with the Shor and

Steane codes as discussed above, we can implement error correction with only $\frac{9-7}{9} = 8n$ qubits. While we could simply apply the Steane code to every qubit, there may be limitations that prevent this, such as performance requirements or differences in the reliability of individual qubits.

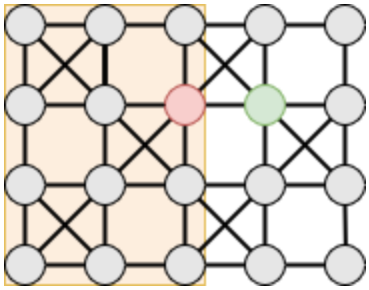


Figure 5.2. Cache-forced swaps.

5.3. Design

In order to simulate quantum caches on superconducting chips, we modify each of the three main parts of the Qiskit library – the coupling mesh, the initial layout, and the swap algorithms. We add a list to the mesh to keep track of the included cache nodes. This provides a base from which to enforce gate restrictions and make the execution cache-aware. The initial layout is modified to prioritize non-cache qubits to avoid unnecessary swaps. The swap algorithms must be aware of the cache qubits to enforce swaps in and out of the cache in addition to its traditional job of ensuring that qubits are adjacent for entanglement.

Beyond simulating on small scale superconducting chips around 20 qubits, we also extend Qiskit to perform large-scale quantum circuits on large-scale superconducting chips, around 100 qubits level. To perform this task, we modify the Qiskit library to allow large-scale circuit compilation and propose a scale-out algorithm to generate these circuits. We modify the simulation process to only perform swap mapping while discarding data operations such as single-qubit gates. This allows us to circumvent the library restrictions and makes simulation

feasible. For the scale-out algorithm, we extract circuit characteristics directly from real quantum applications and feed them into the mixed scale-out algorithm, which aims to run on classical computers with high fidelity relative to the real large-scale circuit.

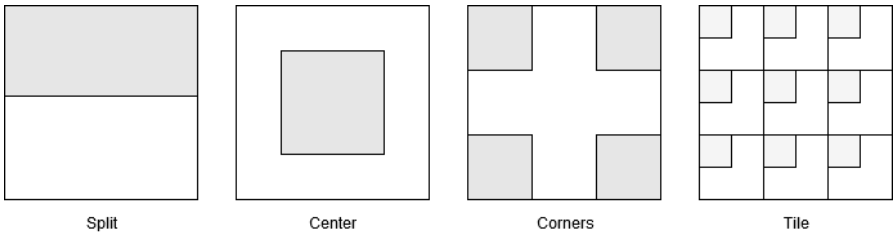


Figure 5.3. Four basic cache topologies.

5.3.1. Coupling Mesh and Cache Topology

The first necessary step is to add the concept of a cache to the simulator. We add a list to hold all nodes that are in the cache. This is used by the layout and swap algorithms to identify which nodes should be prioritized for mapping and swapping. We additionally construct a separate mesh with cache nodes removed to allow for easy swap path identification. The swap algorithms rely heavily on shortest path algorithms, so it is beneficial to have a pre-made separate graph limited to only data qubits to prioritize non-cache swapping where beneficial. Lastly, we add functions to generate the various cache topologies used in our experiments.

In the previous work examining the quantum memory hierarchy, the system is implemented using ion trap technology [90]. Due to its arbitrary qubit entanglement capabilities, there is little distinction between data and cache qubits. However, there is a major difference when using superconducting qubits because the connectivity between qubits is limited, and we cannot simply operate on any two qubits at will. Choosing which qubits to place in the cache thus has an impact on the swaps the algorithm must perform to complete the algorithm. As shown in Figure 5.2, the two marked qubits are adjacent and therefore should be available for

operation. However, with the highlighted cache placement, one of the qubits falls within the cache and thus is not valid for use. To complete the operation and continue with algorithm execution, the qubit must be moved out of the cache, creating more swaps.

Given *n* physical qubits there are 2^n potential cache layouts to consider. In order to avoid searching this whole space, we instead focus on four different topologies, two contiguous and two distributed, as shown in Figure 5.3. The two contiguous methods, a straight split and a central cache, allow for larger contiguous cache and computation areas. The two distributed methods are the four corners of the mesh and a tiled version that spreads the cache equally throughout the mesh. The corner topology also allows for a large contiguous region of computation but divides the cache into parts, while the tile method instead opts to intersperse both cache and compute qubits. These four methods together provide a variety of options to consider for cache design based on algorithm characteristics.

5.3.2. Layout

The initial layout of the algorithm maps the required virtual qubits to the physical qubits in the mesh, as shown in Figure 4.3. Similar to the previous discussion about cache topology, a number of compiler passes directly involving the physical qubits need to be modified to account for the cache, including this initial layout. Poor initial qubit placement can result in additional swaps. It is not as critical as the swap algorithm since it is only the initial state of the system, but it can have considerable impact on shorter algorithms.

Typical layout passes examine both the mesh and the algorithm to decide on qubit placement. First, identify the most and least heavily connected qubits in the mesh. Second, look ahead through the gates that must be executed to identify the order in which qubits must be operated on. Using this information one can map the qubits that are used together to be nearby in

the mesh. However, adding the concept of a cache changes this process, as an unaware layout may map virtual qubits to cache qubits and add unnecessary swaps. We modify this process to also reference the list of cache qubits in the mesh and prioritize non-cache qubits before cache qubits. This allows us to retain the benefits of the pre-existing layouts while avoiding as many unnecessary swaps due to the cache as possible.

5.3.3. Swap Algorithms

The last major part of the compilation process that we must modify are the swap algorithms. These algorithms are responsible for ensuring that qubits are adjacent to each other whenever two qubits are operated on together. As discussed previously, it is critical for the swap algorithm to be cache-aware for the algorithm to complete execution. Otherwise, the simulation would attempt to operate on qubits when it cannot due to the cache, leading to failures. In this sense it is also the responsibility of the swap algorithm to enforce these additional restrictions on the qubits' positioning within the mesh. While there are many different swap algorithms that have been studied to minimize swaps or maximize reliability, we present two different cache-aware swap algorithms and compare their properties and behaviors.

In general, the swap algorithms examine the list of operations that must be executed and insert swap operations that move the qubits to their necessary positions in the mesh. With no cache, single qubit operations do not require any swaps as they can be executed locally at any qubit position. The only operations that force qubit movement are two qubit operations such as the CNOT. Three or more qubit operations do exist, but can be unrolled into a combination of one and two qubit operations in the basis set. Upon locating a two-qubit operation, the swap algorithm checks whether they are adjacent. If not, it finds a shortest path from one qubit to the other and inserts swaps along that path. It is possible for the movement of one qubit to possibly

move other qubits further from their goal locations. Some algorithms implement look-ahead mechanisms to address this problem and increase system-wide efficiency, while others implement probabilistic methods to avoid interference.

Algorithm 5.1. Baseline Swap Algorithm

Input: A list L of all gates and their operating qubits q, v

for all gates in L **do**
  if single qubit gate and q in cache **then**
    swap along shortest path to nearest available non-cache qubit
  end if
  if two qubit gate **then**
    if both q and v in cache **then**
      swap q out to nearest available non-cache qubit
      swap v to nearest available non-cache neighbor of q
    else if only one of q and v in cache **then**
      swap along shortest path to non-cache neighbor of v or q respectively
    else if neither q or v in cache and they are not adjacent **then**
      swap v to nearest available non-cache neighbor of q
    end if
  end if
end for

When incorporating a cache, the first modification involves single-qubit gates as they can no longer be executed on any qubit in the mesh. Instead, they are now capable of forcing movement if a single-qubit operation is set to take place on a cache qubit. By definition of the cache, operations should not act on cache qubits wherever possible to increase system reliability. As such, even single-qubit gates may require moving out of the cache to a non-cache region. For two qubit operations, as previously discussed, both qubits must be out of the cache and adjacent with each other for the operations to be successful. Both algorithms presented follow Algorithm 3.1, but act differently when selecting the paths to take to move qubits together. The first algorithm acts as a baseline, here referred to as the BaselineSwap (BSwap). It finds the direct shortest path between two qubits and routes them together, ensuring their final positions are not within cache qubits. This ensures correct execution of the algorithm, but allows for swaps through the cache. The second swap algorithm aims to minimize the number of swaps occurring

within a cache, here referred to as the NoCache Swap (NCSwap). By utilizing the previously mentioned reduced mesh that does not contain cache qubits, the algorithm can easily find the direct shortest path using only non-cache qubits. In the case that either qubit is in the cache itself, it first moves them out to a non-cache region, then routes them together avoiding cache qubits. The only modification necessary is to use this modified mesh that does not contain the cache qubits when finding the shortest paths in lines 3, 8, 10 and 12. This algorithm does not work if the compute region is not contiguous, as the mesh then becomes disconnected, though it can be modified to simply fall back to BSwap in these circumstances.

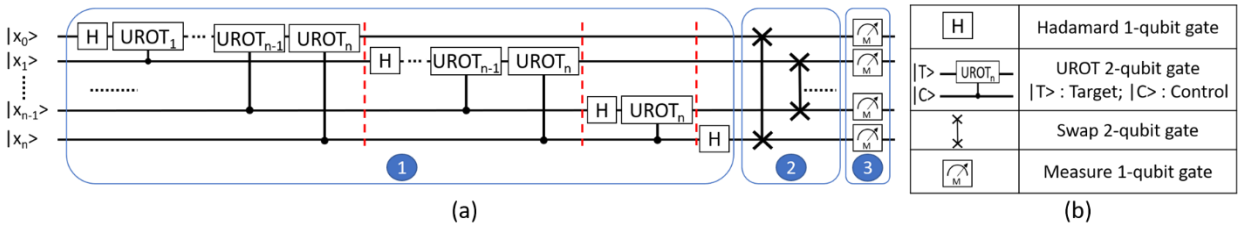


Figure 5.4. n-qubit QFT circuit. (a) Separated by stage. (b) Table of gates with number of qubits involved.

Implementing the cache adds data movement as we must swap qubits in and out of the cache and avoid transferring through cache regions where possible. The cache shape, size and swap algorithms can affect this performance overhead. Our design aims to minimize the number of swaps at various cache shapes and sizes, which we treat as our main performance metric in the following evaluations. Reducing the number of added swaps reduces execution time and increases reliability by reducing the number of total gate operations on the qubits, in addition to enabling the error correction codes at smaller scales.

5.3.4. Large-Scale Implementation

Due to the inherent exponential growth of quantum algorithms, using classical computers to simulate complete quantum circuit generation and computation at large scale is infeasible and

would otherwise contradict quantum supremacy. However, observing the behavior of a real application on a large-scale circuit is one of the critical components for assisting large-scale quantum computer design. Therefore, it is necessary to find a feasible solution which generates a large-scale quantum circuit based on the small-scale algorithm using a classical computer with high fidelity compared to the real large-scale circuit.

By inspecting the growth rates of gate counts within real applications on the small-scale Deutsch-Jozsa algorithm, shown as Figure 5.5, we observe the number of total gates, CX gates, and CX gates per qubit are growing at consistent exponential rates. After studying this trend, we have found an inherent growth behavior of a quantum algorithm that within one quantum algorithm, different stages follow a strict sequence order, and some stages grow at an exponential rate while others only grow linearly. A simple Quantum Fourier Transform (QFT) circuit, shown as Figure 5.4, will be used to illustrate this behavior, with circuit implementation from [79]. As Figure 5.4(a) shows, the QFT algorithm can be separated into sequences of stages based on its functionality, which is true for many algorithms. Stage 1 boxed in blue performs a sequence of one Hadamard gate followed by a series of UROT gates applied to all higher indexed qubits on every qubit. Similar to the CX gate, the UROT gate is a two-qubit controlled rotation gate that requires target and control qubits to be adjacent in the mesh. Stage two and three are swap and measurement stages, which request three CX gates and one measurement gate for each operation respectively. Based on the behavior of each stage shown as the table of Figure 5.4(b), it is clear that stage 1 has an exponential scaling n^2, but stage 2 and stage 3 will have a linear scaling with $\frac{n}{2}$ and n respectively.

Therefore, as shown in Figure 5.4, a large-scale implementation of QFT would follow the same sequence of execution stages as small-scale while using more qubits. The number of gates

necessary for some stages may scale linearly with the number of qubits, such as stage 2, while others may scale exponentially. This general trend is proved in Figure 5.5. This exponential growth quickly becomes impossible to compile and simulate on a classical computer, preventing direct scaling. However, it is still possible to approach similar qubit movement behavior using the Mixed Scale-out Algorithm presented next.

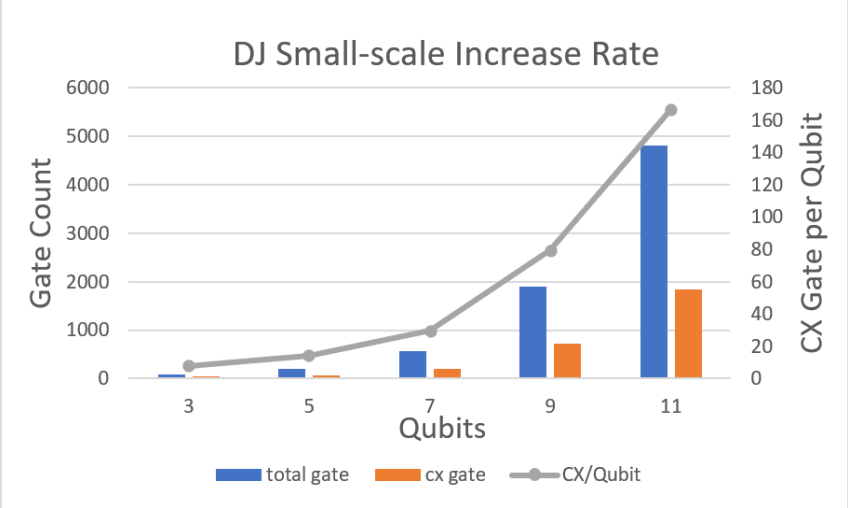


Figure 5.5. DJ gate count scaling.

A mixed scale-out algorithm will enlarge each stage of a real small-scale algorithm with a constant rate of n copies of gates from the target stage using a random mixed fashion. Following this rule, the circuit produced by a mixed scale-out algorithm will have the same stage sequence as the real large-scale quantum algorithm. For the gate number difference, the resulting circuit will only be different at stages that require exponential scaling, which are projected to a constant rate *n*. Since this paper focuses only on the swap mapping and swap count difference between different cache layouts, only 2-qubit gates will be extracted and used, as single-qubit gates do not affect the swap counts. Using the QFT circuit as an example, assume that the small-scale circuit has 6 qubits and that the target large-scale circuit has 120 qubits. A mixed scale-out algorithm will make a copy of the small-scale algorithm 20 times, and aggregate all the gates into

corresponding stages. For stage 2 and 3, this enlarging process will be very close to the real large application. Stage 1 will have $6^2 * 20 = 1{,}240$ swap paths generated by the mixed scale-out algorithm, while the real large-scale circuit will have $120^2 = 14{,}400$ swap paths. The mixed scale-out algorithm successfully reduces the complexity of the swap path generation from $O(n^2)$ to $O(n)$ with all the created swap paths belonging to the real large-scale circuit. Therefore, the mixed scale-out algorithm perfectly suits the purpose of exploring the path behavior in the large-scale quantum chip with a balance between feasibility and fidelity.

Algorithm 5.2. Mixed Scale-out Algorithm

Input: A list of Quantum Gates C, scale-out ration n
Output: A list of Quantum Gates C'
  for each gate $0 \leq g \leq |C|$ **do**
    if $gate != measurement$ && $gate! = barrier$ **then**
      add the gate in C'
    end if
  end for
  for each gate $0 \leq g \leq |C|$ **do**
    Duplicate each gate n times
    Rename every Quantum Register of each new gates
    Save the new gates into C'
  end for
  enable random qubit allocation

The large-scale additions for the Qiskit library inherits features of the three implementations in the preceding sections. Compared with the original Qiskit library, the large-scale Qiskit library only performs the circuit construction and compiling processes while discarding the simulation part for computation. Meanwhile, the limitation on the number of physical qubits for simulation (in place due to memory constraints) has been removed to allow mapping of the virtual qubits to physical qubits with any given size. Since we remove the computation, there are no memory concerns. This isolated process releases the potential of the library to be able to measure the swap count with any given input size.

The proposed mixed scale-out algorithm, shown in Algorithm 5.2, aims to generate circuits on a large scale and keep the balance between feasibility and fidelity. The algorithm

extracts the circuit from a small algorithm and feeds it into the filters that filter out the barriers and measurement operations that influence combination with another circuit. The duplication procedure will rename the quantum registers to avoid mismatching. For executing each gate within the target stage from different small circuits, the algorithm adopts the round-robin policy to execute each gate of the circuits. After the generation of the circuit, the algorithm will enable the random qubits allocation procedure to avoid the qubits from the same copy to aggregate.

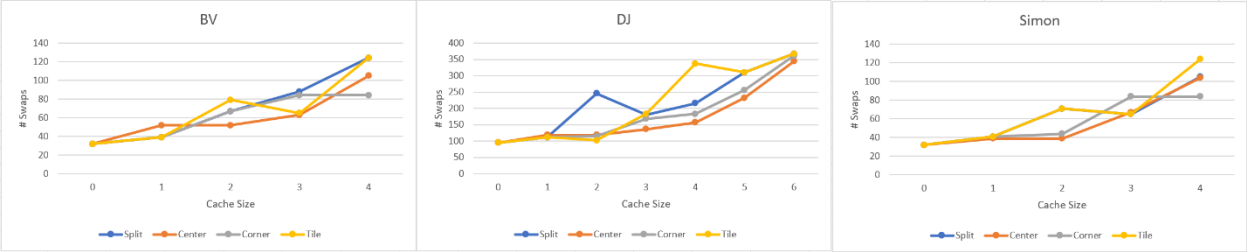


Figure 5.6. Baseline Swap results, small algorithms.

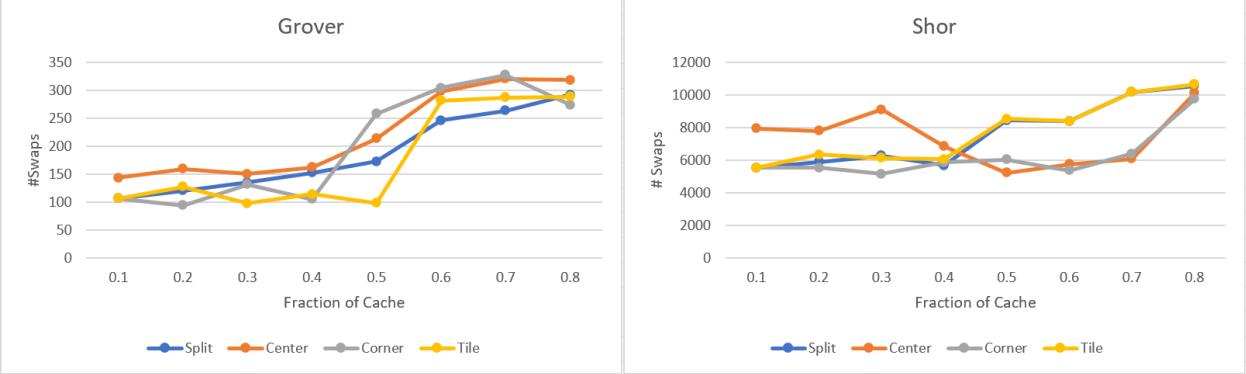


Figure 5.7. Baseline Swap results, larger algorithms.

5.4. Results

For our experimentation, we test five quantum algorithms: Shor's factorization algorithm [86], Grover's search algorithm [42], Simon's algorithm [88], the Deutsch-Josza (DJ) algorithm [25], and the Bernstein-Vazirani (BV) algorithm [11]. These algorithms cover a variety of quantum computation tasks. BV, DJ and Simon's algorithm are three of the first quantum

algorithms used to demonstrate the benefit of quantum computers over classical computers. Grover's search algorithm searches a set of quantum data to identify matching queries. Shor's algorithm, likely the most well-known of the set, is a polynomial time factoring algorithm whose security concerns interest governments worldwide. Each of these algorithms are tested on all four cache topologies at various cache sizes. The number of swaps is recorded to show the swaps incurred by each cache topology. The overhead in the number of swaps can be used to approximate the overhead in computation time following the function $f(n_s) = 3 \times n_s \times t_{cnot}$, where n_s is the swap overhead and t_{cnot} is the average time necessary to execute a single CNOT gate. Results for both the BSwap and NCSwap algorithms are shown to provide comparisons under all settings.

5.4.1. Small Scale

For these small-scale simulations, we implement the cache and swap algorithms on sizes that can be executed with current technology without error correction enabled. We cannot actually implement error correction for testing purposes, as it would require more qubits than can be feasibly simulated. Instead, we are working directly with the qubits and assuming that error correction would be implemented at a larger scale.

The results for the BSwap algorithm are split into two figures, Figure 5.6 and Figure 5.7. The larger algorithms (Shor and Grover) are separated from the smaller algorithms (BV, DJ, Simon) because we use two separate methods for manipulating the cache size. For the larger algorithms, taking a percentage of the total mesh size as cache works with no problem, as there are enough qubits to select from. However, the smaller algorithms use a smaller mesh, resulting in rounding issues when using cache percentages. To provide more clear results about the effects

of increasing cache size, we instead directly increase the number of cache qubits rather than relying on a percentage.

The smaller algorithms exhibit relatively similar behavior for all of the cache topologies. At most cache sizes, either the corner or center topology performs best depending on the algorithm, though the tile and split topologies do not perform much worse. As expected, the number of swaps trends upwards as cache size increases, as more qubits must be moved out of the cache.

The larger algorithms also show a general trend upwards in swaps as cache size increases, though Shor's algorithm shows somewhat more complex behavior. Grover's algorithm stands out as the only one that benefits mostly from the tile topology, and the split topology at higher cache sizes. By comparison, Shor's algorithm has consistently low number of swaps with the central topology at all cache sizes except for the largest cache size. We will discuss why we believe the algorithms display these behaviors in the following discussion section.

In order to underscore the importance of choosing the correct cache topology, we identify the best and worst cache topology at each cache size for each benchmark and calculate the overhead difference between the best and worst topologies. Based on the previous observations in Figure 5.6 and Figure 5.7, we choose to treat the central cache as our default choice as it consistently performs best on the smaller algorithms, and the larger cache sizes for Shor's algorithm. Figure 5.8 displays the minimum, maximum, and mean overhead figures for each benchmark. As shown, the difference between minimum and maximum ratios can be very large, with the greatest difference being roughly 147% for Grover's algorithm. When examining only the default central topology, we similarly find the greatest difference of 115%, or a 2.15x improvement. There is also typically a sizable difference between the average and minimum

overhead, ranging from roughly 9-30% overhead over the best choice of topology. The large differences between the maximum and mean topologies indicate the impact of cache topology on performance. If one were to blindly choose a cache topology for a given algorithm and cache size, they could suffer these large increases in the number of swaps and therefore execution time. As such, our recommendation is to profile a given test program to identify which cache topology is optimal, but the central cache is a consistent choice for all of our tested algorithms.

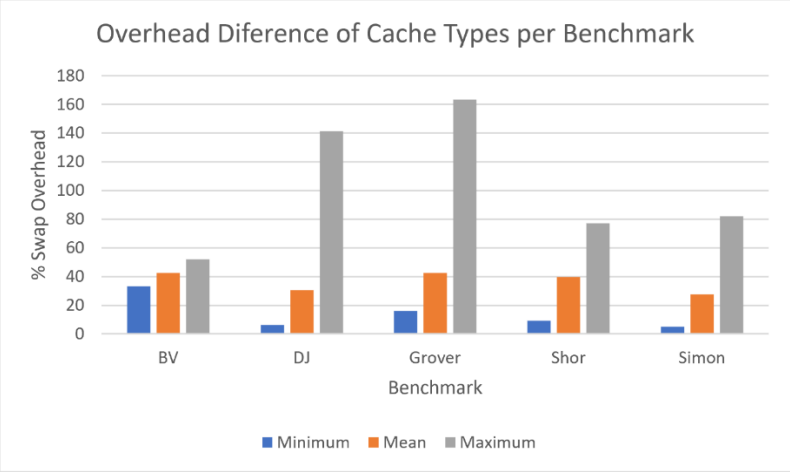


Figure 5.8. Best, mean and worst-case performance between topologies of each benchmark at all cache sizes.

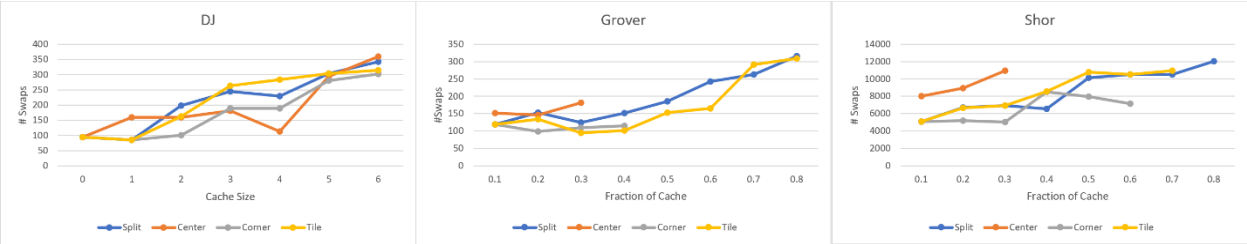


Figure 5.9. NoCache Swap results.

In addition to the previous results for the baseline swap algorithm, we also include results for the second NCSswap algorithm. This algorithm was created to minimize the number of cache-influenced swaps (a swap where one or both of the qubits were in the cache) during execution. The swap results are shown in Figure 5.9. For brevity, only DJ is shown from the

smaller class of algorithms, along with Shor's and Grover's algorithm. We chose these three algorithms as our sample because DJ, BV and Simon's algorithm show fairly similar behavior and can be represented by one representative.

As seen with BSwap, for DJ the center and corner topologies typically perform better. However, Shor and Grover show very different behavior. The most prevalent observation is that not all cache topologies can be run at higher cache sizes, namely center and corner, which had performed best with the baseline swap algorithm. The cause of this is the requirement that the algorithm must not swap through the cache, except when one or both qubits originate in the cache. If distinct compute regions are isolated from one another by a cache region, qubits cannot move across the boundary and the computation cannot finish. This explains why the center topology is the first to fail, as it grows to divide the mesh in half. It is possible in this case to fall back on the baseline swap algorithm, but it is interesting to observe at which point the topologies begin to fail. Only the split topology is capable of completing both algorithms, which is sensible as it results in a large contiguous compute region. The tile configuration also performs well, and for the Grover algorithm also has the minimum number of swaps.

To provide more insight into the impact of the no-cache swap algorithm, we measure how many swaps occur where at least one of the two qubits are present in the cache. We then calculate the difference between the percentage of these swaps for the baseline and no-cache swap algorithms. We expect that no-cache swap will display a substantial reduction in the frequency of these cache involved swaps as we actively avoid swapping through the cache where possible. We again calculate the geometric mean over the various cache sizes and topologies to present the total effect of the no-cache swap algorithm regardless of chosen topology or cache

size, though it is worth noting that both algorithms have similar performance at the largest cache sizes as it becomes impossible to avoid swapping through the cache.

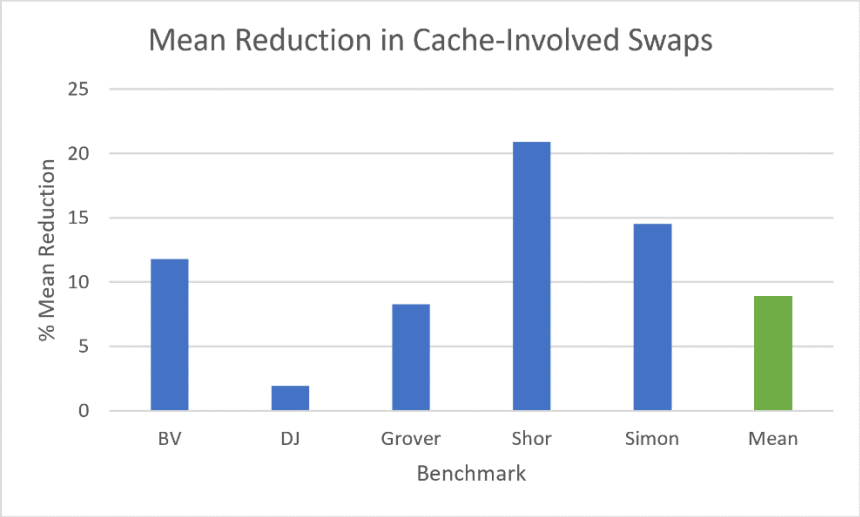


Figure 5.10. Percent reduction of cache-involved swaps.

The results of these measurements for each benchmark are shown in Figure 5.10. As expected, the no-cache swap algorithm reduces the number of cache-involved swaps in every benchmark regardless of cache size or topology. Shor's algorithm shows the greatest improvement at nearly 21%, with a minimal improvement of roughly 2% on DJ. Across all five benchmarks, the no-cache swap algorithm provides a mean 9.85% reduction in the number of cache-involved swaps. Note that this reduction does come at an overhead in total number of swaps, as can be seen by comparing Figure 5.6, Figure 5.7 and Figure 5.9. However, reducing the number of swap operations involving the cache further reduces the number of error correction operations that would be necessary in the cache.

5.4.2. Large Scale

As Figure 5.11 showed, the results come from feeding DJ benchmark into the large-scale Qiskit with baseline swap using 96 physical qubits. After comparing with its smaller version, the overall behavior of the different topologies is similar. The rankings of the layouts at large-scale

are stable that centers perform best, and split as the worst. One of the reasons that split takes the most swaps to execute might be that split has the longest path of the max possible distance between one pair of cache and non-cache qubits, which becomes even more pronounced as the mesh increases in size. This demonstrates that the policy will be applicable for scale-out large quantum algorithm. As shown in the figures, for different layouts at a different size, a wise policy can reduce the number of swap operations by one order of magnitude for switching between the optimal layout in different cases. The large-scale results also support the assumption that with a very aggressive memory-dense design as 80%, the policy can keep the extra swap overhead within 2 times range and achieve a 3 times reduction on quantum chip size.

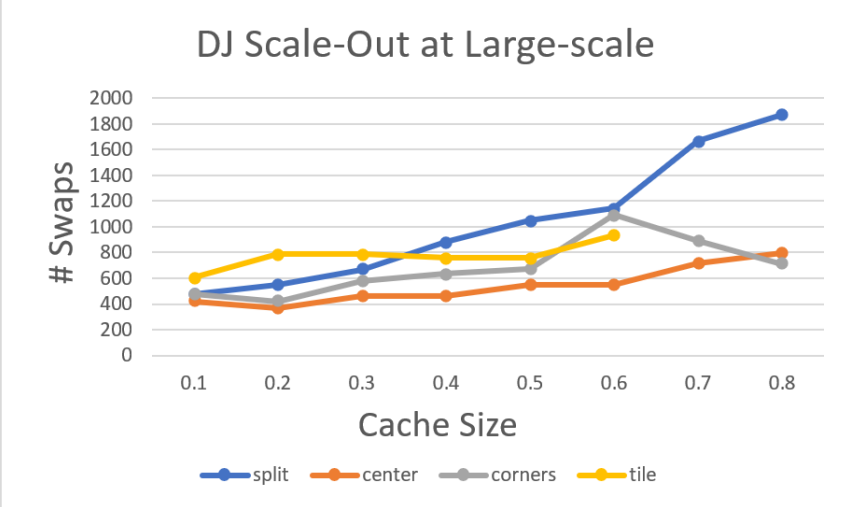


Figure 5.11. Large-scale results for DJ.

5.4.3. Discussion

In order to provide insight into the behavior of the algorithms we tested, we present here a small discussion on their properties and how they influence our observations. First, each of the smaller algorithms are relatively similar. They begin with a set of Hadamard gates to create superpositions of the qubits, perform a sequence of operations that depend on the given oracle the circuit is made to execute, then end with another set of Hadamard gates and measurements to

extract the results. The cache mostly impacts multi-qubit gates as qubits must be moved to adjacent positions. These three algorithms benefit from larger contiguous compute regions, though not extremely as they do not implement many multi-qubit gates depending on the oracle.

By comparison, both Shor's and Grover's algorithm are more complex and rely on a larger number of multi-qubit gates. Shor's algorithm in particular shows a considerable difference between cache topologies at large cache sizes likely due to the large number of multi-qubit gates. Center and corner cache topologies provide contiguous compute regions without having to move as across the entire graph as often to meet adjacency requirements. Grover's algorithm stands apart from all of the others, actually benefiting most commonly from the tile topology. This is likely due to the implementation of the input oracle, which happens to execute in a way that benefits from the tile topology.

5.5. Related Work

In a previous work, Thaker et al. discuss the possibility of multiple levels of encoding at various overheads for computation, cache and memory regions in ion-trap-based quantum computers [90]. While traditional cache designs provide performance increases by holding frequently used data in high-speed memory regions, the quantum cache design focuses on decreasing chip area when deploying error correction codes. By utilizing two error codes, they avoid wasting a large number of qubits for error correction on qubits that are less frequently used or less prone to error (the cache qubits). This allows for slower or less precise error correction methods in memory regions that are used less frequently, while deploying faster error correction in computation regions that must be applied after each operation. They also include code conversion circuits for transferring qubits in one encoding to another, allowing for fast transfer between cache and computation encodings.

5.6. Conclusion

In order to execute important quantum algorithms, it is necessary to increase the number of available qubits in a quantum computer. Quantum caches are one such method by reducing the number of ancilla qubits necessary for implementing quantum error correction codes. We have extended this concept from ion trap computers to superconducting meshes and modified the Qiskit quantum simulator to accommodate quantum caches. With these modifications, we have examined various cache sizes and topologies on five different quantum algorithms. Our observations show that central caches typically minimize the number of swaps added by the cache during algorithm execution, but it is optimal to profile each individual algorithm. We proposed an alternative cache-aware swap algorithm that reduces the cache disturbance caused by swapping qubits, further reducing cache operations and increasing reliability. In combination, these methods will increase the number of usable qubits on systems that implement error correction.

6. Gate-Based Partial Compilation of Quantum Neural Networks

6.1. Introduction

Interest in quantum technology has grown massively in the last two decades. Many technology companies and national governments are investing to develop quantum hardware and software for security and computation purposes. Quantum technology is not limited to these fields, and has also been deployed both in random number generators and medical imaging devices to improve upon pre-existing machines. However, most interest lies in quantum computation and information theory. Quantum algorithms have been shown to greatly improve execution times for certain problems, such as Shor's algorithm [86], which can nearly exponentially accelerate the factoring of large numbers. Communication systems built using quantum encryption are provably secure to outside tampering, including quantum interference. Although quantum hardware is still under heavy development, quantum computers have recently reached a state of quantum supremacy for specific problems, meaning that it takes minutes to complete tasks that would take classical computers thousands of years [4].

Current cutting-edge quantum computers are considered noisy intermediate-scale quantum (NISQ) processors, as they are made of a small number of vulnerable qubits [33, 50]. These NISQ devices serve as powerful research tools to study both quantum hardware and software. Currently, most NISQ machines use superconducting technology to implement qubits, though other methods involving charged ions or photons also exist. Most quantum algorithms make use superposition or entanglement to aid computation. They are not without limitations, however, as care must be taken to extract desired information without collapsing the vulnerable states. Additionally, current NISQ computers are inherently vulnerable to environmental and gate errors. Although quantum error correction methods exist, they cannot be applied on NISQ

machines due to their limited size. Quantum computers are also commonly not standalone machines but instead act as accelerators, where one can prepare and submit jobs from a host machine. These jobs contain circuits to execute, which are compiled on the host machine for the target physical quantum computer before submission.

A problem arises when a circuit must be recompiled frequently, as the overhead for compiling and submitting the job becomes costly. This can occur in algorithms with changing gates or those that adjust values of parameterized operations after receiving feedback from their results. Well-known examples of this form of algorithm are neural networks, which during training are adjusted every iteration to learn parameters. Quantum neural networks (QNNs) follow similar patterns to learn from input data. This can cause compilation to become a considerable portion of training time. However, not every stage of the algorithm changes every iteration, as we typically only adjust weights and not, for example, the application of threshold functions.

Drawing from these observations, we investigate a form of partial compilation to prepare circuits for NISQ machines that aims to reduce this repetitive compilation overhead by avoiding unnecessary compilation for constant regions of the circuit. By storing the compiled static sections of the circuit and compiling only the variable parts of the circuit every iteration, we aim to reduce the overall time spent on compilation to improve iterative QNN training. We modify IBM's Qiskit quantum toolkit [24] to include partial compilation for quantum circuits. We utilize their directed acyclic graph (DAG) model of circuits to enable partial compilation of circuit blocks. These blocks can then be combined to reconstruct the original circuit without recompiling the entire circuit. We then test the approach on a collection of popular neural

network architectures and measure the time saved with partial compilation. Lastly, we provide insight into more generic circuits that may not follow strictly to the neural network pattern.

The major contributions of this work can be listed as follows:

- Our approach is one of the first to apply partial compilation to quantum neural networks specifically with the Qiskit toolkit and its compiler.
- Our approach reduces compilation time by up to 77% per iteration for well-known QNN architectures.
- Our approach reduces total training time by up to 66% over 1000 iterations.
- Our work explores the impact of optimization passes on partial compilation, providing insight for future work.

6.2. Design

In order to implement partial compilation for Qiskit programs, we rely on their directed acyclic graph (DAG) model for circuits. This allows us to compile individual parts of a circuit then combine them together to create the finished circuit. We refer to regions that do not need to be recompiled as static blocks and those that must be recompiled as dynamic blocks. The following sections cover this DAG model, identifying static and dynamic blocks, and the neural network circuits we use to evaluate the approach.

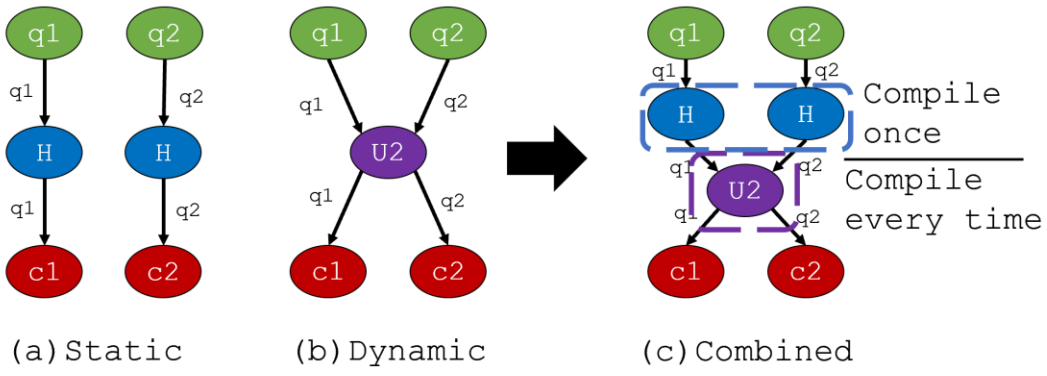


Figure 6.1. Simple DAG representation and concatenation.

6.2.1. DAG Representation

Qiskit uses both a circuit representation and DAG representation for quantum programs. Circuits contain a list of gates to execute and registers to execute them on. The DAG instead presents a data flow view of the circuit where nodes represent gates and edges represent qubits that pass from gate to gate. An example DAG is shown in Figure 6.1. The top-most nodes are our quantum registers that the circuit uses, while the bottom-most nodes are the classical registers that the quantum registers get measured into to extract data back into a classical space. Without partial compilation we would recompile the entire full circuit every time, but with partial compilation we only recompile the portions coming from dynamic blocks.

The DAG offers useful features for compilation in general. First, we can convert back and forth between circuit and DAG, allowing us to operate on the DAG when beneficial and return to an executable circuit. Second, we can perform existing compilation passes on the DAG representation, allowing us to compile DAGs individually. Third, we can concatenate DAGs together or replace individual nodes within the DAG by mapping their input and output registers together. Together this provides a toolkit that we use to implement partial compilation by allowing us to individually compile blocks, assemble them into a complete DAG, run additional

global passes if necessary, then convert the DAG back to a circuit for execution. These steps are shown in Figure 6.2.

For step **1**, we must first convert a given circuit to its DAG form. This enables us to manipulate the circuit more freely. Then **2**, we must divide the circuit into static and dynamic blocks. This step is discussed in the next subsection. Steps **3** and **4** come as a pair - if a block is static and we have a stored version, we use it. Otherwise, we recompile the block. Step **5** then requires appending the blocks together as shown previously, and **6** converts back to a circuit for execution. There is overhead involved with these steps compared to compiling the circuit together as a whole, but our results will show that the time saved exceeds this overhead substantially.

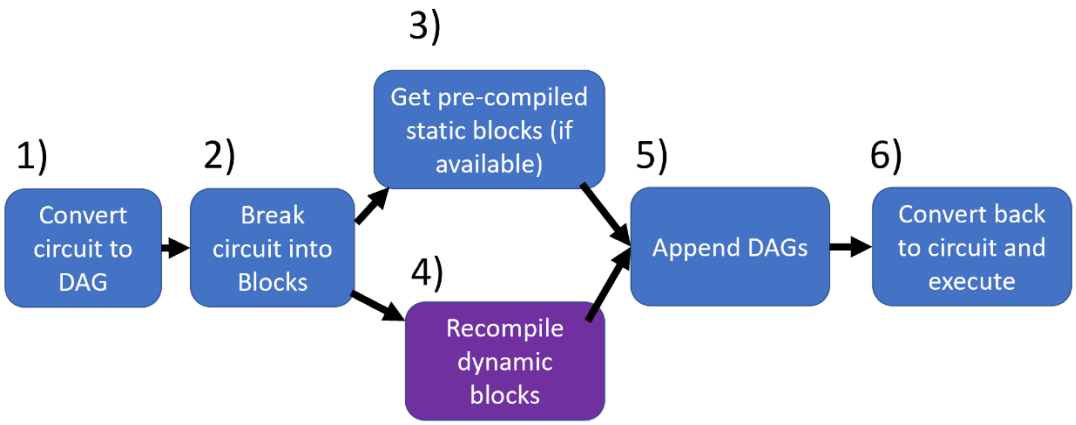


Figure 6.2. Partial compilation workflow.

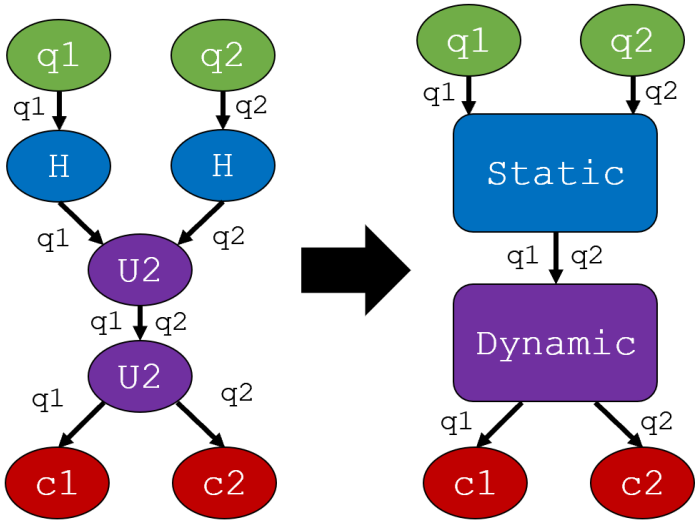


Figure 6.3. Aggregating blocks to reduce overhead.

6.2.2. Blocks

Classical neurons typically involve three phases - applying weights to inputs, summing up the results, and potentially applying a threshold function. Quantum neural networks (QNNs) typically follow a similar structure as they attempt to mimic behavior. When we consider these stages in the context of partial compilation, both the summation and threshold functions are constant across iterations while the weights change from iteration to iteration. We break up the QNNs into static and dynamic blocks similarly. To identify blocks in a circuit, we examine each gate to determine if it is constant or parameterized. We then combine consecutive gates of the same type into blocks to reduce the total number of blocks and the overhead for appending them together during compilation, as shown in Figure 6.3. For the neural networks, this typically coincides with each layer.

A second benefit for partial compilation comes from the structure of the networks themselves. The connections between neurons are normally static throughout the training of a network. In the quantum case, this means that we have a consistent order the qubits will be used.

The layout of physical qubits, and thus the order we must swap qubits during execution to ensure two-qubit operations execute correctly, does not need to change from iteration to iteration. Therefore, we do not need to rerun these sorts of compilation passes each iteration.

6.2.3. Compilation Passes

Qiskit provides a number of compilation passes along with the ability to write custom passes when necessary. These passes are grouped into pass managers that can be applied to DAGs. During our following experimentation we test with four provided pass managers that act as optimization levels during compilation. However, some discussion of individual passes and how they affect compilation is necessary to understand partial compilation with these passes. Some passes, such as unrolling or decomposing gates, can be applied locally to blocks as they have little to no impact to surrounding blocks. Others must be applied globally as they influence the entire circuit. Lastly, some have consistent effects, such as the previously discussed swap passes, that always create the same results. For our experimentation, we default to running passes locally except where it is necessary to run them globally for correct functionality. Further investigation into the effects of specific passes and partial compilation may be a useful direction for future work.

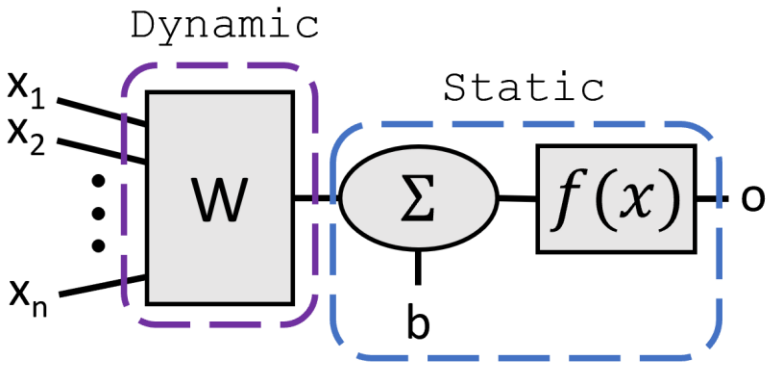
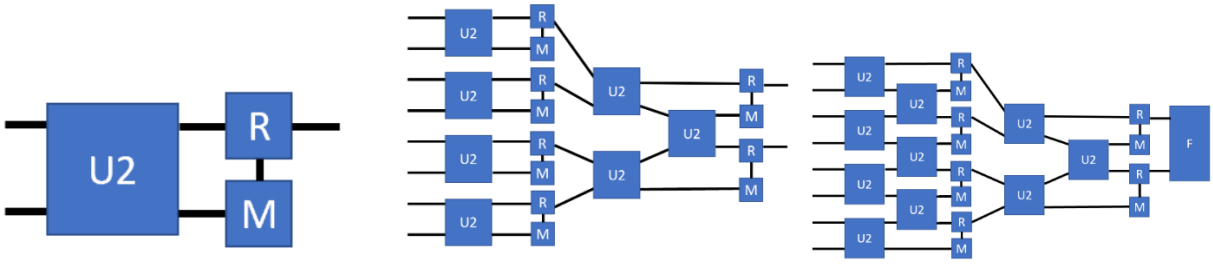


Figure 6.4. Classical neuron structure.



(a) Neuron (b) Artificial Neural Network (c) Convolutional Neural Network [5]

Figure 6.5. Example QNN architectures.

6.3. Evaluation and Results

We evaluate our approach on the previously discussed networks by measuring the compilation time overhead per iteration for each network. We compare the execution and compile times for each network to quantitatively demonstrate the importance of reducing compile time. This includes measuring the impact on training time. We also evaluate any impact on the accuracy of the circuit by computing the fidelity between normal and partial compilation. Lastly, we provide insight into the limitations of our approach by investigating the impact of PassManager optimization levels and the proportion of static and dynamic blocks within a circuit.

6.3.1. Benchmarks

For experimentation we test on quantum versions of well-known classical neural networks. Specifically, we test implementations of individual neurons, basic artificial neural networks (ANNs), including sparse and fully-connected versions, convolutional neural networks (CNNs) and recurrent neural networks (RNNs). Motivations for these circuits stem from pre-existing QNNs [23, 49]. A sample of our test networks can be seen in Figure 6.5. As in classical networks, many of the larger networks are built out of the simple single neurons. Except for the neuron, which is a small two-qubit circuit, all larger networks operate with 16 qubits by default.

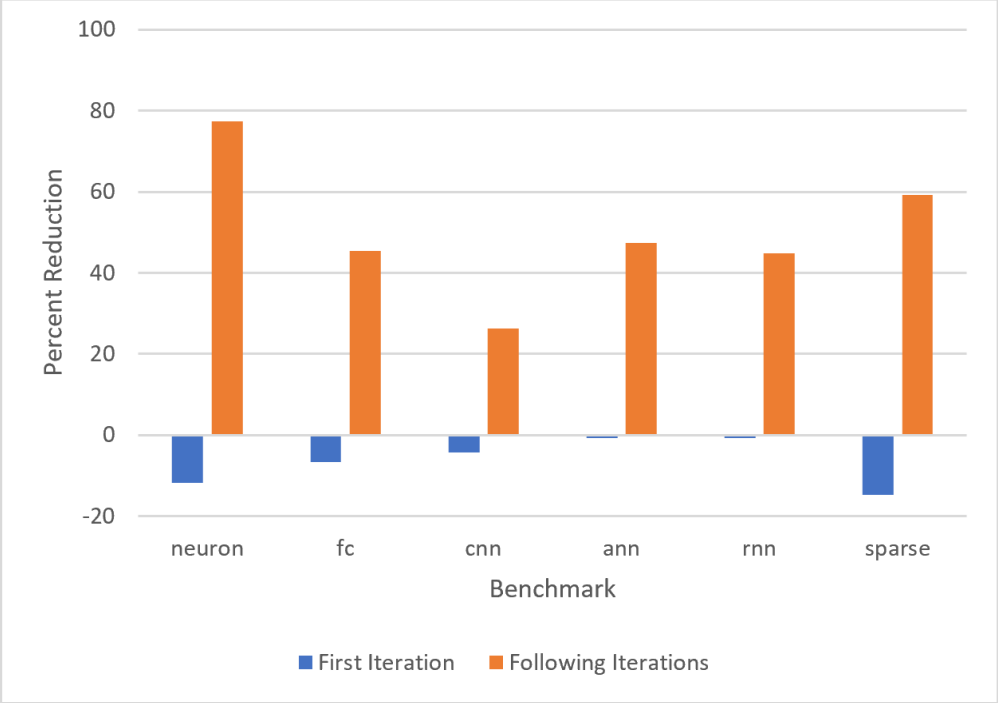


Figure 6.6. First and following iteration improvements.

6.3.2. Partial Compilation Improvements

Measurements for each network are shown in Figure 6.6. We measure the compilation time of the first iteration separate from the following iterations. Similar to the warm-up period of a cache, the first iteration experiences a penalty (1-7%) to compilation as it must compile every block and append them. Every following iteration shows improvements in compilation time (25-60%) by avoiding compiling the static blocks. By paying an upfront performance cost, we can reduce each following iteration by a substantial percentage. Following results demonstrate how this affects total execution time. It is interesting to note that the more complex networks like CNN show less improvement than circuits. This is due to the inclusion of more layers in the network and thus more blocks to append. As the number of blocks increases, the overhead for appending them comes to rival the compilation time.

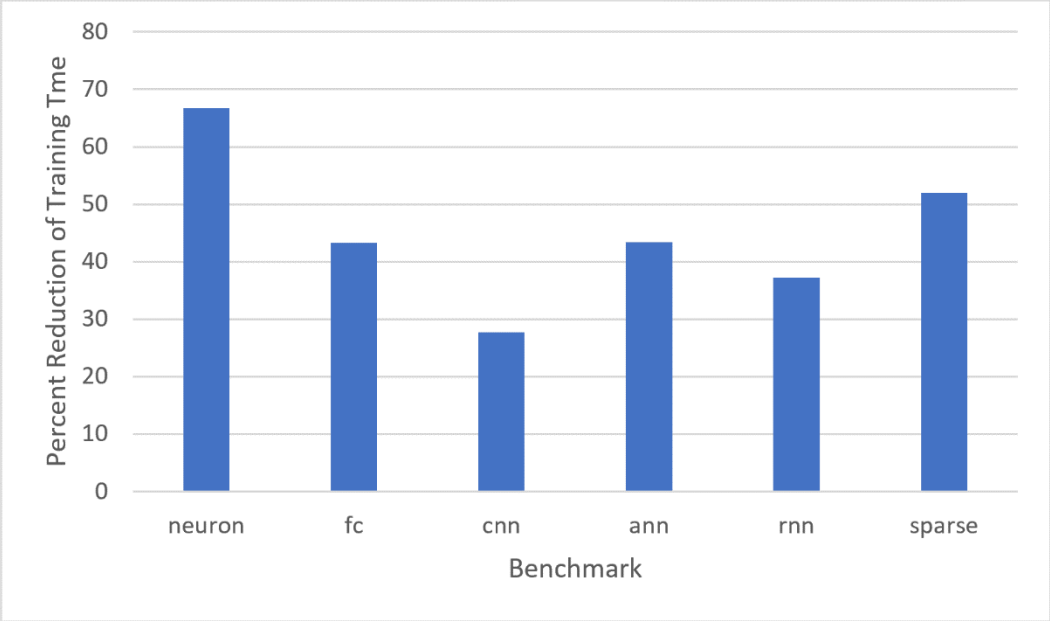


Figure 6.7. Total reduction of training time for 1000 iterations.

6.3.3. Effect on Training Time

We now demonstrate the reduction in total training time for each network. We run each circuit through 1000 training iterations with the level 3 PassManager and compare the full compilation and partial compilation training times. The reductions are shown in Figure 6.7. As expected, it resembles the per-iteration savings in Figure 6.6 at high iteration counts, as the initial iteration cost becomes less impactful. Although we have not tested in here, it is reasonable to assume that lower iteration counts will have reduced savings, while higher iteration counts will approach the per-iteration savings as a theoretical maximum. Therefore, our approach should continue to scale well as the volume of data and number of training cycles increases.

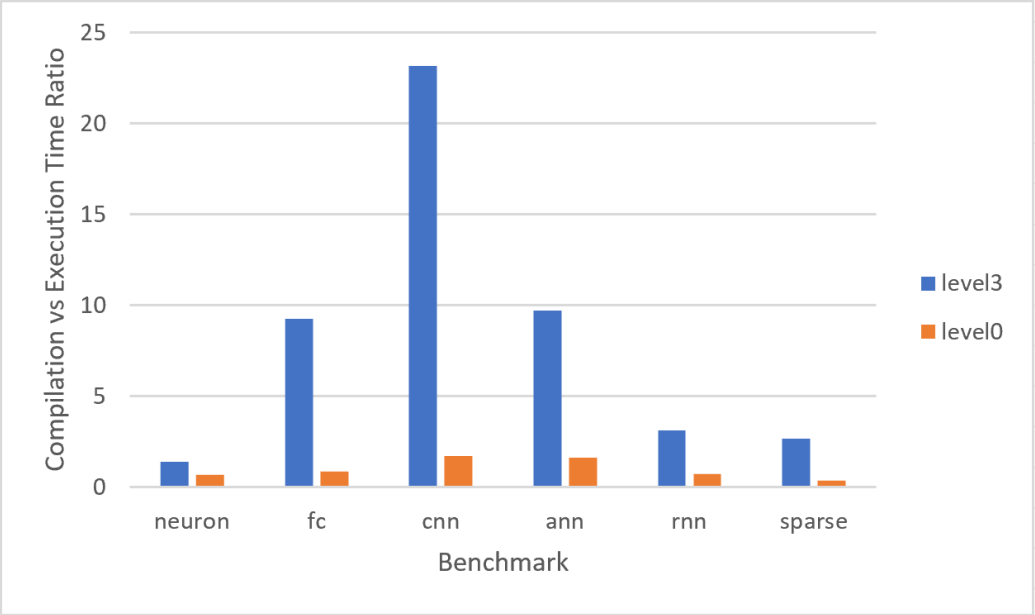


Figure 6.8. Ratio of compile times to execution times.

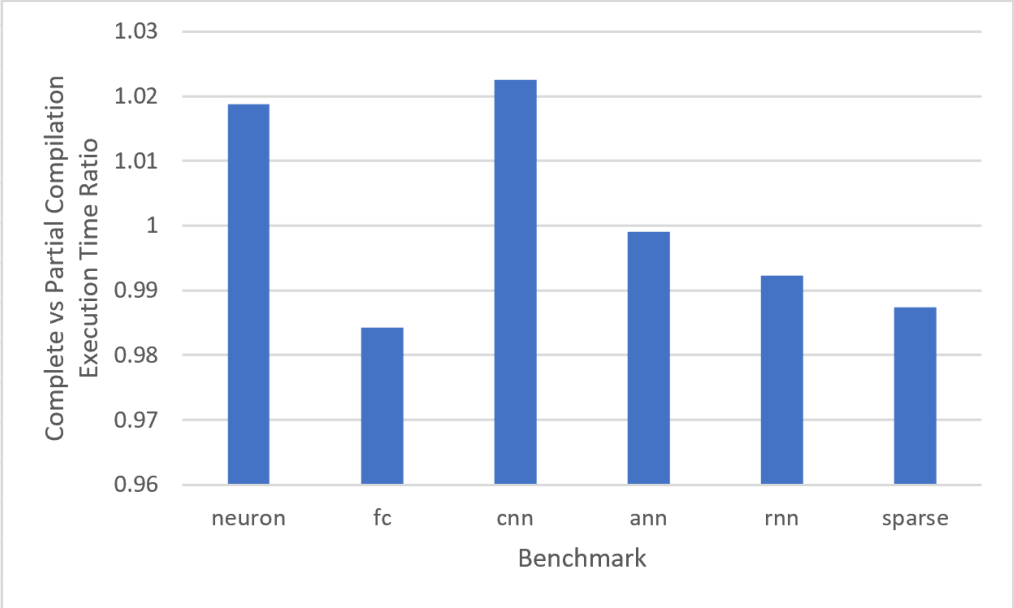


Figure 6.9. Ratio of execution times for baseline and partially compiled circuits.

6.3.4. Compilation vs Execution Time

In order to provide more clarity into the cause of previous results, we compare the previously measured compilation times to the execution times for one iteration of the networks.

To further provide insight into the impact of optimization and compiler passes, we measure both with the level zero and level three PassManager implementations. The ratio of compilation to execution time for a single iteration (not the first iteration) is shown in Figure 6.8. Compilation time exceeds execution time by multiple factors for the networks, ranging from 2x to 23x for the level 3 PassManager and 0.3x to 1.7x for the level 0 PassManager. These results enforce our previous claim that the results are dependent on the number of layers (and therefore blocks to append) contained in a network, otherwise we would expect to see greater performance gains for CNN due to its large compilation time factor. Due to the increased complexity of higher optimization levels and the improved performance when optimized, we see that compilation accounts for a substantially larger portion of time for all networks when using level3 optimization instead of level0. This also explains why our method performs better at higher optimization levels - when compilation takes substantially longer, we receive larger benefits by avoiding unnecessary compilation. If we were to execute the circuit over many iterations for training, we expect a large impact on the total training time of the network.

We also provide the ratio of execution times between the baseline (complete compilation) and partially compiled circuits to evaluate any impact on the performance of the circuit when utilizing partial compilation. These results are shown in Figure 6.9. As shown, the execution time of the partially compiled circuits are within 2.5% the execution time of the baseline circuits. This indicates that our method does not negatively impact the execution time of a circuit when accelerating the compilation time of the circuit.

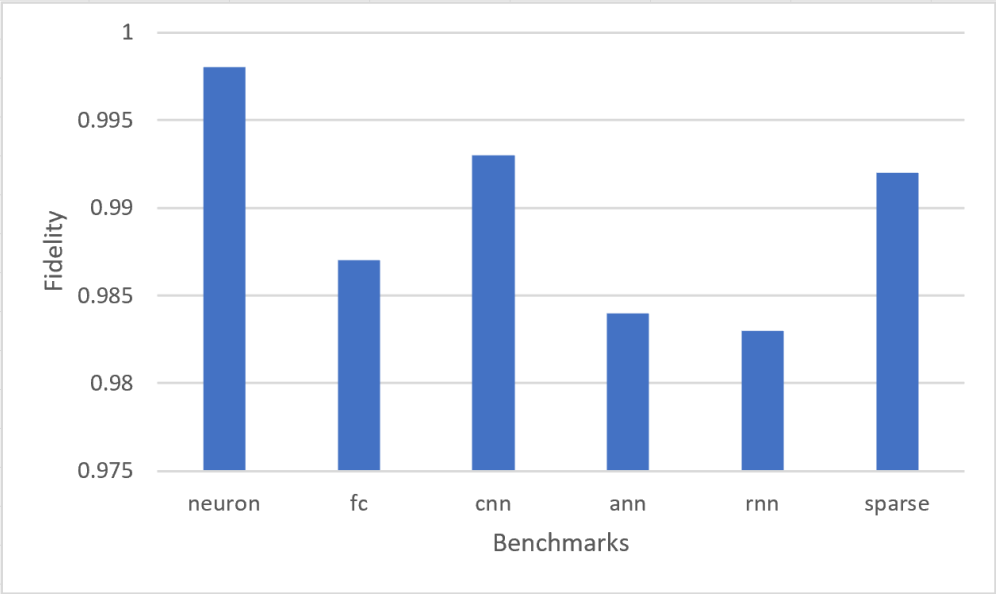


Figure 6.10. Fidelity between baseline and partially compiled circuits.

6.3.5. Fidelity

Fidelity is a commonly used metric to compute the similarity between two quantum states. The metric ranges from 0 to 1, with a value closer to 1 indicating that the two quantum states are more similar to one another. In order to identify any impact in the accuracy of the circuit, we compute the average fidelity between the baseline and partially compiled circuits after running each circuit 1000 times. These results are shown in Figure 6.10. As shown, the fidelity for each circuit is very close to 1, indicating little to no impact on the behavior of the circuit.

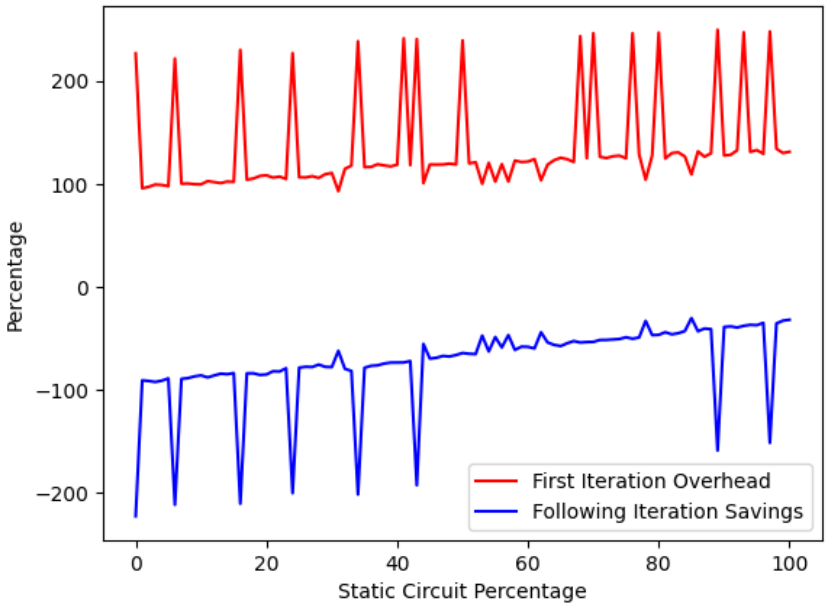


Figure 6.11. Static vs Dynamic Distribution for PassManager 0

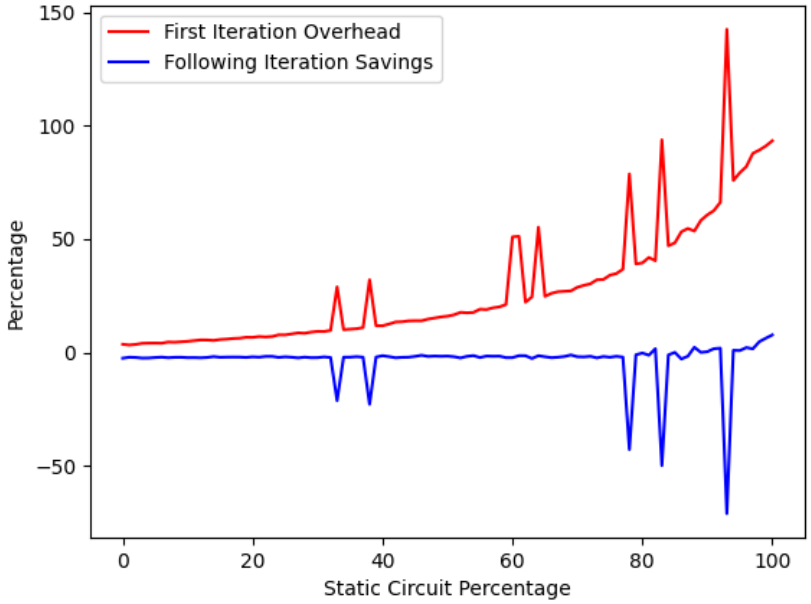


Figure 6.12. Static vs Dynamic Distribution for PassManager 1

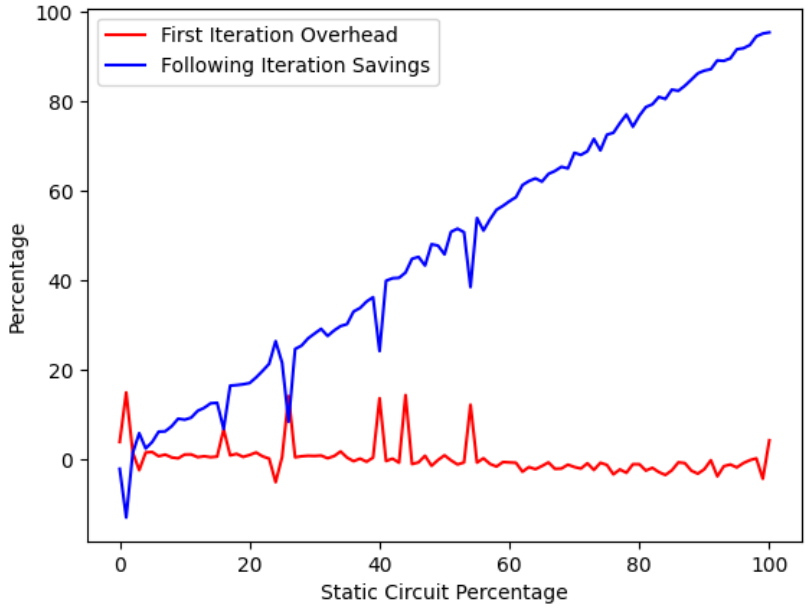


Figure 6.13. Static vs Dynamic Distribution for PassManager 2

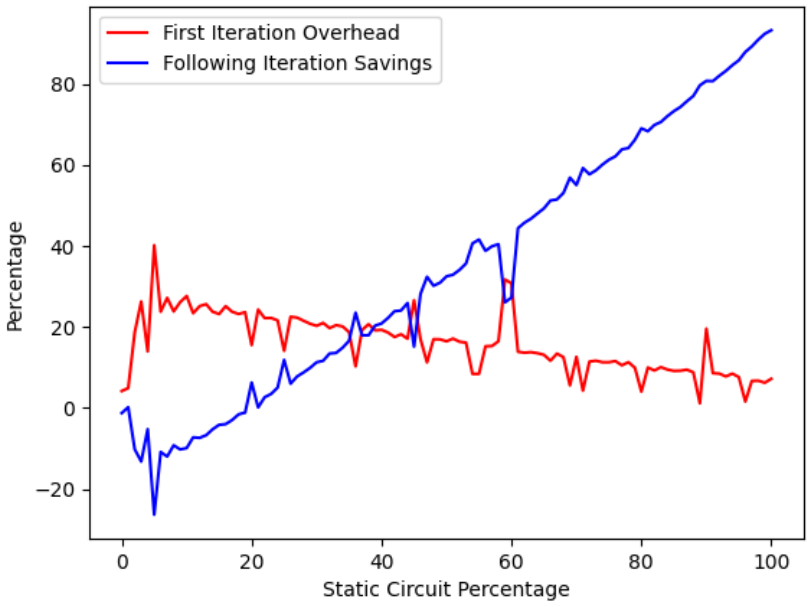


Figure 6.14. Static vs Dynamic Distribution for PassManager 3

6.3.6. Static vs Dynamic Distribution

A major factor that can influence the performance of partial compilation is the percentage of the circuit that can or cannot be recompiled. Considering extremes, if a circuit is completely static, we should have near zero compilation time. Similarly with a fully dynamic circuit, we should never expect performance above that of standard compilation. Depending on the compiler passes we are using, we expect different static/dynamic thresholds to obtain performance increases. To investigate this, we test a simple circuit of 100 gates and gradually increase the number of dynamic gates from 0 to 100. We measure both the first iteration cost and the following iteration improvements when compiling with all four PassManager optimization levels, shown in Figure 6.11 to Figure 6.14. In each figure, the threshold can be found at the point where the blue line (following iteration savings) exceeds the red line (first iteration cost). Note that this is only comparing single iterations and not the total time saved when running multiple iterations. When considering many iterations, we expect that a lower threshold may be necessary to achieve improvements.

As shown, the lower optimization levels never show a performance improvement, as the compilation times are simply too low to surpass the costs of appending the DAGs. However, at higher optimization levels, we begin to see consistent performance benefits at lower thresholds due to the increasing compilation time. As the level of optimization increases, compilation time increases, and it becomes more important to avoid recompiling the circuit when we do not need to. This is a useful observation that extends beyond our testing to neural networks, but to applications of partial compilation with Qiskit in general. Depending on the distribution of a given circuit and the optimization desired, we can assume whether or not partial compilation will be beneficial or detrimental.

6.4. Related Works

Interest in quantum machine learning has grown alongside the interest in quantum technology due to the prevalence of classical machine learning. Most methods for machine learning have analogous methods in the quantum realm, including generic supervised and unsupervised learning for classification [30, 69], artificial neural networks [9, 60, 71], deep learning networks [9, 13], and convolutional neural networks [21]. These structures are all variational in nature, exposing them to potential recompilation issues during training. Due to the great interest in these circuits, we use QNNs as our test circuits for partial compilation.

These variational algorithms are a popular subject of research beyond machine learning as well. This includes discussing their resilience to noise [9, 85, 98] and various optimizations during compilation and execution [40, 61, 72]. However, the most directly related work to our approach is that of [41], which also applies partial compilation to quantum programs. However, they use a different form of compilation known as GRAPE. Quantum gates can be thought of as rotation matrices and can be concatenated similar to multiplying rotation matrices in computer graphics. A quantum circuit can therefore be reinterpreted as a single multi-qubit gate that performs the same function as multiple smaller gates. GRAPE makes use of this method to combine multiple gates and reduce total time spent on applying gate pulses. The focus of their work is on the application of GRAPE while mitigating increased compilation latency. By comparison, our work focuses on including gate decomposition when partially compiling using the Qiskit framework.

6.5. Conclusion

In this work, we have shown the performance benefits of partial compilation on a set of quantum neural network algorithms. By combining pre-compiled static blocks with compiled-

per-iteration dynamic blocks, we can greatly reduce the compilation time, up to 77% for simple neuron circuits. This improvement allows for faster compilation and training of quantum neural networks, which will enable more widespread usage and experimentation of QNNs. Additionally, we have observed that compilation can be a substantially large portion of this training time. Reducing compilation time can achieve up to a 66% reduction in total training time alone. Future work in this area may wish to explore the impact of specific compilation passes within the four PassManager optimization levels, or to broaden the range of benchmarks we test here.

7. Improving Qubit Mapping through GNN-Assisted Compilation

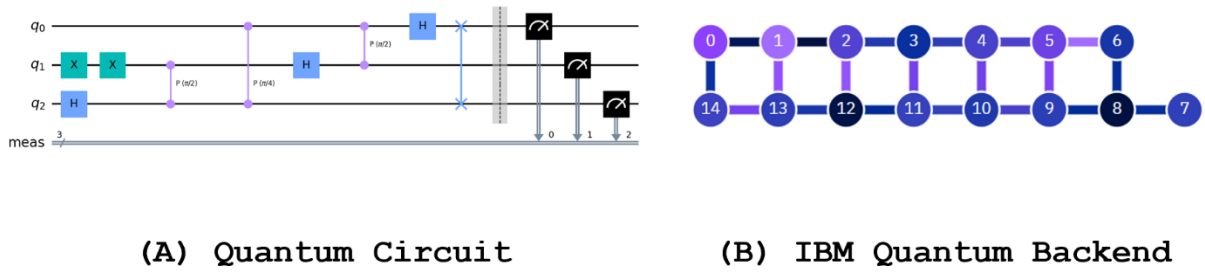
7.1. Introduction

Quantum computing has quickly become a popular field of research with great potential for future technology. Taking advantage of quantum mechanics allows for several possible operations and interactions that are not possible with classical systems. Quantum systems have the potential to improve communication, encryption, physical simulation, and some algorithms such as factorization. Many companies and governments around the world are working to develop and improve quantum systems to create quantum networks and tools. There are several potential physical implementations of quantum information systems, though the two most popular systems today are utilizing superconducting technology [19] and trapped ions [59]. Our work is done exclusively with superconducting quantum computers, as they are currently well-developed and accessible. However, the technologies, in general, are still immature and face limitations, mainly in size and reliability.

Modern quantum computers are classified as noisy intermediate-scale quantum (NISQ) devices. These NISQ devices are named as such due to their limitations on both the number of qubits (or quantum bits) available and the reliability of these qubits and their operations. Most NISQ devices contain from ten to one hundred noisy qubits, though many systems are smaller on the smaller end of this range, containing only 5-32 qubits. A common metric to evaluate these systems is quantum volume [23], which incorporates both the number of qubits and their degree of vulnerability to error. Due to the relatively high error rates in quantum computers, many executions of algorithms are unlikely to complete without some error. As such, much effort has

been put forth to both make the algorithms resilient and reduce the vulnerability of the physical qubits.

Quantum error correction (QEC) methods do exist, but they are not applicable to NISQ systems. Many QEC methods implement mechanics similar to classical replication or redundancy systems, where the data in one or more bits are encoded into a larger number of bits to reduce the effect of incident errors. However, due to the quantum no-cloning theorem, rather than copying bits, one must rely on entanglement instead. Again, one or more qubits can be entangled with additional qubits to provide redundancy and mitigate the effect of errors. Shor's code, the first to demonstrate the existence of QEC methods, encoded one qubit into 9, effectively triplicating twice to account for both phase and magnitude errors. However, when qubits are a valuable resource, it is not possible to both implement these QEC methods and retain enough qubits for computation. Many systems may not be large enough to allow for even one secure qubit depending on the error codes used.



(A) Quantum Circuit **(B) IBM Quantum Backend**

Figure 7.1. (A) An example quantum circuit, three-qubit QFT algorithm (B) An example IBM backend. Darker edge and node colors indicate higher error rates.

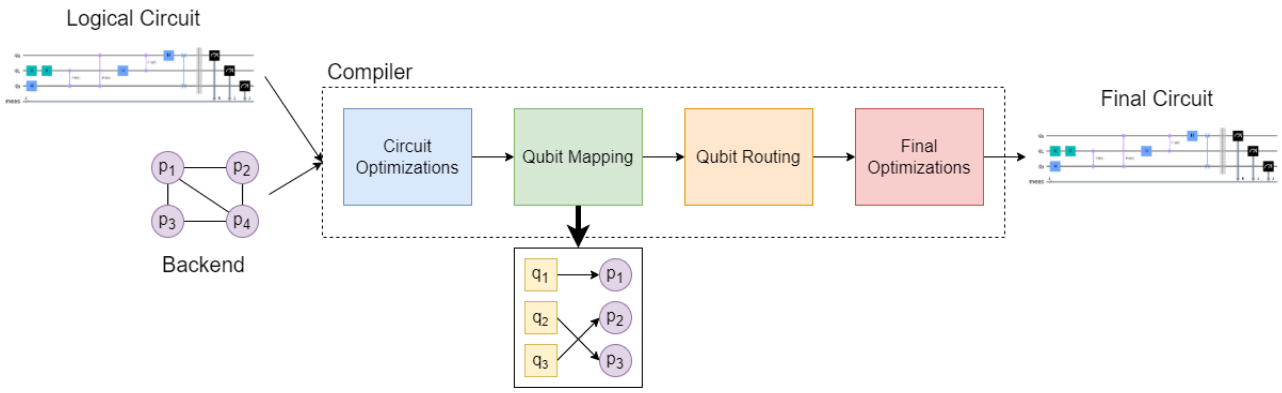


Figure 7.2. Compilation process overview, with layout mapping highlighted.

Many other approaches are used to increase the reliability of quantum circuits during execution rather than completely remove errors. Most of them modify the circuit during compilation to choose more reliable configurations when applying the circuit to a physical backend. Using different qubits, connections, and operations can have a large impact on the outcome of the circuit as the qubits may exhibit very different error profiles, as demonstrated in Figure 7.1, where the color of qubits and their connections indicate their error rates. These error rates can vary day to day with the environmental conditions. The problem is generally broken up into two parts: choosing the initial layout to map virtual qubits of the circuit to physical qubits of the backend (qubit allocation, qubit mapping, layout selection), and moving qubits through the mesh using swap operations to satisfy adjacency requirements for two-qubit operations (qubit routing, SWAP mapping). Due to a large number of possibilities when applying a circuit to a backend, it is difficult to identify the best possible configuration, though many pursuits have found success with a variety of methods [35, 75, 82, 95].

Our work aims to improve upon existing qubit allocation approaches, as our investigation shows that there are considerable performance improvements to be made. To solve the qubit allocation problem, we incorporate graph neural networks (GNNs) to aid in processing the

inherent graph representation of the superconducting quantum backend, creating a Graph Neural Network Assisted Compilation strategy (GNAQC). We combine this GNN processing of the backend with feedforward networks for processing input circuits to create a total system for providing suggested layouts as solutions to the qubit allocation problem. We implement GNAQC using Qiskit and TensorFlow and evaluate its performance on two different IBM backend configurations and six different quantum circuits. We find that GNAQC generally outperforms the other layout methods with some variation across the backends and circuits, increasing relative fidelity by approximately 12.7%. We also find that GNAQC is more consistent at choosing more effective layouts, providing a more reliable allocation method.

Our contributions can be summarized as follows:

- We demonstrate the limitations of pre-existing layout methods.
- We provide GNAQC, a new solution to the qubit allocation problem built on GCNs with feedforward networks.
- We test GNAQC on two physical backends of 7 and 27 qubits using 6 different benchmarks, finding that GNAQC can consistently provide better or comparable initial layouts to pre-existing methods.
- We demonstrate that GNAQC reduces the error of quantum circuits by providing more reliable layouts, yielding a 12.7% relative increase in fidelity.

7.2. Background

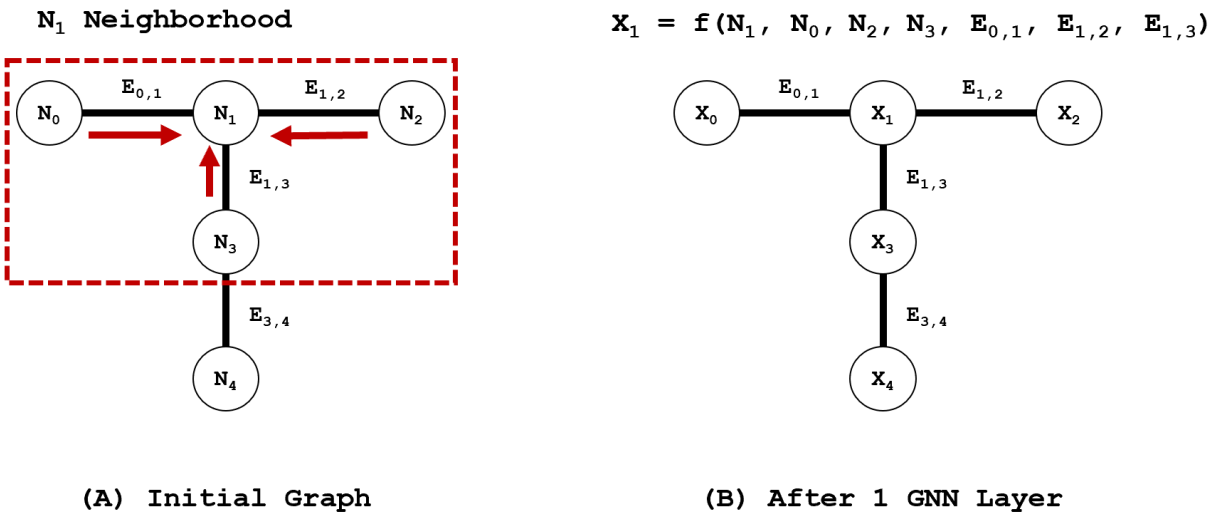


Figure 7.3. GNN update of node N1 as a function of neighboring node and edge values.

7.2.1. Graph Neural Networks (GNNs)

Graph neural networks (GNNs) are a relatively new network architecture in the neural network toolkit [28]. They are specialized in handling and interpreting graph-based data that may normally be difficult for standard feedforward networks or convolutional networks. GNNs are useful for the selection and prediction of edges and nodes, learning condensed representations of a graph as a whole, locating particular sub-graphs, specialized graph traversals, and other applications. They are particularly powerful when the data naturally has a graph representation where features of both nodes and edges are important for making decisions.

GNNs operate by sharing and diffusing information from node to node across the edges. Given an input graph, a GNN layer will compute a new representation for each node (and possibly edge) based on the values of nodes and edges within the immediate neighborhood of the node, as shown in Figure 7.3. The function is typically a weighted linear combination of the node and edge features, where the weights are learned throughout the training process. This can be

followed by an activation function similar to standard dense layers. Normally the neighborhood is the set of those nodes within one distance from the node in question, though this can be defined and restricted as necessary for a given problem. Multiple stacked GNN layers thus expand the neighborhood of a node, where the maximum distance of the neighborhood is equal to the number of stacked GNN layers. More layers effectively create a stronger diffusion of information across the graph. While this can be beneficial to share information, it has been observed that too many layers may decrease the performance of models containing GNNs as every node then tends to approach the same representation, an average of the graph as a whole. This state destroys the individual identity of each node and negatively impacts the performance of further processing. The number of recommended GNN layers varies on the problem but is generally from one to three layers depending on the size of the graph.

GNNs can be simplified by representing the layer operations as a series of matrix multiplications. One can make assumptions on the input graph to loosen the restrictions for convergence present in the original design of GNNs. The forward diffusion operation simply becomes a multiplication of the graph's normalized adjacency matrix, the node feature matrix, and the learned weight matrix, as shown in Equation 7.1 below [89].

$$X^{(k)} = \sigma(\tilde{A}X^{(k-1)}W)$$

Equation 7.1. GNN forward diffusion operation.

Here, $X^{(k)}$ is the node representation matrix, A is the renormalized adjacency matrix, W is the learned weight matrix, and σ is an activation function, commonly ReLU.

From here the process is further simplified by the observation that multiple layers are simply a repeated multiplication of the node matrix with an adjacency matrix, as the multiple weight matrices can be consolidated into one during the learning process. Ultimately, one can perform the work of multiple GNN layers by simply applying an activation function to the product of a power of the adjacency matrix, the node matrix, and the learned weight matrix, as shown in Equation 7.2 [29].

$$X^{(k)} = \sigma(\tilde{A}^k X^{(0)} W)$$

Equation 7.2. Simplified GNN equation.

Other work has been done to further enhance the applicability of GNNs to more complex types of graphs. The most relevant for our work is the addition of edge features [63]. By replacing the adjacency matrix with an edge matrix E where $E_{i,j}$ equals the weight from node i to node j, the GNN can incorporate edge features while maintaining a simple design. This can be extended if the edge has multi-dimensional features by extending the dimensions of E. It is recommended to normalize the matrix using double-stochastic normalization to accelerate training. There are additional modifications that can be made to account for directed graphs by utilizing two concatenated edge matrices.

GNNs can also be combined with other neural network architectures depending on the problem at hand. We can think of the GNN operation shown in Equation 7.2 as being in two steps: an initial propagation stage $S = AX^{(k-1)}$, followed by a linear inference stage $X^k = \sigma(SW)$. This linear stage can be replaced with other structures to solve a greater range of problems. Work has been done to demonstrate recurrent graph neural networks with gated units (GGCNs) [16, 104] and even attention-based graph neural networks [91]. While these

architectures are not used in this work, the recurrent nature of these structures may be beneficial for future investigations into the qubit routing problem.

7.2.2. Reinforcement Learning

Reinforcement learning is a form of unsupervised learning that solves problems by exploring and receiving feedback from the problem environment. An agent is allowed to observe the current state of the environment and choose an action to take, changing the state of the environment and receiving some reward. Through the learning process, the agent aims to maximize the total reward earned before reaching some terminal state of the environment. Defining a reinforcement learning problem involves describing a set of components: the set of actions an agent can take, the description of the environment (including its state and state transitions), the reward function for taking actions, the method for choosing actions, and the method for learning to maximize rewards.

The actions, environment, and rewards are directly dependent on the problem at hand. For example, if the goal is to find the shortest path in a grid-tiled environment, the actions would be the set of movements the agent can take (moving up, down, left, right), while the state of the environment would be the current position on the grid represented in (x,y) coordinates. The rewards may vary based on whether the state is a terminal or non-terminal state. Following the same example, for non-terminal states, the reward could be dependent on the distance from the current position to the end tile, while the terminal state could provide a large constant reward.

By comparison, the decision and learning methods are more general. The most commonly used decision and learning method is based on Q-learning and the Bellman optimality equation. The goal is to provide an estimation of the reward for each potential action given the current state. From these estimations, it is common to simply select the action with the greatest estimated

reward, observe the actual reward, and adjust the estimation for the (state, action) pair. These values are commonly tracked using a Q-table, containing values for every possible (state, action) pair. The simulation of the problem should then be run many times to converge to accurate reward estimations and thus accurate solutions to the problem.

As the number of states and actions grows larger, the Q-table becomes too large and unreasonable. Modern approaches instead use a neural network to learn the reward function, known as a Q-network. These Q-networks can be designed depending on the environment and the problem at hand, though they generally follow a certain structure - receiving the current state of the environment as input and providing a score for each possible action as output. These networks are then trained via standard back propagation using the error between the estimated and observed rewards.

7.3. Motivation

We utilize IBM's Qiskit API [24, 50] to investigate the current performance of qubit allocation methods. Qiskit natively contains four different allocation methods: trivial, dense, noise-adaptive [75], and sabre [35]. The four methods address the mapping problem using very different approaches. Specifically, the trivial layout simply maps the virtual qubits (q_1, q_2 … q_n), in order, to the physical qubits ($0, 1 \ldots N$). The dense layout identifies highly connected sub-graphs of the mesh and places qubits in these areas. The noise-adaptive layout is the first to rely on the most recent backend configuration data, aiming to utilize the most reliable two-qubit connections available. The sabre method utilizes an iterative process to fully route the circuit to find the final layout, then reversing the circuit using the previous final layout as a proposed initial layout. This process is repeated several times to minimize the number of necessary SWAP operations.

We tested these four layout methods on IBM's 7-qubit *ibm_nairobi* backend using 3-qubit to 7-qubit quantum phase estimation (QPE) circuits. We limit to a maximum of seven qubits as access to larger machines is limited. To evaluate their effects, we first run a trial of each circuit using Qiskit's simulator with no error involved to attain a theoretical flawless outcome that we use as the ground truth for every circuit. While the measurements of the qubits are probabilistic in nature, we execute all trials with 10000 shots to minimize the random influence. We then execute the six test circuits on the *ibm_nairobi* backend using each layout method during compilation, again using 10000 shots. All other compilation settings were kept default, including the routing methods. We then compared the resulting output distribution with the ground truth distribution by computing the fidelity between them. The fidelity acts as a similarity metric between the perfect ground-truth state and the real output state provided by the physical backend. A higher fidelity (bound [0,1]) indicates a higher similarity between states.

For ease of computing fidelity *F*, we rely on the Hellinger distance formula described below:

$$F = \frac{1}{\sqrt{2}} \sqrt{\sum_{i=1}^{N} (\sqrt{p_i^{GT}} - \sqrt{p_i^{T}})^2}$$

Equation 7.3. Hellinger Fidelity

Here, N is the total number of observed outputs, p_i^{GT} is the probability of output i for the ground truth distribution, and p_i^{T} is the probability of output i for the test distribution. The results of these trials are shown in Figure 7.4. In order to provide a metric for comparison, we decided to execute and evaluate the error of every possible layout and examine the effects. Note that this is only feasible as we are working with a relatively small number of qubits, as the total number

of layouts grows extremely quickly with an increase in qubits. We display the exact maximum fidelity achieved as *best* in Figure 7.4. As shown, we find that the layout method closest to the best is inconsistent. When looking at all 5 circuit sizes, we see situations where the trivial, noise-adaptive, and sabre methods are the closest of the four options.

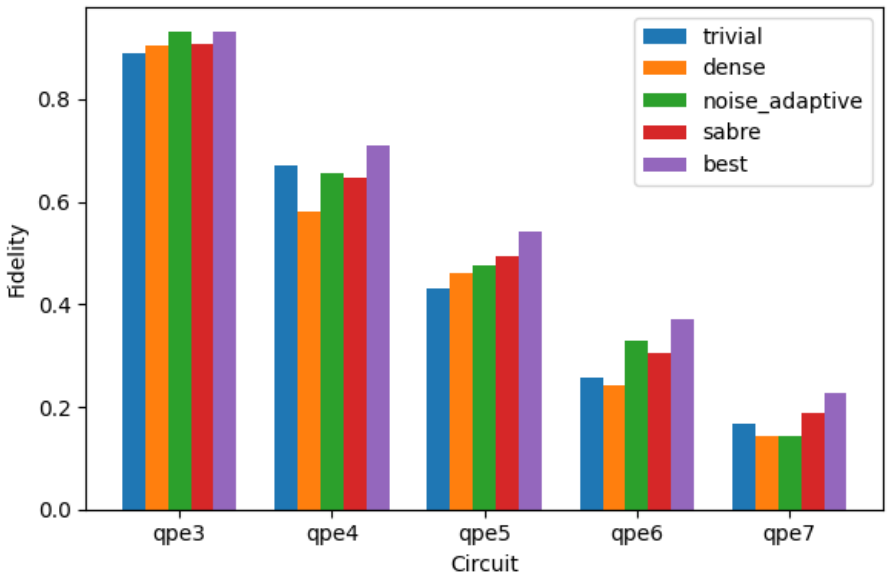


Figure 7.4. Fidelity of Qiskit's four qubit allocation methods on the (3-7)-qubit Quantum Phase Estimation (QPE) algorithm after execution on ibm_nairobi.

To provide more insight into the differences between layout methods, we evaluated every layout on one month of daily calibration data for *ibm_nairobi*, as shown in Figure 7.5. This allows us to see how frequently each allocation method performs best or worst. As expected, the best allocation method is frequently either the noise-adaptive or sabre layout methods. However, the accuracy improvements are inconsistent, and we frequently see changes between which is best over time. Occasionally, they are even outperformed by the dense or trivial layouts.

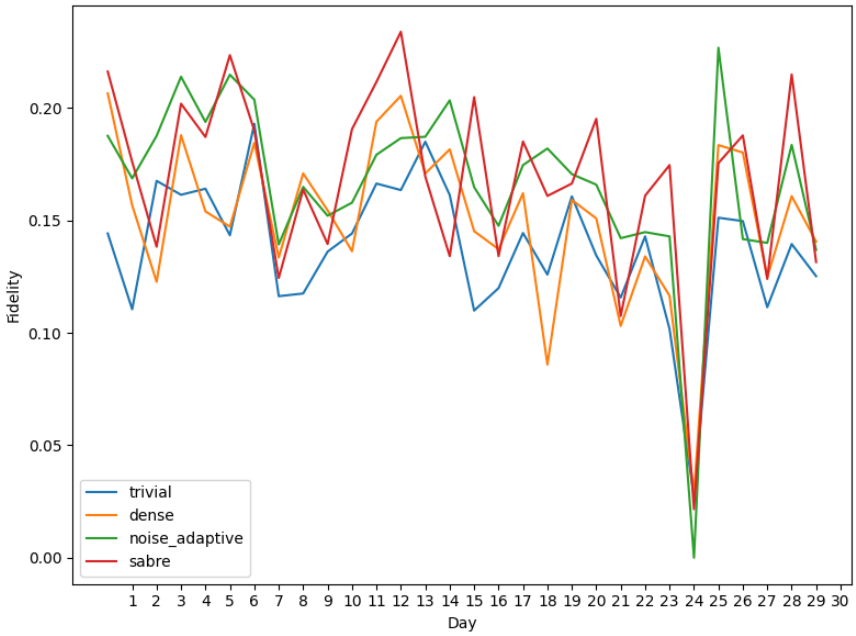


Figure 7.5. Fidelity of 7-qubit QPE when compiling with Qiskit's four allocation methods across one month of backend configurations for ibm_nairobi.

It is not expected for the choice of layout to completely remove all errors and achieve a fidelity of near 1. However, we did expect more improvements in their behavior. To investigate the full impact of the initial layout on the outcome error and provide a metric for comparison, we provide the full distribution of different layouts in Figure 7.6 for 4-qubit QPE. It is clear that no layout is perfect, though there is a large difference between the best (above 0.7 fidelity) and worst (below 0.55 fidelity) layouts, demonstrating the importance of choosing an initial layout. Additionally, the four allocation algorithms commonly fail to identify the best layout and frequently do not even choose one of the better-than-average layouts. In total, these experiments demonstrate two main points: **1)** the choice of initial layout can have a considerable impact on circuit fidelity, and **2)** pre-existing methods are inconsistent at choosing effective layouts. There is room for improvement when selecting layouts to reduce vulnerability to error.

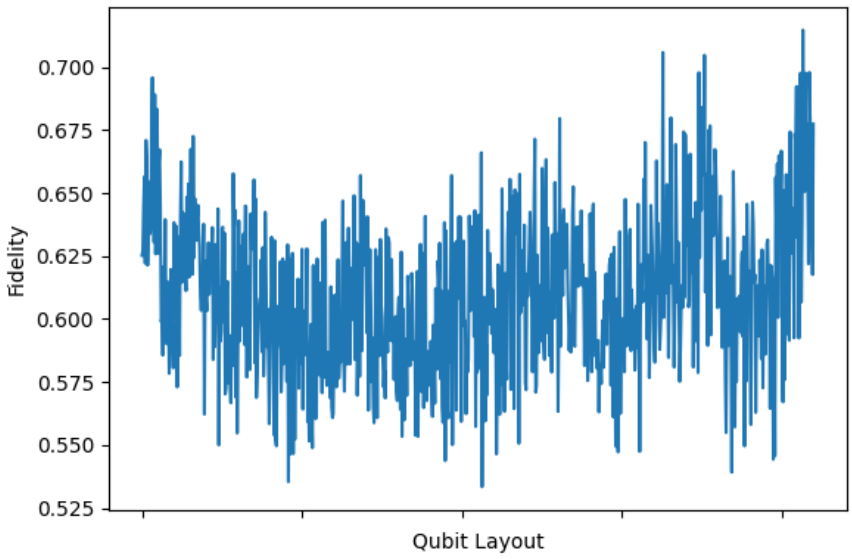


Figure 7.6. Fidelity of all possible layouts on ibm_nairobi's calibration from 01-07-2022. Results are for 4-qubit QPE.

7.4. Architecture and Data Representation

To improve the performance of current layout methods, we look to use graph neural networks as the quantum backends are naturally represented in a graph form. We combine GCNs with additional feedforward layers to predict optimal layouts given the backend error properties and an input circuit. The following subsections discuss our network architecture in detail, including three main areas: the backend graph input representation and processing, the circuit input representation and processing, and the output layout format and processing. The overall architecture is shown in Figure 7.7. Dense layers are marked with their output sizes.

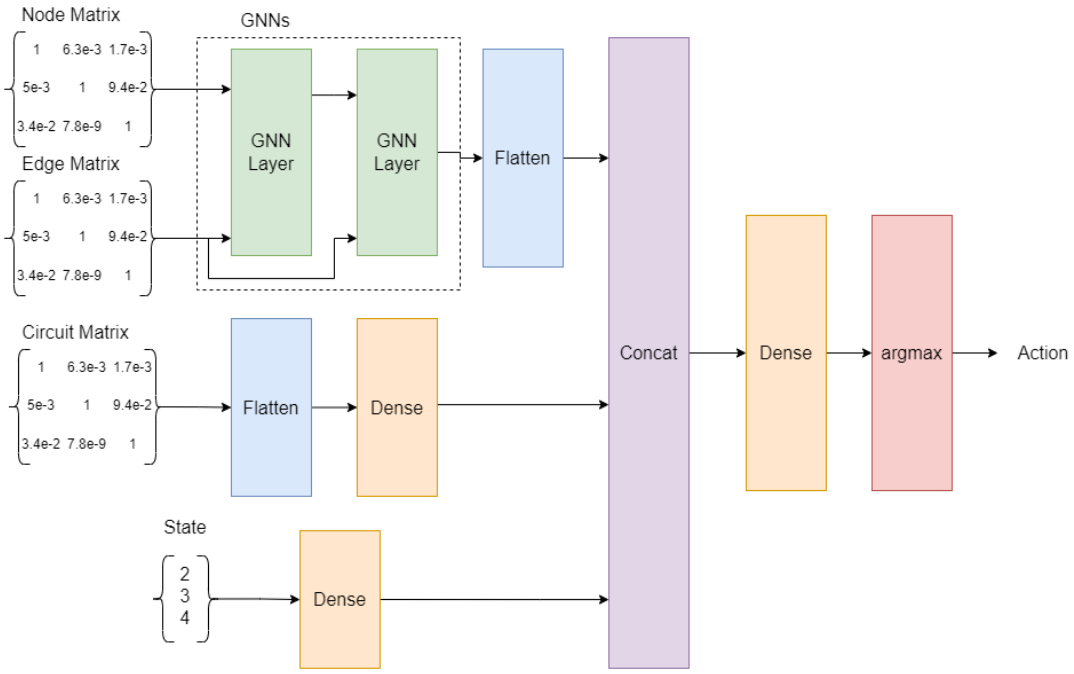


Figure 7.7. Overall architecture of the QNAQC Q-Network

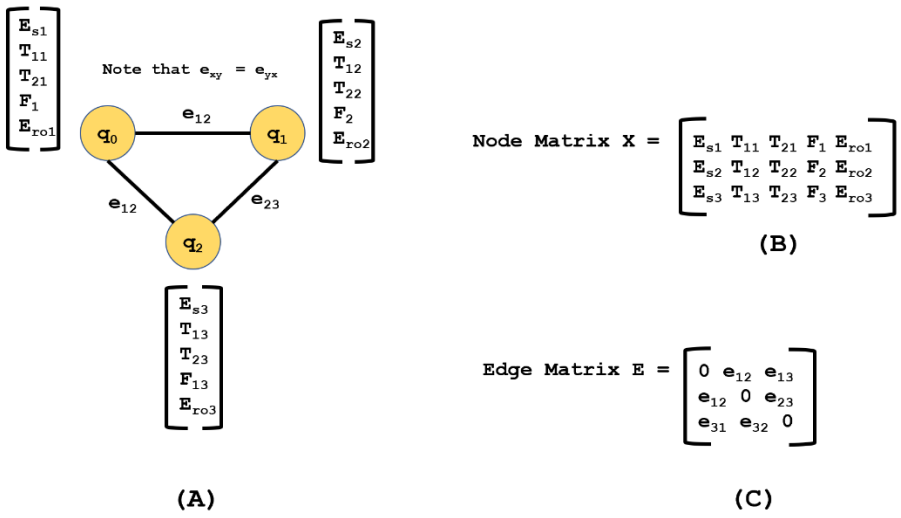


Figure 7.8. An example 3-qubit backend with five sample node features. (B) Converting the backend into both edge and node matrices for input to the layout selector.

7.4.1. Backend Representation and Processing

The superconducting backends are commonly represented as a graph, as shown in Figure 7.8B, where each node is a physical qubit with a set of properties, such as the single-qubit gate

error rates, frequencies, and measurement errors. The edges, representing available CNOT connections, are weighted by the associated CNOT error rates. This configuration naturally lends itself to the edge-aware GCN variants. A sample backend with example properties is shown in Figure 7.8A.

Table 7.1. Backend Features

Feature	Description
E_{ID}	Identity gate error
L_{ID}	Identity gate length
E_{RZ}	RZ gate error
L_{RZ}	RZ gate length
E_{SX}	SX gate error
L_{SX}	SX gate length
E_X	X gate error
L_X	X gate length
T_1	Relaxation time
T_2	Dephasing time
F	Qubit Frequency
E_M	Measurement error
P_{01}	Probability measure 0 as 1
P_{10}	Probability measure 1 as 0

To prepare the backend for the GCN layers, we must construct both a node and edge matrix (replacing the adjacency matrix in standard GNN/GCNs). For the node matrix *X*, we collect several properties from each node and arrange the matrix where each row holds the properties of an individual node, as shown in Figure 7.8B. The total set of properties that we collect is found in Table 7.1, totaling 14 different error rates and gate lengths. The final size of the node matrix is thus $N \times 14$, where *N* is the total number of physical qubits in the backend. We access these properties using Qiskit's IBMQ provider API. The set of single-qubit gate data we collect varies depending on the basis set of gates, though all of the backends we test contain the same basis set. We choose to scale the qubit frequency as it is many orders of magnitude larger than the other values. We then normalize the matrix by row to accelerate convergence.

The edge matrix *E* takes the same form as a weighted adjacency matrix, where $E_{i,j}$ equals the CNOT error between qubit *i* and qubit *j*, as shown in Figure 7.8C. Although it is not

required that the CNOT error be symmetrical on all systems, we found that, for the backends we tested, the error rates were always symmetrical. We then normalize the edge matrix in a doubly-stochastic manner, following the design of [63] to ensure that both the rows and columns of E sum to 1 to again aid in convergence. Given that the edge matrix is a variation of the adjacency matrix, its final dimensions are $N \times N$.

These two matrices are then fed into our system, specifically into two stacked GCN layers. Together these layers generate a new representation of the graph, which is then passed through a set of dense layers to condense the representation in preparation for concatenation with the processed circuit matrix. The GCN layers perform an edge-aware version of the forward computation described in Equation 7.4:

$$X^{(k)} = \sigma(\tilde{E}X^{(k-1)}W)$$

Equation 7.4. Modified forward computation equation for GNNs.

Here, E is our previously described edge matrix, while σ is the ReLU activation function.

7.4.2. Circuit Representation and Processing

To provide the circuit information to the prediction network, we first prepare a matrix containing hand-picked features to capture the behavior of the circuit. After testing a variety of different combinations, our final decision of circuit features is shown in Table 7.2. We believe that capturing the single-qubit operations each qubit, the measurement status of each qubit, the count of CNOT operations and a set of CNOT partners for each qubit is sufficient for most basic circuits. This representation would likely fail to represent more complex circuits involving mid-execution measurement and reset, though these operations do not occur in any of our test circuits. An example circuit and associated circuit matrix is shown in Figure 7.9. We simplify the

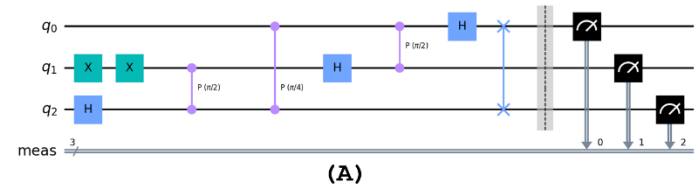
example by counting all single-qubit operations as one feature rather than individual single-qubit operation counts.

Table 7.2. Circuit Features

Feature	Description
N_{ID}	Number of involved identity operations
N_{RZ}	Number of involved RZ operations
N_{SX}	Number of involved SX operations
N_X	Number of involved X operations
N_{CNOT}	Number of involved CNOT operations
M	Measurement status
T_i	ith target qubit for CNOT operations*

It is important to note that we do not use the original logical circuit to prepare these representations, as they may change through the steps of the compilation process before preparing a layout. The most important changes that can occur are decomposing multi-qubit operations and sub-circuits and mapping to basis gates, as these can greatly change the view of which operations the circuit performs. Instead, we acquire the intermediate circuit during the compilation process at the point where qubit mapping normally occurs, after these other operations. This allows us to represent the circuit as accurately as possible for choosing a layout.

Once this feature matrix is created for the circuits, we can feed them to a set of dense layers that condenses the representation similar to that of the graph matrix after the GNN layers, as shown in Figure 7.7. The two representation vectors are then concatenated and transposed before being passed to another set of dense layers that now operate on the complete data of both learned representations of the backend and the circuit. These layers provide an encoded output layout that is then fed to a decoder network, described in the following subsection.



(A)

Circuit Matrix C =

# Gates	Measured	1st CNOT Target
5	1	2
6	1	2
5	1	1

(B)

Figure 7.9. (A) an example 3-qubit QFT circuit. (B) Constructed circuit matrix for the example circuit.

7.5. Reinforcement Learning Setup

In this section, we describe the components of the reinforcement learning process, namely the actions, environment, rewards, and training process. The overall training process can be found in Figure 7.10.

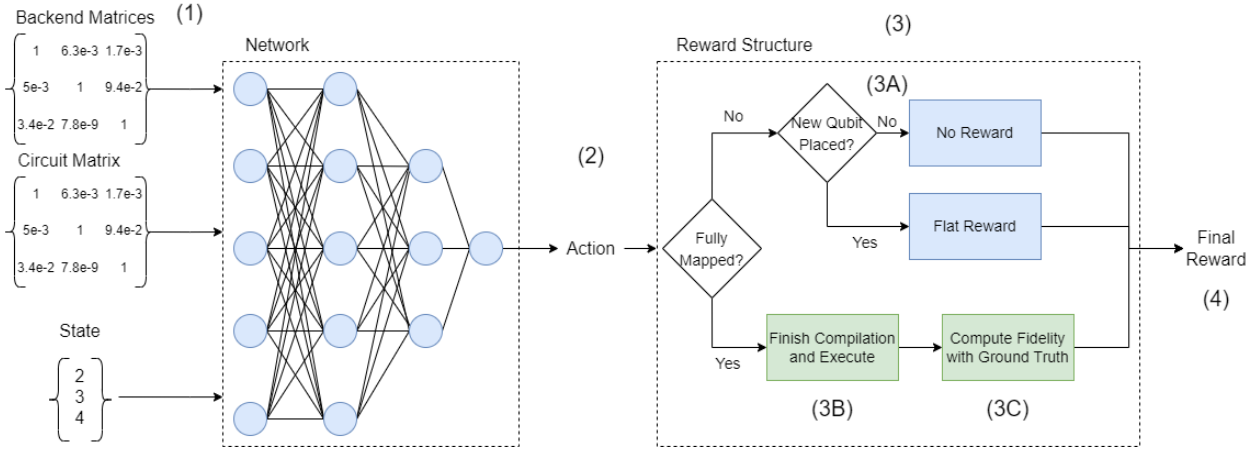


Figure 7.10. Dataflow of reward for network training

7.5.1. Actions

When mapping the qubits, the available actions are simply one placement action for each *(logical, physical)* qubit pair. This placement action represents assigning the logical qubit to the associated physical qubit for the initial layout. To account for circuits with fewer logical qubits than the available physical qubits, we extend the logical qubits with ancilla qubits to equal the number of physical qubits. In total, this results in N_{phys}^2 actions. This also characterizes the total number of outcomes resulting from the final dense layer in Figure 7.7. Following an Epsilon-Greedy policy, with $\varepsilon = 0.05$, we select the action associated with the maximum predicted value with probability *1-* ε and a random action with probability ε to drive our training decisions.

7.5.2. Environment

To define the environment, we first represent the state of the physical hardware and the circuit as described in Chapter 7.4. This requires both an edge and node matrix from the physical backend that describes the error characteristics from the latest calibration, and a circuit matrix that represents the operations that must take place for a given circuit.

These inputs are then complemented with a vector containing the current mapping of qubits, specifically mapping from physical to logical qubits. This captures the current state of the layout, specifically a snapshot of the current layout at a given time during compilation. The vector is initialized to all zero values, indicating no qubits have been placed, and gradually fills with non-zero values as placement actions are taken each iteration. Together, the matrices and state vector capture the problem itself as well as the current intermediate solution.

7.5.3. Rewards

When providing rewards, we first consider the placement of ancilla qubits. As these qubits are not important to the execution of the circuit, placing the qubits provides no reward. Similarly, when attempting to place a qubit that has already been assigned to a physical qubit, no reward is given. In contrast, placing a previously unplaced logical qubit provides a constant reward to encourage prioritization.

The most interesting case is the reward given when completing the mapping of all logical qubits. In this case, we first execute the circuit the circuit on the simulator using the error profile of the backend. We choose to use the simulator as we do not have dedicated access to a physical backend for training. We then compare the output distribution to an error free output distribution that acts as our ground truth. This error-free distribution is obtained by executing the circuit on a simulator with no error simulation. This is effectively a theoretically perfect outcome for the circuit, providing a target for comparison.

To provide a tangible value, we compute the Hellinger fidelity between the two distributions, as shown in Equation 7.3. The more similar the output distributions are, the closer this value approaches 1. This is then scaled by 100 and provided as the final reward. The fidelity guides GNAQC target configurations that are most similar to the error-free distribution. Other methods for measuring the success of quantum circuits, such as success rate, estimated success probability, and CQV [78], can also be used as metrics for learning, though we only explore the Hellinger fidelity here.

7.5.4. Training

To train the network, we rely on the Qiskit Aer simulator to simulate the execution of the circuit using the proposed mapping. We then compute the fidelity between the results of the

simulation and the ground truth output of the circuit as previously discussed. Once again, we rely on the Hellinger fidelity, as described in Equation 7.3, as our reward metric as opposed to success rate as we do not necessarily know the correct output of the given circuit with which to compute a success rate. To make the approach more general, we instead target the entire ground truth distribution using the fidelity. This error is then used for back-propagation for training the network as a whole.

The full training process is shown in Figure 7.10. First, the processed edge, node, and circuit matrices are fed to the prediction network in step (1). The network outputs a suggested action to take, namely a qubit placement, in step (2). The reward for this action is calculated in step (3), where the value for the reward depends on the result of the action. If the action results in a fully-mapped circuit, we finish compilation (routing and final optimization) and simulate the final circuit in step (3B) using Qiskit's Aer simulator. The simulator is prepared with a noise model built on the error properties of the collected backend under test. In step (3B), we collect the output counts from the simulator and compute the fidelity with the ground truth distribution. If the action did not result in a fully-mapped circuit, we instead give either a reward of 0 if the qubit was already placed or 10 if the qubit is newly placed. Finally, we use this reward for the update process following the typical Q-learning update rule in step (4).

It is worth noting here that we do not need to rely on the simulator, which will not be feasible for increasingly large circuits and backing, during this training process. We could rely on other success metrics, like estimated success probability (ESP), which may be more scalable. However, we chose to use the simulator to be more accurate to the physical hardware. Ideally, one would have dedicated access to a physical machine for the training process, which would address both the accuracy and scalability concerns.

7.6. Data Collection and Experimentation

Throughout our experimentation, we rely on a set of various test circuits at different sizes executed on several different physical backends. We focus on a set of six different circuits as mentioned previously in Chapter 7.3: the Deutsch-Jozsa (DJ) algorithm, the Bernstein-Vazirani (BV) algorithm, Simon's algorithm, the quantum Fourier transform (QFT), the quantum phase estimation (QPE) algorithm, and Grover's search algorithm. We prepare these circuits using 3, 4, 5, 6, 7, 15, and 27 qubits. We believe that two qubits are simply too trivial, and we are limited to evaluating on backends with 7 or 27 maximum qubits. The characteristics of these circuits, specifically the count of the final gates used for each algorithm, at each circuit size are detailed in Table 7.3.

Table 7.3. Benchmark Details

Name	Description	#Qubits	#1Q	#2Q
DJ	Deutsch-Jozsa	3	30	12
		5	128	60
		7	524	248
		15	6584	3152
		27	21880	10398
BV	Bernstein-Vazirani	3	30	12
		5	128	60
		7	524	248
		15	6584	3152
		27	21880	10398
$Simon$	Simon's Algorithm	3	26	12
		5	128	60
		7	484	248
		15	6328	3152
		27	18468	10398
QFT	Quantum Fourier Transform	3	17	9
		4	28	18
		5	43	26
		6	224	78
		7	457	208
		15	2076	1226
		27	16937	3688
QPE	Quantum Phase Estimation	3	21	8
		4	43	20
		5	80	42
		6	240	98
		7	483	244
		15	2206	1486
		27	18164	4094
$Grover$	Grover's Search Algorithm	3	98	30
		4	202	102
		5	459	256
		6	905	428
		7	1820	768
		15	10238	5820
		27	89374	20838

For the backends, we collected calibrations for *ibm_nairobi*, a 7-qubit backend, and *ibm_algiers*, a 27-qubit backend. We selected these two as a sample of the available 7-qubit and 27-qubit machines we can access. We specifically collected the archived daily calibrations from January 1st, 2022 through the end of May 2022. The backends vary in topology, with *ibm_nairobi* having an I shape and *ibm_algiers* having an adjusted square shape. Both backends share the same set of basis gates. These details are summarized in Table 7.4.

Table 7.4. Backend Details

Name	# Qubits	Basis Gates	Topology
ibm_nairobi	7	CX, ID, RZ, SX, X	I
ibm_algiers	27	CX, ID, RZ, SX, X	Square

7.7. Results

To evaluate the general performance of GNAQC, we predict layouts for the circuits using the most recent calibrations at their time of execution. The historical backend calibrations are used for training. We compare these results to the previously measured errors for the four layout methods contained within Qiskit. These results are shown in Figure 7.11.

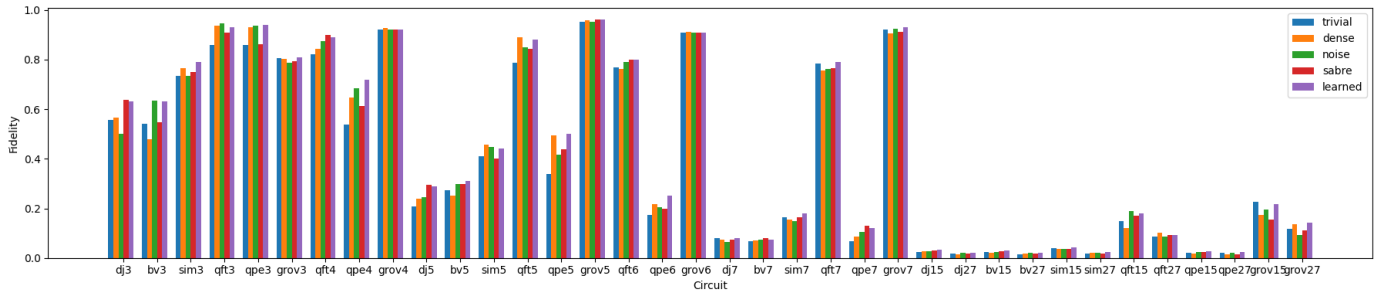


Figure 7.11. Total results for all layout methods on all benchmarks.

It can be observed that the GNAQC layout generally outperforms the pre-existing layouts for each benchmark at different algorithm sizes. The GNAQC layouts consistently perform better on simpler algorithms like DJ, BV, and Simon. There is reduced, though fairly consistent, improvement on the larger algorithms. On average, however, we see a relative improvement in fidelity of approximately 12.7%.

We group the data by each circuit regardless of backend or qubit size to inspect the mean performance on the individual algorithms. These results are shown in Figure 7.12.

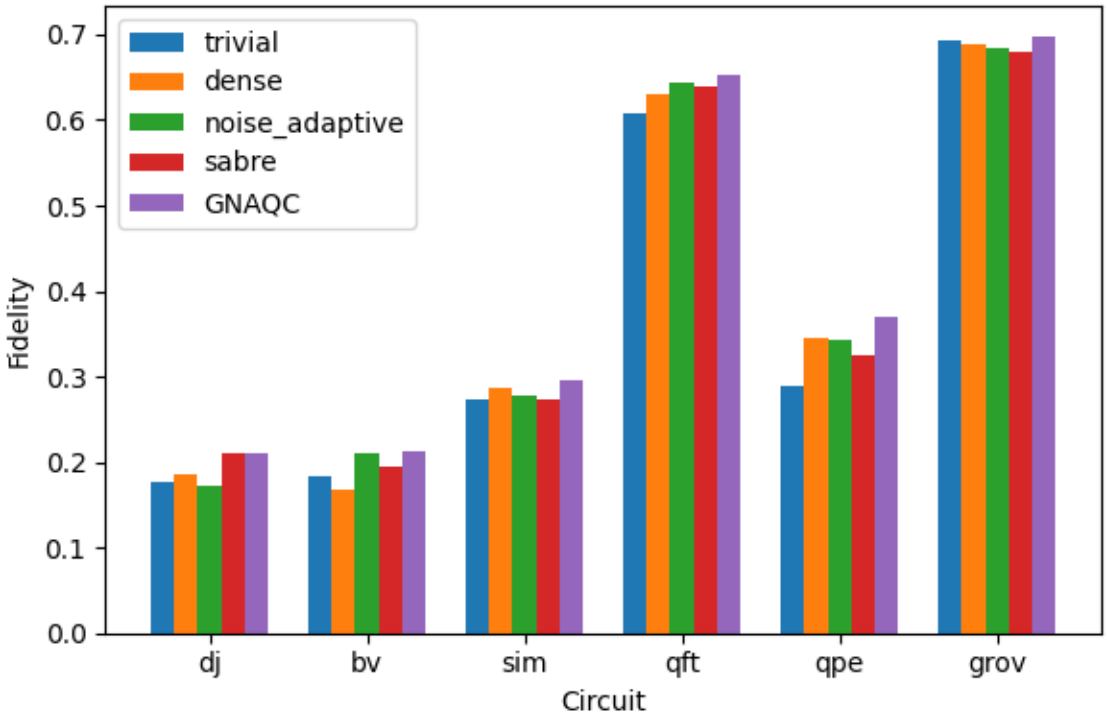


Figure 7.12. Fidelity of layout methods for each circuit.

Here, GNAQC layouts show improvement or consistent behavior on most circuits. In the worse cases, GNAQC performs comparably to the best alternative layout method. The magnitude of the improvement varies from circuit to circuit and is also dependent on the next-best choice. In general, we believe this variation is due to the effect different layouts have depending on the length of the algorithm, where shorter circuits are simply more influenced by the initial position of qubits, while longer algorithms are likely more influenced by the routing methods.

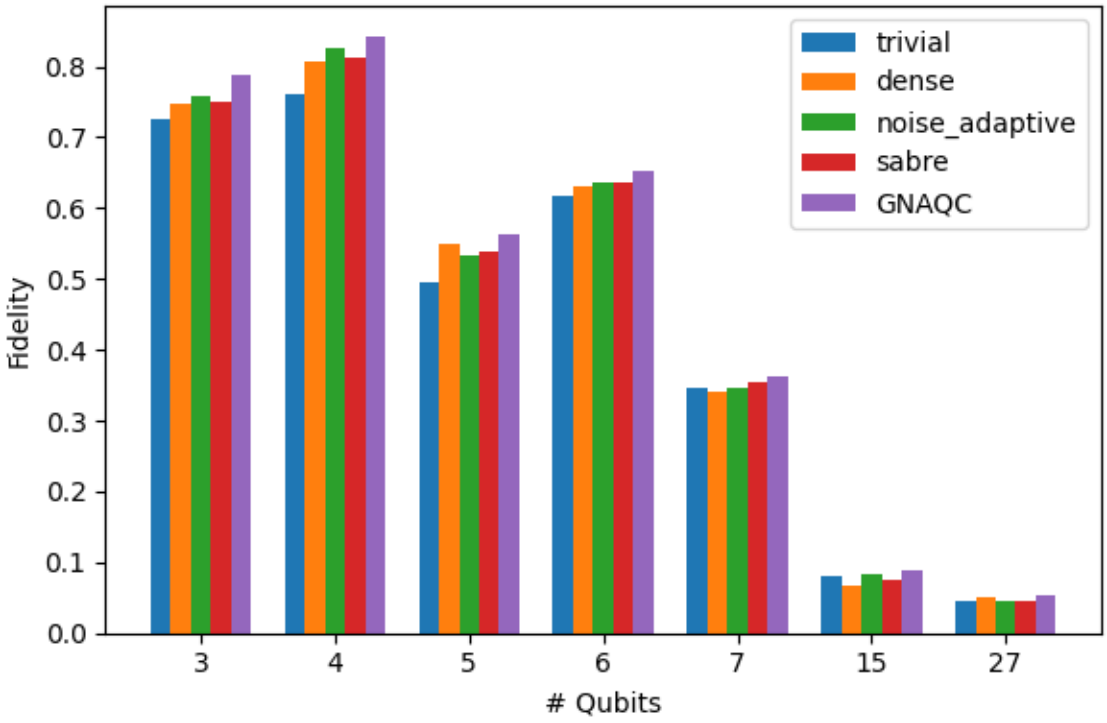


Figure 7.13. Fidelity of different layout methods grouped by number of qubits in circuit.

Next, we group the results by the number of qubits involved in the algorithm to observe performance based on the size of the circuits, as shown in Figure 7.13. As we can see, the largest improvement is found in smaller circuit sizes. We see the most variation in behavior among the layouts at 3-5 qubits, with more consistent performance among all five methods at larger sizes. We identify two main reasons for this variation in behavior. First, as the depth of the circuit increases due to the increased number of qubits, the fidelity decreases drastically. This results in less room for the layouts to vary as the fidelity is simply so low. Second, we believe that this has to do with the percentage of qubits used on the backend and the topology of the machine itself. When using all of the qubits on the machine, more SWAPs will likely need to be added to allow the circuit to function regardless of the initial position of qubits. At smaller sizes, particularly three qubits, the number of added SWAPs may vary greatly based on the initial position of

qubits. It may be possible to place them in a configuration where no SWAPs are necessary, such as a triangle section in the mesh, or to place them at opposite sides of the mesh where many SWAPs are necessary.

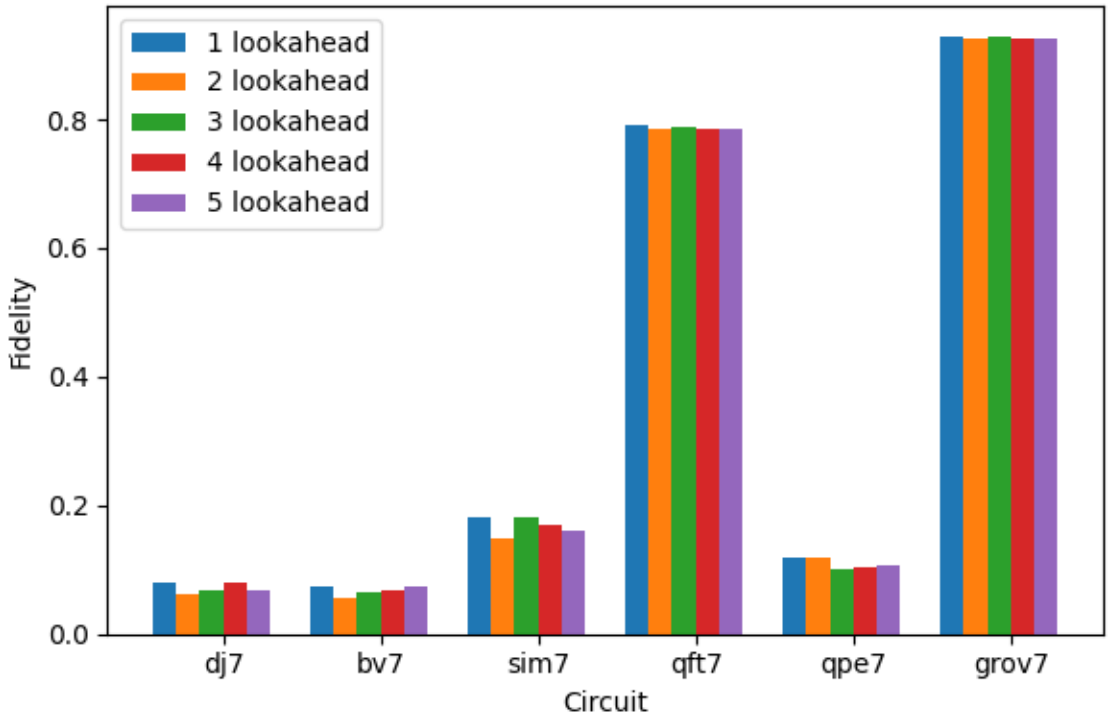


Figure 7.14. Fidelity of GNAQC with varying lookahead

In all of the previous results shown, we have utilized a CNOT look-ahead window of length 1, as described in Chapter 7.4.2. However, we also examined a variable look-ahead window from length 1 to 5. For a chosen look-ahead value *LA*, our circuit matrix will hold the first *LA* CNOT targets for each qubit. We evaluated our method for all $1 \leq LA \leq 5$ for 7-qubit circuits using a simulator based on the properties of *ibm_nairobi*. We chose to use the simulator here due to time restrictions and the number of executions necessary. The results are shown in Figure 7.14. As shown, the value of *LA* does not have a large impact on performance. This could be due to insufficient training data when increasing the number of parameters.

7.8. Related Works

Increasing circuit resilience to errors is a major field in quantum computing research. A common approach involves modifying compilation, either the allocation or routing passes. [75] and [82] were the first to identify that routing should focus not only on the number of inserted SWAP operations but also the reliability of the qubit connections. This included a modified routing method to be aware of the CNOT error rates. [74, 101] observed that performing CNOT operations in parallel with other nearby operations would increase error rates, suggesting that routing methods should plan for this cross-talk error and attempt to avoid parallel operations on adjacent qubits where possible. [83] has suggested that we can improve reliability by executing a circuit in multiple parts then reconstructing the overall distribution. This allows for focus on only a few qubits and using only the most reliable physical qubits and avoiding unnecessary cross-talk.

Meanwhile, the research on debugging, protecting, and reusing resources is also trying to find solutions to mitigate the constraints of error in the field. The quantum assertion technique was proposed and evolved in [54, 55, 103] to locate the errors and bugs while running quantum algorithms. Applying quantum error correction to superconducting quantum chips is also being actively studied in [44, 46, 57, 80]. Reusing the valuable quantum resources, the physical qubits, is being studied in [102].

The two most relevant allocation methods are the two already contained within Qiskit, the noise-adaptive [75] and sabre [35] methods. Other methods utilize locating the optimal layout at small sizes to produce heuristics that are then tested on larger-scale systems [5, 70, 76]. [95] similarly searches the set of possible layouts while guided by fidelity. All of these approaches aim to minimize the vulnerability of the circuit through the choice of an initial layout.

The base GNN [28] design has many modifications for a variety of different applications. [16] and [104] both build on the original GNN design to include recurrent units. Graph convolutional networks (GCNs) simplify the process and improve upon the original design for node classification tasks [29, 89]. [96] and [77] are frequently used to learn graph representations. [81] similarly takes an adversarial approach to learn graph embeddings. The idea of using GNNs to improve compilation has similarly been used in hardware placement for classical circuits [1, 36, 105].

7.9. Conclusion

We have proposed GNAQC, a new GNN-based neural network architecture for improving the reliability of superconducting quantum circuits by identifying more resilient layouts. We compare the learned layouts with the results of pre-existing layout methods contained within Qiskit and find a mean 12.7% relative increase in fidelity across both backends configurations with six different circuits. In the future, we believe we could achieve even greater results by expanding the work to include a routing method using recurrent GNNs or experimenting with different feature representations.

8. Conclusion

In this work, we have presented a variety of methods to improve classical and quantum program reliability. For classical programs, we first improve the reliability of single-threaded Big Data kernels using algorithm specific error checking or redundancy mechanisms. Our observations demonstrated that algorithmic invariants can provide low-cost error detection methods that, when combined with recovery mechanisms, can greatly increase the success of the Big Data kernels. We also investigated the error vulnerability of synchronization mechanisms in multi-threaded implementations of the kernels and aimed to protect them using fine-grained logging. We found that this approach can nearly eliminate errors within the synchronization mechanisms while maintaining program correctness. Additionally, the overhead for the logging mechanisms is smaller than what is normally experienced with transactional memory implementations, a system commonly used to replace locks in parallel programs that also relies on fine-grained logging of thread behavior.

For quantum circuits, we've attempted to bring QEC methods to NISQ hardware using prioritized cache qubits and varying error correction codes while minimizing swap overhead. The cache hierarchy allows for the usage of different error correction codes to help save qubits if the extended reliability is not needed. However, to apply this cache hierarchy to SQCs, we need to adjust the qubit routing methods during compilation to account for cache qubits. We have introduced two cache-aware routing methods built on previous routing methods that minimize the number of cache-involved SWAPs during compilation.

In addition, we have discussed two additional methods to improve quantum circuit performance and reliability using partial compilation and GNN-assisted compilation. The first method takes advantage of the fact that some portions of variational circuits, namely quantum

neural networks, do not change and thus do not need to be recompiled when adjusting weights. By separating the circuit into static and dynamic blocks, we recompile only the dynamic blocks then append them with the pre-compiled static blocks to recreate the final circuit. The goal is to reduce the training time for these networks by avoiding recompiling the entire circuit every time we adjust the weights.

The other quantum compilation improvement we have implemented is GNN-assisted compilation, namely for qubit allocation. By using GNNs to process the graph-structured information inherent in the SQC qubit meshes, we can create a noise-aware compiler that can more effectively make decisions about qubit allocation and routing during compilation. The GNN is combined with a standard feed-forward network for processing the quantum circuit information to complete the compilation process. The network can be trained using feedback from the real physical machines, a simulator, or success rate estimations based on the provided error information of the backend through reinforcement learning. Results show promising improvements over pre-existing methods, though experiments at larger scales show less improvement due to the relatively small success rates.

Overall, error vulnerability in classical and particularly quantum computers is a pressing problem that must be addressed to guarantee safe, accurate, and consistent computing. The works presented within this work investigate different methods to solve these problems. In particular, the quantum solutions can help make quantum computing a more reliable and accessible technology.

Appendix. Copyright Information

A.1. Publishing Agreement for Soft Error Resilience of Big Data Kernels through Algorithmic Approaches

SPRINGER NATURE LICENSE
TERMS AND CONDITIONS

Nov 03, 2022

This Agreement between Louisiana State University -- Travis LeCompte ("You") and Springer Nature ("Springer Nature") consists of your license details and the terms and conditions provided by Springer Nature and Copyright Clearance Center.

License Number	5421380911091
License date	Nov 03, 2022
Licensed Content Publisher	Springer Nature
Licensed Content Publication	Journal of Supercomputing, The
Licensed Content Title	Soft error resilience of Big Data kernels through algorithmic approaches
Licensed Content Author	Travis LeCompte et al
Licensed Content Date	Apr 18, 2017
Type of Use	Thesis/Dissertation
Requestor type	academic/university or research institute
Format	electronic
Portion	full article/chapter
Will you be translating?	no

Circulation/distribution	50000 or greater
Author of this Springer Nature content	yes
Title	Compilation Optimizations to Enhance Resilience of Big Data Programs and Quantum Processors
Institution name	Louisiana State University
Expected presentation date	Dec 2022
Order reference number	1
Requestor Location	Louisiana State University 3423 Brightside Dr. BATON ROUGE, LA 70820 United States Attn: Louisiana State University
Total	0.00 USD

Terms and Conditions

Springer Nature Customer Service Centre GmbH
Terms and Conditions

This agreement sets out the terms and conditions of the licence (the **Licence**) between you and **Springer Nature Customer Service Centre GmbH** (the **Licensor**). By clicking 'accept' and completing the transaction for the material (**Licensed Material**), you also confirm your acceptance of these terms and conditions.

1. Grant of License

1. 1. The Licensor grants you a personal, non-exclusive, non-transferable, world-wide licence to reproduce the Licensed Material for the purpose specified in your order only. Licences are granted for the specific use requested in the order and for no other use, subject to the conditions below.

1. 2. The Licensor warrants that it has, to the best of its knowledge, the rights to license reuse of the Licensed Material. However, you should ensure that the material you are requesting is original to the Licensor and does not carry the copyright of

another entity (as credited in the published version).

1. 3. If the credit line on any part of the material you have requested indicates that it was reprinted or adapted with permission from another source, then you should also seek permission from that source to reuse the material.

2. Scope of Licence

2. 1. You may only use the Licensed Content in the manner and to the extent permitted by these Ts&Cs and any applicable laws.

2. 2. A separate licence may be required for any additional use of the Licensed Material, e.g. where a licence has been purchased for print only use, separate permission must be obtained for electronic re-use. Similarly, a licence is only valid in the language selected and does not apply for editions in other languages unless additional translation rights have been granted separately in the licence. Any content owned by third parties are expressly excluded from the licence.

2. 3. Similarly, rights for additional components such as custom editions and derivatives require additional permission and may be subject to an additional fee. Please apply to Journalpermissions@springernature.com/bookpermissions@springernature.com for these rights.

2. 4. Where permission has been granted **free of charge** for material in print, permission may also be granted for any electronic version of that work, provided that the material is incidental to your work as a whole and that the electronic version is essentially equivalent to, or substitutes for, the print version.

2. 5. An alternative scope of licence may apply to signatories of the STM Permissions Guidelines, as amended from time to time.

3. Duration of Licence

3. 1. A licence for is valid from the date of purchase ('Licence Date') at the end of the relevant period in the below table:

Scope of Licence	Duration of Licence
Post on a website	12 months
Presentations	12 months
Books and journals	Lifetime of the edition in the language purchased

4. Acknowledgement

4. 1. The Licensor's permission must be acknowledged next to the Licenced Material in print. In electronic form, this acknowledgement must be visible at the same time as the figures/tables/illustrations or abstract, and must be hyperlinked to the journal/book's homepage. Our required acknowledgement format is in the Appendix below.

5. Restrictions on use

5. 1. Use of the Licensed Material may be permitted for incidental promotional use and minor editing privileges e.g. minor adaptations of single figures, changes of format, colour and/or style where the adaptation is credited as set out in Appendix 1 below. Any other changes including but not limited to, cropping, adapting, omitting material that affect the meaning, intention or moral rights of the author are strictly prohibited.

5. 2. You must not use any Licensed Material as part of any design or trademark.

5. 3. Licensed Material may be used in Open Access Publications (OAP) before publication by Springer Nature, but any Licensed Material must be removed from OAP sites prior to final publication.

6. Ownership of Rights

6. 1. Licensed Material remains the property of either Licensor or the relevant third party and any rights not explicitly granted herein are expressly reserved.

7. Warranty

IN NO EVENT SHALL LICENSOR BE LIABLE TO YOU OR ANY OTHER PARTY OR ANY OTHER PERSON OR FOR ANY SPECIAL, CONSEQUENTIAL, INCIDENTAL OR INDIRECT DAMAGES, HOWEVER CAUSED, ARISING OUT OF OR IN CONNECTION WITH THE DOWNLOADING, VIEWING OR USE OF THE MATERIALS REGARDLESS OF THE FORM OF ACTION, WHETHER FOR BREACH OF CONTRACT, BREACH OF WARRANTY, TORT, NEGLIGENCE, INFRINGEMENT OR OTHERWISE (INCLUDING, WITHOUT LIMITATION, DAMAGES BASED ON LOSS OF PROFITS, DATA, FILES, USE, BUSINESS OPPORTUNITY OR CLAIMS OF THIRD PARTIES), AND
WHETHER OR NOT THE PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THIS LIMITATION SHALL APPLY NOTWITHSTANDING ANY FAILURE OF ESSENTIAL PURPOSE OF ANY LIMITED REMEDY PROVIDED HEREIN.

8. Limitations

8. 1. ***BOOKS ONLY:***Where **'reuse in a dissertation/thesis'** has been selected the following terms apply: Print rights of the final author's accepted manuscript (for clarity, NOT the published version) for up to 100 copies, electronic rights for use only on a personal website or institutional repository as defined by the Sherpa guideline (www.sherpa.ac.uk/romeo/).

8. 2. For content reuse requests that qualify for permission under the STM Permissions Guidelines, which may be updated from time to time, the STM Permissions Guidelines supersede the terms and conditions contained in this licence.

9. Termination and Cancellation

9. 1. Licences will expire after the period shown in Clause 3 (above).

9. 2. Licensee reserves the right to terminate the Licence in the event that payment is not received in full or if there has been a breach of this agreement by you.

Appendix 1 — Acknowledgements:

For Journal Content:
Reprinted by permission from [**the Licensor**]: [**Journal Publisher** (e.g. Nature/Springer/Palgrave)] [**JOURNAL NAME**] [**REFERENCE CITATION** (Article name, Author(s) Name), [**COPYRIGHT**] (year of publication)

For **Advance Online Publication papers:**
Reprinted by permission from [**the Licensor**]: [**Journal Publisher** (e.g. Nature/Springer/Palgrave)] [**JOURNAL NAME**] [**REFERENCE CITATION** (Article name, Author(s) Name), [**COPYRIGHT**] (year of publication), advance online publication, day month year (doi: 10.1038/sj.[JOURNAL ACRONYM].)

For Adaptations/Translations:
Adapted/Translated by permission from [**the Licensor**]: [**Journal Publisher** (e.g. Nature/Springer/Palgrave)] [**JOURNAL NAME**] [**REFERENCE CITATION** (Article name, Author(s) Name), [**COPYRIGHT**] (year of publication)

Note: For any republication from the British Journal of Cancer, the following credit line style applies:

Reprinted/adapted/translated by permission from [**the Licensor**]: on behalf of Cancer Research UK: : [**Journal Publisher** (e.g. Nature/Springer/Palgrave)] [**JOURNAL NAME**] [**REFERENCE CITATION** (Article name, Author(s) Name), [**COPYRIGHT]** (year of publication)

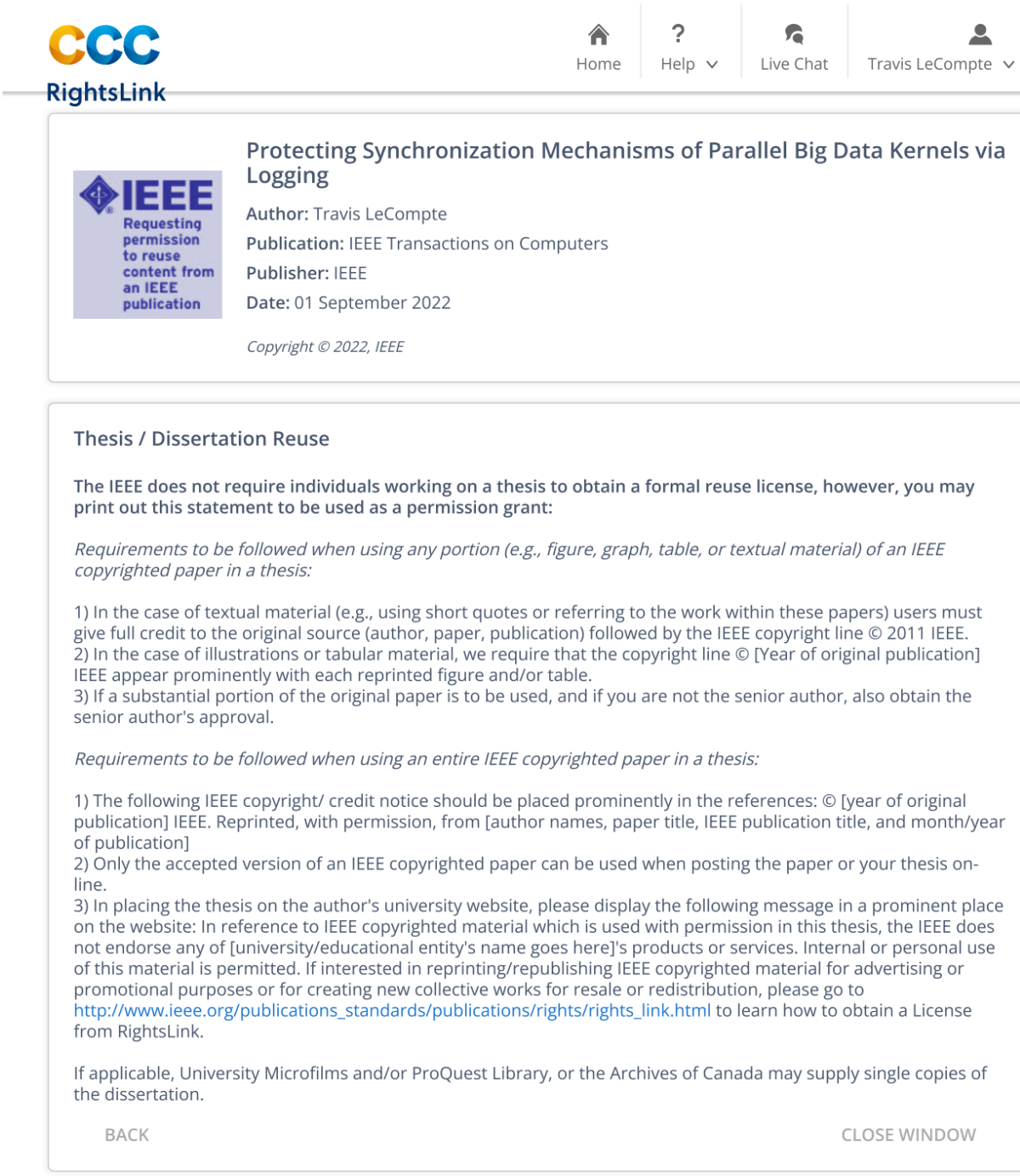
For **Advance Online Publication** papers:
Reprinted by permission from The [**the Licensor**]: on behalf of Cancer Research UK: [**Journal Publisher** (e.g. Nature/Springer/Palgrave)] [**JOURNAL NAME**] [**REFERENCE CITATION** (Article name, Author(s) Name), [**COPYRIGHT**] (year of publication), advance online publication, day month year (doi: 10.1038/sj.[JOURNAL ACRONYM])

For Book content:
Reprinted/adapted by permission from [**the Licensor**]: [**Book Publisher** (e.g. Palgrave Macmillan, Springer etc) [**Book Title**] by [**Book author**(s)] [**COPYRIGHT**] (year of publication)

Other Conditions:

Version 1.3

A.2. Publishing Agreement for Protecting Synchronization Mechanisms of Big Data Applications



RightsLink

Home | Help | Live Chat | Travis LeCompte

IEEE Requesting permission to reuse content from an IEEE publication

Protecting Synchronization Mechanisms of Parallel Big Data Kernels via Logging

Author: Travis LeCompte

Publication: IEEE Transactions on Computers

Publisher: IEEE

Date: 01 September 2022

Copyright © 2022, IEEE

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

BACK CLOSE WINDOW

A.3. Publishing Agreement for Robust Cache-Aware Quantum Processor Layout

CCC RightsLink

Home | Help | Live Chat | Sign in | Create Account

IEEE Requesting permission to reuse content from an IEEE publication

Robust Cache-Aware Quantum Processor Layout

Conference Proceedings:
2020 International Symposium on Reliable Distributed Systems (SRDS)

Author: Travis LeCompte

Publisher: IEEE

Date: September 2020

Copyright © 2020, IEEE

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

BACK CLOSE WINDOW

References

1. Agnesina, K. Chang, and S. K. Lim, “Vlsi placement parameter optimization using deep reinforcement learning,” in Proceedings of the 39th International Conference on Computer-Aided Design, 2020, pp. 1–9.

2. AMPLab at University of California, Berkeley (2014) AMPLab big data benchmark. https://amplab.cs.berkeley.edu/benchmark/

3. Armstrong TG, Ponnekanti V, Borthakur D, Callaghan M (2013) Linkbench: a database benchmark based on the Facebook social graph. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, SIGMOD ’13, pp 1185–1196. doi:10.1145/2463676.2465296

4. Arute, Frank, et al. "Quantum supremacy using a programmable superconducting processor." Nature 574.7779 (2019): 505-510.

5. Ash-Saki, M. Alam, and S. Ghosh, “Qure: Qubit re-allocation in noisy intermediate-scale quantum computers,” in Proceedings of the 56th Annual Design Automation Conference 2019, 2019, pp. 1–6

6. Austin, T. Diva: a reliable substrate for deep submicron microarchitecture design. In: Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO 1999).

7. Bausch, Johannes. "Recurrent Quantum Neural Networks." Advances in Neural Information Processing Systems 33 (2020).

8. Barnett, Thomas. “The Zettabyte Era Officially Begins (How Much Is That?)” *SP360: Service Provider, Cisco Blogs.* (2016) https://blogs.cisco.com/sp/the-zettabyte-era-officially-begins-how-much-is-that

9. Beer, Kerstin, et al. "Training deep quantum neural networks." Nature communications 11.1 (2020): 1-6.

10. Bender C, Sanda PN, Kudva P, Mata R, Pokala V, Haraden R, Schallhorn M (2008) Soft-error resilience of the ibm power6 processor input/output subsystem. IBM J Res Dev 52(3):285–292. doi:10.1147/rd.523.0285

11. Bernstein, Ethan, and Umesh Vazirani. "Quantum complexity theory." SIAM Journal on computing 26.5 (1997): 1411-1473.

12. BigDataBench Benchmark Suite. Available at https://www.benchcouncil.org/BigDataBench/index.html.

13. Cai, Ruizhe, et al. "A stochastic-computing based deep learning framework using adiabatic quantum-flux-parametron superconducting technology." 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2019.

14. Cappello, Franck, et al. "Toward exascale resilience: 2014 update." Supercomputing Frontiers and Innovations: an International Journal 1.1 (2014): 5-28.

15. Chen, Sui, et al. "Soft error resilience in Big Data kernels through modular analysis." The Journal of Supercomputing 72.4 (2016): 1570-1596.

16. Chen, K. Li, S. G. Teo, X. Zou, K. Wang, J. Wang, and Z. Zeng, "Gated residual recurrent graph neural networks for traffic prediction," in Proceedings of the AAAI conference on artificial intelligence, vol. 33, no. 01, 2019, pp. 485–492.

17. Chen, S., and Peng, L. "Efficient GPU hardware transactional memory through early conflict resolution." Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2016.

18. Chung J, Lee I, Sullivan M, Ryoo JH, Kim DW, Yoon DH, Kaplan L, Erez M (2012) "Containment domains: a scalable, efficient, and flexible resilience scheme for exascale systems". In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC12)

19. Clarke, John, and Frank K. Wilhelm. "Superconducting quantum bits." Nature 453.7198 (2008): 1031-1042.

20. Collange, Caroline, et al. "Numerical reproducibility for the parallel reduction on multi-and many-core architectures." Parallel Computing 49 (2015): 83-97. http://www.sciencedirect.com/science/article/pii/S0167819115001155

21. Cong, Iris, Soonwon Choi, and Mikhail D. Lukin. "Quantum convolutional neural networks." Nature Physics 15.12 (2019): 1273-1278.

22. Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R (2010) Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM Symposium on Cloud Computing, ACM, New York, NY, USA, SoCC '10, pp 143–154, doi:10.1145/1807128.1807152

23. Cross, Andrew W., et al. "Validating quantum computers using randomized model circuits." Physical Review A 100.3 (2019): 032328.

24. Cross, Andrew. "The IBM Q experience and QISKit open-source quantum computing software", APS Meeting Abstracts, 2018.

25. Deutsch, David, and Richard Jozsa. "Rapid solution of problems by quantum computation." Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences 439.1907 (1992): 553-558.

26. Dowling, Jonathan P. "On The Dowling-'Neven' Law." Quantum Pundit. 11 July 2019. http://quantumpundit.blogspot.com/2019/07/on-dowling-neven-law.html

27. Elliott, J., Kharbas, K., Fiala, D., Mueller, F., Ferreira, K., and Engelmann, C. ``Combining partial redundancy and checkpointing for HPC." Proceedings of the 2012 IEEE 32nd International Conference on Distributed Computing Systems. IEEE, 2012.

28. F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, “The graph neural network model,” IEEE transactions on neural networks, vol. 20, no. 1, pp. 61–80, 2008.

29. F. Wu, A. Souza, T. Zhang, C. Fifty, T. Yu, and K. Weinberger, “Simplifying graph convolutional networks,” in International conference on machine learning. PMLR, 2019, pp. 6861–6871.

30. Farhi, Edward, and Hartmut Neven. "Classification with quantum neural networks on near term processors." arXiv preprint arXiv:1802.06002 (2018).

31. Felber, P., Fetzer, C., Marlier, P., and Riegel, T. “Time-based software transactional memory." IEEE Transactions on Parallel and Distributed Systems 21.12 (2010): 1793-1807.

32. Ferdman M, Adileh A, Koçberber YO, Volos S, Alisafaee M, Jevdjic D, Kaynak C, Popescu AD,Ailamaki A, Falsafi B (2012) Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In: Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3–7, 2012, pp 37–48. doi:10.1145/2150976.2150982

33. Fu, Xiang, et al. "An experimental microarchitecture for a superconducting quantum processor." Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2017.

34. Fung, W. W., Singh, I., Brownsword, A., and Aamodt, T. M. “Hardware transactional memory for GPU architectures." Proceedings of the 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2011.

35. 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, 2019, pp. 1001–1014.

36. G. Zhang, H. He, and D. Katabi, “Circuit-gnn: Graph neural networks for distributed circuit design,” in International Conference on Machine Learning. PMLR, 2019, pp. 7364–7373.

37. Gabrys, Ryan, Eitan Yaakobi, and Lara Dolecek. “Graded bit-error-correcting codes with applications to flash memory." IEEE Transactions on Information Theory 59.4 (2012): 2315-2327.

38. Gao W, Luo C, Zhan J, Ye H, He X, Wang L, Zhu Y, Tian X (2015) Identifying dwarfs workloads in big data analytics. http://arxiv.org/abs/1505.06872

39. Ghazal A, Rabl T, Hu M, Raab F, Poess M, Crolotte A, Jacobsen HA (2013) Bigbench: towards an industry standard benchmark for big data analytics. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, SIGMOD ’13, pp 1197–1208. doi:10.1145/2463676.2463712

40. Gokhale, Pranav, et al. "Minimizing state preparations in variational quantum eigensolver by partitioning into commuting families." arXiv preprint arXiv:1907.13623 (2019).

41. Gokhale, Pranav, et al. "Partial compilation of variational algorithms for noisy intermediate-scale quantum machines." Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. 2019.

42. Grover, Lov K. "A fast quantum mechanical algorithm for database search." Proceedings of the twenty-eighth annual ACM symposium on theory of computing. 1996.

43. Guan Q, Debardeleben N, Blanchard S, Wu P, Monrow L, Chen Z (2016) P-FSEFI: a parallel soft error fault injection framework for parallel applications. In: Proceedings of the 12th Workshop on Silicon Error in Logic-System Effect (SELSE)

44. Harris M, NVidia (2015) https://devblogs.nvidia.com/parallelforall/new-features-cuda-7-5/

45. Holmes, M. R. Jokar, G. Pasandi, Y. Ding, M. Pedram, and F. T. Chong, "Nisq+: Boosting quantum computing power by approximating quantum error correction," in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020, pp. 556–569.

46. Holmes, Y. Ding, A. Javadi-Abhari, D. Franklin, M. Martonosi, and F. T. Chong, "Resource optimized quantum architectures for surface code implementations of magic-state distillation," Microprocessors and Microsystems, vol. 67, pp. 56–70, 2019.

47. Huang S, Huang J, Dai J, Xie T, Huang B (2010) The HiBench benchmark suite: characterization of the MapReduce-based data analysis. In: 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW), pp 41–51. doi:10.1109/ICDEW.2010.5452747

48. Iakymchuk R, Collagne S, Defour D, Graillat S (2015) Exblas: reproducible and accurate BLAS library. In the Proceedings of the Numerical Reproducibility at Exascale (NRE2015) workshop held as part of the Supercomputing Conference (SC15). Austin, TX, USA, November 15-20, 2015. HAL ID: hal-01202396

49. IBM Qiskit GAN Implementation. https://github.com/Qiskit/qiskit-aqua/blob/master/qiskit/aqua/algorithms/distribution_learners/qgan.py

50. IBM, "Open-source quantum development," https://qiskit.org/, retrieved on 04-16-2021.

51. Intel Corporation, "Pin 3.2 User Guide." Available at https://software.intel.com/sites/landingpage/pintool/docs/81205/Pin/html/.

52. Intel Pin Fault Injector (PINFI). Available at https://github.com/DependableSystemsLab/PINFI.

53. ITRS International technology roadmap for semiconductors. *ITRS Technical Report*. (2013)

54. J. Liu and H. Zhou, “Systematic approaches for precise and approximate quantum state runtime assertion,” in 27th IEEE International Symposium on High-Performance Computer Architecture, ser. HPCA, vol. 21, 2021.

55. J. Liu, G. T. Byrd, and H. Zhou, “Quantum circuits for dynamic runtime assertions in quantum computation,” in Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 1017–1030.

56. Jangjaimon, I. and Tzeng, N.-F. “Effective Cost Reduction for Elastic Clouds under Spot Instance Pricing through Adaptive Checkpointing," IEEE Transactions on Computers, vol. 64, no. 2, pp. 396-409, February 2015.

57. Javadi-Abhari, P. Gokhale, A. Holmes, D. Franklin, K. R. Brown, M. Martonosi, and F. T. Chong, “Optimized surface code communi- cation in superconducting quantum computers,” in Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, 2017, pp. 692–705.

58. Joshi, A., Nagarajan, V., Cintra, M., and Viglas, S. “DHTM: Durable hardware transactional memory." Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2018.

59. Kielpinski, David, Chris Monroe, and David J. Wineland. "Architecture for a large-scale ion-trap quantum computer." Nature 417.6890 (2002): 709-711.

60. Killoran, Nathan, et al. "Continuous-variable quantum neural networks." Physical Review Research 1.3 (2019): 033063.

61. Kübler, Jonas M., et al. "An adaptive optimizer for measurement-frugal variational algorithms." Quantum 4 (2020): 263.

62. Kumar S, Hari S, Adve SV, Naeimi H, Ramachandran P (2012) Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults. In: Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)

63. L. Gong and Q. Cheng, “Exploiting edge features for graph neural networks,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 9211–9219

64. LeCompte, T. et al. “Soft Error Resilience of Big Data Kernels through Algorithmic Approaches.” Springer Journal of Supercomputing. Vol 73, pp. 4739-4772. Nov 2017.

65. T. LeCompte, L. Peng, X. Yuan and N. -F. Tzeng, "Protecting Synchronization Mechanisms of Parallel Big Data Kernels via Logging," in IEEE Transactions on Computers, vol. 71, no. 9, pp. 2156-2162, 1 Sept. 2022, doi: 10.1109/TC.2021.3122993.

66. LeCompte, T., Qi, F., and Peng, L. “Robust Cache-Aware Quantum Processor Layout,” In Proceedings of the 39th IEEE International Symposium on Reliable Distributed Systems (SRDS), Shanghai, China, Sep. 2020.

67. Li, H., Chen, Z.m and Gupta, R. “Parastack: Efficient hang detection for MPI programs at large scale." Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2017.

68. Liu W, Zhang W, Wang X, Xu J (2016) Distributed sensor network-on-chip for performance optimization of soft-error-tolerant multiprocessor system-on-chip. IEEE Trans Very Large Scale Integr (VLSI) Syst 24(4):1546–1559. doi:10.1109/TVLSI.2015.2452910

69. Lloyd, Seth, Masoud Mohseni, and Patrick Rebentrost. "Quantum algorithms for supervised and unsupervised machine learning." arXiv preprint arXiv:1307.0411 (2013).

70. 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, 2018, pp. 113–125

71. Mari, Andrea, et al. "Transfer learning in hybrid classical-quantum neural networks." Quantum 4 (2020): 340.

72. Meitei, Oinam Romesh, et al. "Gate-free state preparation for fast variational quantum eigensolver simulations: ctrl-VQE." arXiv preprint arXiv:2008.04302 (2020).

73. NVIDIA Tesla k20 gpu accelerator. (2013) http://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v07.pdf

74. P. Murali, D. C. McKay, M. Martonosi, and A. Javadi-Abhari, “Software mitigation of crosstalk on noisy intermediate-scale quantum computers,” in Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 1001–1016

75. 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, 2019, pp. 1015–1029.

76. P. Zhu, X. Cheng, and Z. Guan, “An exact qubit allocation approach for nisq architectures,” Quantum Information Processing, vol. 19, no. 11, pp. 1–21, 2020

77. Perozzi, R. Al-Rfou, and S. Skiena, “Deepwalk: Online learning of social representations,” in Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining, 2014, pp. 701–710.

78. Qi, Fang, et al. "Quantum Vulnerability Analysis to Accurate Estimate the Quantum Algorithm Success Rate." arXiv preprint arXiv:2207.14446 (2022).

79. Quantum Fourier Transform. 27 July 2020, qiskit.org/textbook/ch-algorithms/quantum-fourier-transform.html

80. R. Barends, J. Kelly, A. Megrant, A. Veitia, D. Sank, E. Jeffrey, T. C. White, J. Mutus, A. G. Fowler, B. Campbell et al., "Superconducting quantum circuits at the surface code threshold for fault tolerance," Nature, vol. 508, no. 7497, pp. 500–503, 2014.

81. S. Pan, R. Hu, G. Long, J. Jiang, L. Yao, and C. Zhang, "Adversarially regularized graph autoencoder for graph embedding," arXiv preprint arXiv:1802.04407, 2018.

82. 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, 2019, pp. 987–999

83. S. S. Tannu and M. Qureshi, "Ensemble of diverse mappings: Improving reliability of quantum computers by orchestrating dissimilar mistakes," in Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019, pp. 253–265.

84. Serrano F, Clemente JA, Mecha H (2015) A methodology to emulate single event upsets in flip-flops using FPGAs through partial reconfiguration and instrumentation. IEEE Trans Nucl Sci 62(4):1617–1624. doi:10.1109/TNS.2015.2447391

85. Sharma, Kunal, et al. "Noise resilience of variational quantum compiling." New Journal of Physics 22.4 (2020): 043006.

86. Shor, Peter W. "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer." SIAM review 41.2 (1999): 303-332.

87. Sigdel, P. and Tzeng, N.-F. "Coalescing and Deduplicating Incremental Checkpoint Files for Restore-Express Multi-Level Checkpointing," IEEE Transactions on Parallel and Distributed Systems, vol. 29, no. 12, pp. 2713-2727, December 2018.

88. Simon, Daniel R. "On the power of quantum computation." SIAM journal on computing 26.5 (1997): 1474-1483.

89. T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," arXiv preprint arXiv:1609.02907, 2016.

90. Thaker, Darshan D., et al. "Quantum memory hierarchies: Efficient designs to match available parallelism in quantum computing." ACM SIGARCH Computer Architecture News 34.2 (2006): 378-390.

91. Thekumparampil, Kiran K., et al. "Attention-based graph neural network for semi-supervised learning." arXiv preprint arXiv:1803.03735 (2018).

92. Thomas, T. E., Bhattad, A. J., Mitra, S., and Bagchi, S. "Sirius: Neural network based probabilistic assertions for detecting silent data corruption in parallel programs." Proceedings of the 2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS). IEEE, 2016.

93. Tiwari D, Gupta S, Gallarno G, Rogers J, Maxwell D (2015) Reliability lessons learned from GPU experience with the titan supercomputer at oak ridge leadership computing facility. In: SC15: International Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–12. doi:10.1145/2807591.2807666).

94. Volos, H., Tack, A. J., Swift, M. M., and Lu, S. “Applying transactional memory to concurrency bugs." ACM SIGPLAN Notices 47.4 (2012): 211-222.

95. W. Finigan, M. Cubeddu, T. Lively, J. Flick, and P. Narang, “Qubit allocation for noisy intermediate-scale quantum computers,” arXiv preprint arXiv:1810.08291, 2018.

96. W. L. Hamilton, R. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” in Proceedings of the 31st International Conference on Neural Information Processing Systems, 2017, pp. 1025–1035.

97. Wang L, Bertran R, Buyuktosunoglu A, Bose P, Skadron K (2014) Characterization of transient error tolerance for a class of mobile embedded applications. In: 2014 IEEE International Symposium on Workload Characterization (IISWC), pp 74–75. doi:10.1109/IISWC.2014.6983042

98. Wang, Samson, et al. "Noise-induced barren plateaus in variational quantum algorithms." arXiv preprint arXiv:2007.14384 (2020).

99. Wang, Y., Um, M., Zhang, J. et al. Single-qubit quantum memory exceeding ten-minute coherence time. Nature Photon 11, 646–650 (2017). https://doi.org/10.1038/s41566-017-0007-1

100. Xiaoguang, R., Xinhai, X., Qian, W., Juan, C., Miao, W., and Xuejun, Y. “GS-DMR: Low-overhead soft error detection scheme for stencil-based computation." Parallel Computing 41 (2015): 50-65.

101. Y. Ding, P. Gokhale, S. F. Lin, R. Rines, T. Propson, and F. T. Chong, “Systematic crosstalk mitigation for superconducting qubits via frequency-aware compilation,” in 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2020, pp. 201–214.

102. Y. Ding, X.-C. Wu, A. Holmes, A. Wiseth, D. Franklin, M. Martonosi, and F. T. Chong, “Square: strategic quantum ancilla reuse for modular quantum programs via cost-effective uncomputation,” in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020, pp. 570–583.

103. Y. Huang and M. Martonosi, “Statistical assertions for validating patterns and finding bugs in quantum programs,” in Proceedings of the 46th International Symposium on Computer Architecture, 2019, pp. 541–553.

104. Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, “Gated graph sequence neural networks,” arXiv preprint arXiv:1511.05493, 2015.

105. Y.-C. Lu, S. Pentapati, and S. K. Lim, “Vlsi placement optimization using graph neural networks,” in 34th Conference on Neural Information Processing Systems (NeurIPS 2020), ML for Systems Workshop, 2020.

106. Yeh TY, Reinman G, Patel SJ, Faloutsos P (2009) Fool me twice: exploring and exploiting error tolerance in physics-based animation. ACM Trans Graph 29(1):5:1–5:11. doi:10.1145/1640443.1640448

Vita

Travis LeCompte was born in 1995 in Thibodaux, Louisiana. He received his Bachelor of Science degree in Computer Science and his Bachelor of Science degree in Electrical Engineering from Louisiana State University in Baton Rouge, Louisiana, in May 2017. Since then, he has been enrolled in the Division of Electrical & Computer Engineering at Louisiana State University to purse his Ph.D. degree. During this time, he has passed his qualifying exam in Spring 2019 and his general exam in Spring 2022.

Travis's research interests include software resilience, quantum computing, and machine learning. He has published several papers on these topics in various conferences and journals.