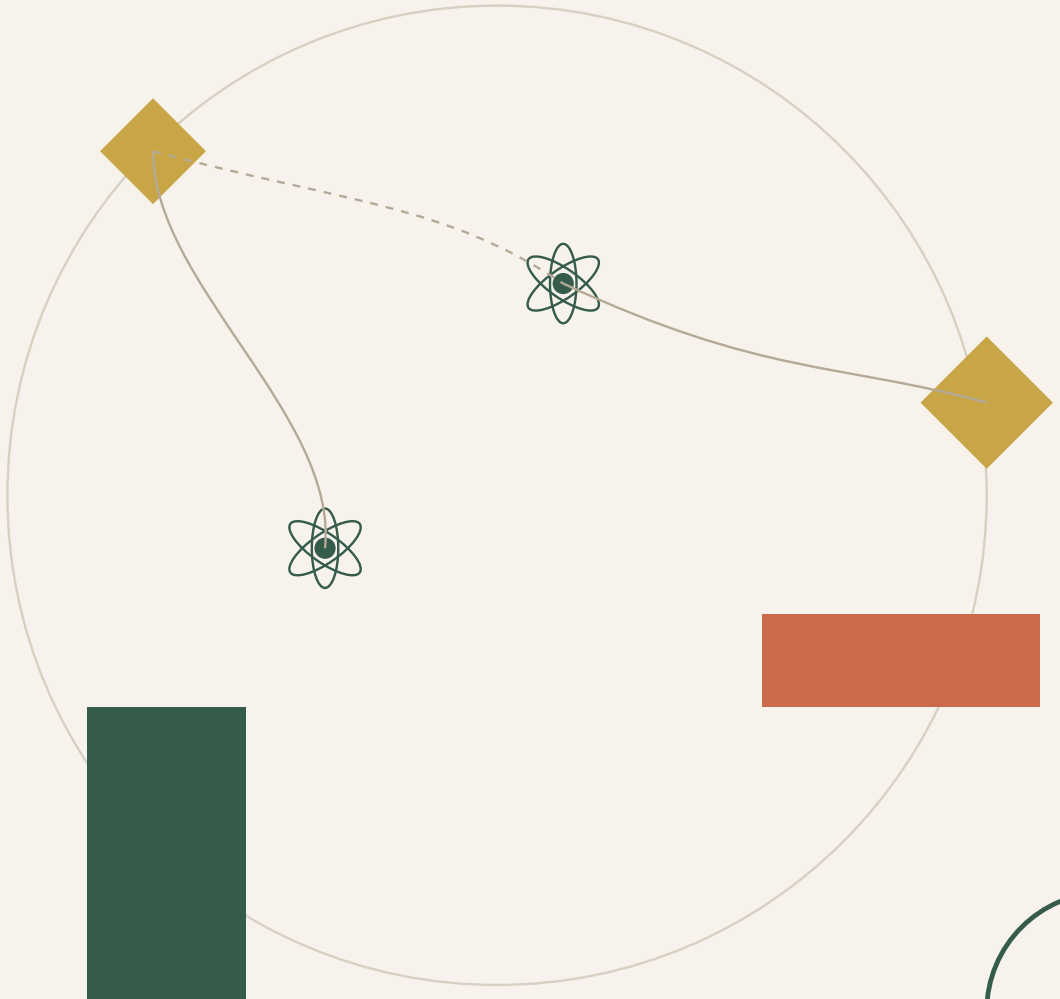
Solving Deadlocks in Quantum Network Nodes: A Comparison of Opposing Strategies

Master's Thesis



Carl Johan Gützkow

QuTech, Delft University of Technology

Solving Deadlocks in Quantum Network Nodes: A Comparison of Opposing Strategies

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Carl Johan Gützkow
born in Oslo, Norway

Faculty Electrical Engineering, Mathematics and
Computer Science
QuTech, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Wehner Group, QuTech
Delft, the Netherlands
www.qutech.nl

Solving Deadlocks in Quantum Network Nodes: A Comparison of Opposing Strategies

Author: Carl Johan Gützkow
Student id: 6294510
Email: c.j.gutzkow@student.tudelft.nl

Abstract

Quantum networks provide functionality not found in classical networks and realizing their potential requires nodes to execute quantum applications reliably. Qoala, an execution environment for hybrid quantum-classical applications, runs several program instances concurrently on a single node, where they contend for a limited pool of qubits. Because qubits are held with exclusive access and cannot be preempted into classical storage without destroying their state, the conditions for deadlock arise, possibly leaving programs unable to proceed. In this thesis, we implement and compare the three established approaches to the deadlock problem, detection with recovery, prevention, and avoidance, in Qoala. Waiting carries costs beyond execution time, because qubits in memory decohere. A blocked program is less likely to succeed, and terminating a program to free up resources discards entanglement that is slow to generate. We evaluate the strategies on classical and quantum metrics across workloads, study how they scale across various configuration, and introduce a Deadlock Impact Score whose coefficients tune recovery to prioritize either runtime or qubit decoherence during termination. We find that, with a network schedule, success probabilities are largely similar across strategies, typically within about one percentage point, while avoidance consistently achieves the lowest makespan. Without a network schedule, success probabilities vary more, by up to 13 percentage points, and prevention performs best, with a makespan comparable to avoidance. This ordering was broadly stable across the network configurations we tested, though not without exceptions. Our work enables concurrent execution of workloads in Qoala that previously had to run sequentially and opens the door to further research on concurrency in quantum network nodes.

Thesis Committee:

Chair:	Prof. Dr. S.D.C. Wehner, Faculty EEMCS, TU Delft
University supervisor:	S. Oslovich, Faculty EEMCS, TU Delft
Committee Member:	Dr. T.J. Coopmans, Faculty EEMCS, TU Delft
Committee Member:	Dr. S. Chakraborty, Faculty EEMCS, TU Delft

Preface

I am very grateful for all the support I have had during this thesis. First, I want to thank my supervisor Samuel Oslovich for all the discussions, support, and helpful feedback. Making me understand the depth of quantum computing from a computer science background has been very helpful for my project. Next, I want to give my gratitude to Stephanie Wehner for letting me write my thesis with you. This great experience would not have happened without your educational videos about the quantum internet that I watched several years ago. Thank you for your feedback. Being part of Wehner research group has given me a lot of insight into the field of quantum networks. I also want to thank my thesis committee members Dr. Soham Chakraborty and Dr. Tim Coopmans for reading my thesis, taking part in my defense and evaluating my work. Thank you to my friends for continued interest and support and really trying to understand what I have worked on. It has been amazing having people around me throughout this thesis to meet for our Fried Fridays, BBQs, volleyball, biking, and late study sessions. I also want to thank my family. Your unconditional support is invaluable and I am forever grateful that you have pushed me to become the best version of myself. To my girlfriend, thank you for always being there for me, listening to me talk about my work, making it clear when it became too difficult to understand, and still supporting me even when you were equally stressed. Thank you all for being part of my thesis.

Carl Johan Gützkow
Delft, the Netherlands
June 28, 2026

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

A quantum internet would let remote parties share and operate on quantum states alongside the data they exchange over the classical internet [38]. It relies on entanglement, a property of quantum systems allowing communication not previously possible. This enables protocols with no classical counterpart, such as quantum key distribution (QKD) for enhanced secure communication [29] and blind quantum computation (BQC) to perform hidden operations on a server [9]. Realizing these protocols requires more than just generating entanglement. Every node in a network must execute its quantum programs to completion reliably. This reliability is also hindered by decoherence, which reduces the likelihood of successful execution. Reliable program execution on each node is therefore as much a part of a quantum internet as the entanglement that links the nodes together.

Delivering this kind of reliability has motivated a software stack for quantum network nodes. QNodeOS [16] and its extension Qoala [36] allow a node to execute classical-quantum hybrid programs that communicate with remote nodes. Similarly to a classical operating system, several program instances can run concurrently on a node.

Concurrent execution on a single node, however, makes the programs compete over available qubits. A program instance acquires virtual qubits through a memory manager and holds them with exclusive access as illustrated by Figure 1.1. While holding on to a qubit, a program instance can continue to request more qubits. Furthermore, when a qubit is in memory, we cannot preempt the resource to temporarily store its state in classical storage or other mediums and continue operations with it afterward. This is in contrast to classical memory that can temporarily be stored on disk which has more available space. Retrieving qubit state entails measuring the qubit, which collapses the quantum state. A held qubit also loses information over time through decoherence, so any wait imposed by competition shortens the window in which the qubit is still usable. The four conditions for deadlock (mutual exclusion, hold-and-wait, no preemption, and circular wait) can therefore all be satisfied on a quantum network node, and two of them (mutual exclusion and no preemption) are inherent to the platform. When several instances incrementally request qubits in a system with limited resources, they can reach a state where none can proceed. This is the deadlock problem. A network of nodes that are exposed to deadlocks is an unreliable one.

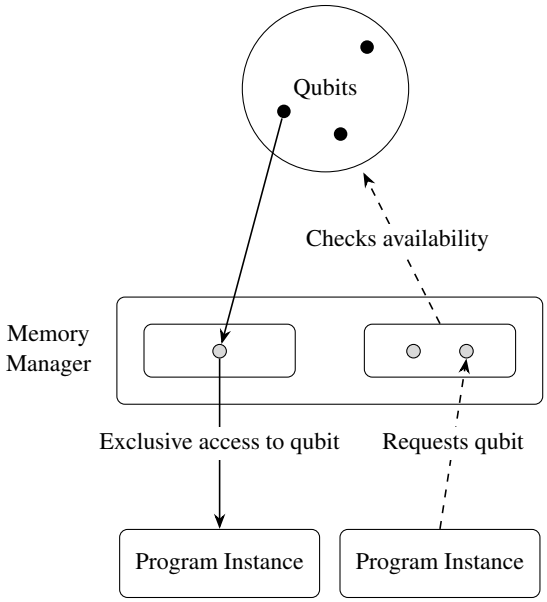


Figure 1.1: Program instances request virtual qubits through the memory manager that maps to physical qubits. Dashed arrows represent requests from a program instance and solid arrows represent allocations.

There are multiple ways to tackle the deadlock problem. They can be grouped into three different strategies excluding the trivial solution of ignoring it. First, detection of deadlock can be used in combination with termination or recovery to release resources and continue running. Second, prevention removes a deadlock condition, for example by allocating all resources at once to eliminate hold-and-wait. Lastly, avoidance strategies guarantee that all processes complete by withholding requests that might lead to deadlock [23].

Applying these strategies on a quantum network node differs from the classical case in a few practical ways. Remote entanglement generation is the slowest step in current quantum networks, on the order of milliseconds to seconds, whereas local classical tasks finish in nanoseconds and classical messages arrive in milliseconds [16, 36]. Waiting itself is not free. Qubits in memory are continuously exposed to decoherence, so the longer a program instance is blocked, the more likely its measurements are to return incorrect results. Any strategy that breaks a deadlock by terminating a program therefore discards entanglement that the network spent time generating, while any strategy that lets a program keep waiting pays for that wait with unsuccessful execution.

Implementing deadlock handling strategies has been done in many different domains like real-time systems [28], distributed databases [23], and manufacturing systems [25]. Each domain makes its own assumptions, for example the number of available resources or the degree to which program behavior is known before runtime, and the right strategy depends on which assumptions hold. Deadlock handling has not been studied in the setting of quantum network nodes, where the cost of waiting is measured in the degrading state of qubits

and not just completion time.

Programs on quantum network nodes mix classical and quantum instructions in several ways, and testing every permutation is infeasible. The evaluation therefore uses a representative set drawn from existing applications (e.g., BQC) together with synthetic workloads designed to expose deadlock behavior.

We evaluate each strategy on classical and quantum performance metrics. Each workload can be impacted differently by each deadlock handling strategy depending on factors such as the frequency of deadlocks, which motivates the first research question.

> **RQ1.** How do deadlock handling strategies affect classical and quantum performance metrics across low- and high-contention workloads?

Future applications will need more qubits and improved hardware, which raises the question of how these strategies scale.

> **RQ2.** How does the performance of deadlock handling strategies scale with program size, qubit topology and network configuration?

We pair deadlock detection with a recovery step that terminates a program instance and releases its resources. A heuristic is needed to determine which program instances to terminate [12]. The right choice of heuristic depends on what the system should optimize for.

> **RQ3.** How do different victim-selection heuristics for terminating deadlocked program instances affect the combined performance of running programs?

This thesis has the following contributions. (1) We implement deadlock detection combined with a recovery step that releases resources, prevention by allocating all qubits at program instantiation, and avoidance through requests that keep the system in a safe state. (2) We define methods to determine which program instance to terminate to release resources. (3) We formalize classical and quantum evaluation metrics for deadlock detection, prevention, and avoidance. (4) We compare the performance of the proposed algorithms against baseline configurations without dedicated deadlock handling.

The thesis is structured as follows. Chapter 2 gives background of the intrinsic properties of quantum network nodes, the literature of deadlocks, and the research gap this thesis is aimed at. Chapter 3 describes the deadlock-handling algorithms used (detection with recovery, prevention, and avoidance) and how the deadlock conditions map to a quantum network node. Chapter 4 explains how those algorithms are integrated into the Qoala architecture. Chapter 5 introduces evaluation methods and employs them to produce comparable results. Chapter 6 and Chapter 7 present and discuss the results from the evaluation. Chapter 8 provides future research directions and summarizes the thesis.

Chapter 2

Background

This chapter introduces the concepts that this thesis builds on. We first cover the foundations of quantum computing, quantum networks, and deadlocks. We then survey related work that addresses similar problems in other domains.

2.1 Quantum Computing

The theory of quantum computation and quantum information is well established, even though practical implementations remain in their early stages. This section reviews the fundamentals of quantum computing following [30], together with more recent results from the quantum networking literature.

2.1.1 Qubits

Quantum computing centers on manipulating quantum bits, or qubits, to hold state. A classical bit holds the value 0 or 1, whereas a qubit can be in a superposition of the two. A qubit in superposition is in a linear combination of the basis states $\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$ ($|0\rangle$) and $\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$ ($|1\rangle$). The state of a qubit cannot be inspected directly, and a measurement collapses it to either 0 or 1, irreversibly destroying any superposition it held. A common analogy is a coin. While the coin lies flat it is either heads or tails, but while it is spinning the outcome is not yet defined and only takes a value once it comes to rest. As a consequence, the state of a qubit cannot be stored in classical memory as a backup, since doing so would require reading it out and reading collapses it.

A qubit can be represented as a point on the surface of the Bloch sphere, as shown in Figure 2.1. While the poles correspond to $|0\rangle$ and $|1\rangle$, the points where the x- and y-axes meet the surface represent equal superpositions that differ only in their relative phase: $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ lie on the x-axis, while $|i\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$ and $|-i\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle)$ lie on the y-axis.

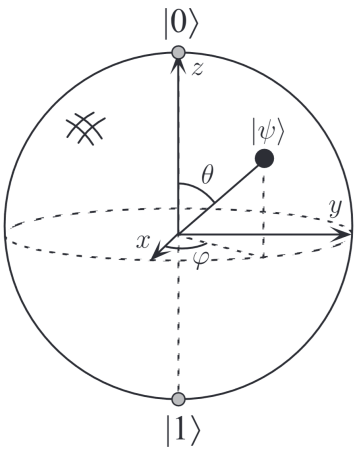


Figure 2.1: The Bloch sphere representation of a qubit from [30]. The state of the qubit lies on its surface. The north and south poles represent the states $|0\rangle$ and $|1\rangle$ respectively.

2.1.2 Noise

A qubit is susceptible to noise from its environment. Common sources include imperfect operations applied to the qubit, amplitude damping, and phase damping.

Amplitude damping, characterized by the time constant T1, describes the loss of energy from a qubit. The more energy is lost, the closer the qubit moves toward the $|0\rangle$ state. Phase damping, also called dephasing and characterized by the time constant T2, describes the loss of information without an accompanying loss of energy. As a qubit dephases, the x and y components of its state become less defined, while states near the poles are largely preserved. Information about the position of the qubit on the equator of the Bloch sphere is therefore lost. It can be shown that $T_2 \leq T_1/2$, but their exact values are dependent on the hardware. Figure 2.2 illustrates how the qubit state changes under each process.

The effect of noise on a qubit can be quantified by its *fidelity*, a measure of how close the actual state is to a desired *pure* state. Fidelity ranges from 0.0 to 1.0, where 1.0 indicates a perfect match to a pure state that has not been affected by noise. Imperfect quantum gates applied to the qubit reduce fidelity in the same way.

2.1.3 Gates

A quantum gate is an operation that manipulates the state of one or more qubits. Single-qubit gates such as the `NOT` gate interchange the coefficients of $|0\rangle$ and $|1\rangle$, while multi-qubit gates such as `CNOT` apply such an operation conditionally on another qubit. Each gate is represented as a matrix, and single-qubit gates correspond to rotations on the Bloch sphere. Gates can be composed to form more complex operations, some analogous to classical logic gates and others unique to quantum computing.

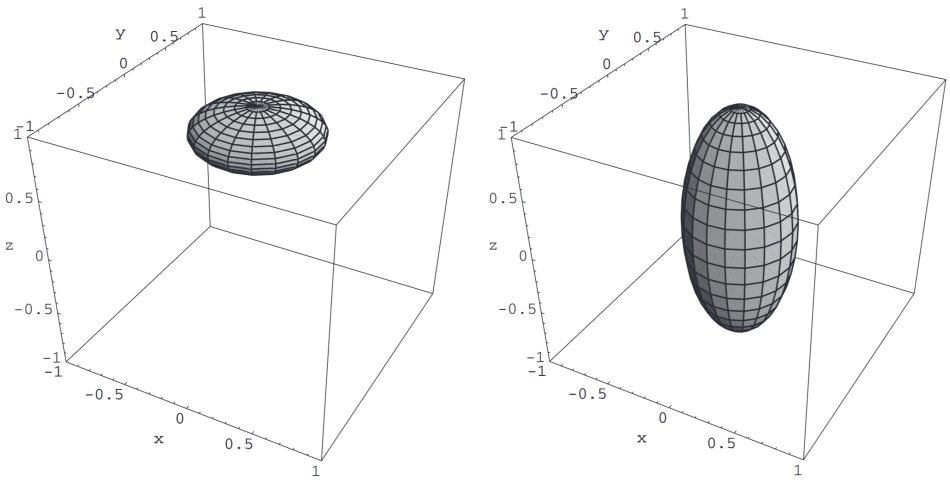


Figure 2.2: Bloch sphere representations of amplitude damping (left) and phase damping (right) from [30]. Amplitude damping moves states towards $|0\rangle$, while phase damping reduces the x and y-axis components of the state.

2.1.4 Entanglement

Another property that distinguishes quantum computing from classical computing is entanglement. When a pair of qubits is entangled, their measurement outcomes are correlated more strongly than is possible in any classical system. A canonical example is the Bell state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, in which measuring either qubit in the computational basis fully determines the outcome of the other. Such a pair is commonly called an EPR pair after Einstein, Podolsky, and Rosen, and we use the two terms interchangeably in this thesis. Entanglement plays a central role in quantum networks, as discussed in Section 2.2.

2.1.5 Memory Topologies

Qubits are divided into memory and communication qubits based on the memory topology of the underlying hardware. Nitrogen-vacancy centers in diamond (NV) [8], for example, have a star topology with a single communication qubit at the center and multiple memory qubits around it. Trapped ions, by contrast, can have a fully connected topology in which every ion in the same *trap* can interact with every other ion. Some of the ions are designated as communication qubits for remote entanglement generation, while the remaining ions serve as memory qubits that store quantum information [22, 10].

The topology determines which pairs of qubits can interact directly, since a two-qubit gate can only be applied between connected qubits. Only communication qubits can take part in remote entanglement generation, so generating entanglement repeatedly requires states to be moved onto memory qubits using a `SWAP` gate, a two-qubit gate that exchanges the states of two qubits in place. In quantum networks in particular, entanglement is generated frequently in order to establish connections and exchange information between nodes.

2.2 Quantum Internet

The quantum internet enables new forms of information exchange that have no classical counterpart [38]. Research has produced a range of applications, including enhanced secure communication [6] and secure hidden computation on remote quantum servers [11, 9].

These applications use both entanglement and communication over the classical internet to exchange information between nodes. To realize them, the quantum computers serving as nodes must reliably store quantum states between entanglement attempts [2]. Running such applications at scale also requires a software layer that abstracts over the underlying control electronics and exposes a programmable interface to applications. QNodeOS is one such layer, a quantum operating system for nodes in a quantum network that has been demonstrated on hardware by Delle Donne et al. [16].

In the quantum network proposed by Dahlberg et al. [13], entanglement is created at a heralding station between two nodes. Generating an entangled pair requires repeated attempts and completes on the order of milliseconds to seconds. Classical messages exchanged between nodes arrive on the order of milliseconds, while local classical processing and local quantum routines execute on the order of nanoseconds to microseconds. Structured time bins specify when entanglement can be generated to meet the demands of the network [18]. Each bin marks a window during which a particular remote entanglement attempt may begin between two program instances and nodes. If it fails, the nodes have to wait for the next time bin. See Figure 2.3 for an illustration of how time bins structure entanglements between programs.

This is the quantum network stack that Qoala is designed for [36].

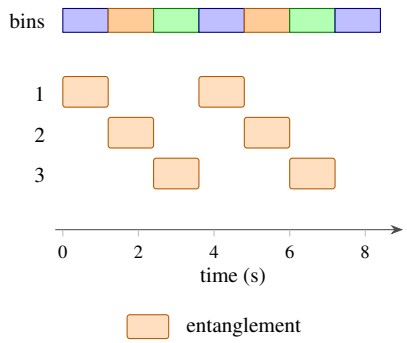


Figure 2.3: Time bins for entanglement for BQC. Local quantum or classical operations are not visible because their time scale compared to entanglement is miniscule.

2.3 Qoala

Qoala [36] extends QNodeOS with a unified program format and runtime support for hybrid classical-quantum programs. The runtime is realized as the Qoala Simulator, and programs are written as `.iqoala` files that are parsed and dispatched to the simulator [36]. This

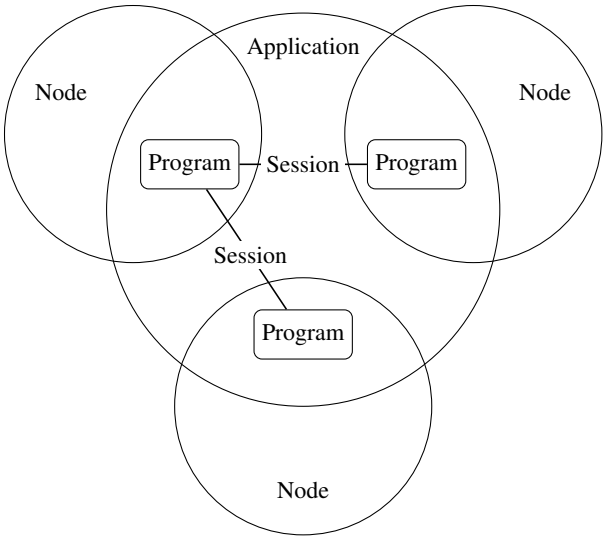


Figure 2.4: A distributed application is a collection of programs, one per participating node. Programs communicate with each other through sessions over entanglement and classical messages.

section gives a detailed overview of the Qoala runtime. We describe the structure of applications, how resources are managed between them, and how programs are scheduled.

2.3.1 Applications and Programs

Each node in the network runs its own Qoala instance, which locally schedules the program instances hosted on that node. A distributed application consists of a collection of programs, one per participating node, whose instances communicate with each other through entanglement and classical messages (Figure 2.4).

A Qoala program is a sequence of quantum communication (QC), quantum local (QL), classical communication (CC), and classical local (CL) blocks. These blocks are translated into a task graph that the runtime executes.

2.3.2 The Task Graph

The node scheduler in Qoala maintains a task graph of tasks that are scheduled to run. A program's converted tasks can either be *predictable* if the entire task graph is known before execution or *unpredictable* when each block is converted to tasks on the fly after a jump instruction between blocks. This distinction separates programs that follow a linear execution trail versus those that use conditionals and loops between blocks.

A quantum task, either a local routine (QL) consisting of gates and measurements or a request routine (QC) that initiates entanglement generation with a remote node, can request access to qubits in order to measure them or to keep them for a future task [36]. If the

corresponding program already has access to the qubit, no new request is needed. Host local (CL) and host event (CC) tasks, on the other hand, do not require access to qubits.

These tasks are divided between the schedulers for the quantum processing system (QPS) and the classical processing system (CPS).

2.3.3 Scheduling

Qoala uses two processor schedulers to prioritize tasks, the classical processor scheduler and the quantum processor scheduler. They schedule tasks for the central processing unit (CPU) and the quantum processing unit (QPU) respectively. The node scheduler forwards newly ready tasks to these schedulers, which then decide which task to run next and remove tasks once they have completed. The node scheduler and the two processor schedulers communicate over typed message channels, and the node scheduler is the only component that submits new tasks or removes existing ones from a processor scheduler. A task can also carry precedence constraints that require another task to complete before it can run, either internal to the same processor scheduler or external from the other.

Whenever a ready task is available, the processor schedulers select one such task to execute. A task is considered ready when the following conditions hold [36].

- It has no internal or external precedence constraints.
- If it is an entanglement request routine, the current time is at the beginning of the corresponding network time bin.
- If it is a host event task (classical communication), the message buffer contains at least one message from a remote node.

Among the executable tasks, the processor schedulers can prioritize using first-come-first-serve (FCFS), earliest-deadline-first (EDF), or a non-deterministic policy that picks at random from the ready tasks. Qoala borrows the concept of soft deadlines from real-time operating systems. Under EDF, the programmer can attach optional relative deadlines between blocks, which the scheduler can then use to reduce the time qubits are left to decohere. When such deadlines exist, the task with the shortest deadline is prioritized. Qoala does not enforce strict deadline guarantees, and a deadline only takes effect once the program has been annotated with one. In addition to deadlines, programs can also define critical sections. Once a critical section is entered, only tasks belonging to the same critical section may execute.

Entanglement is generated through the QPS of a node, while classical messages between nodes are sent through the CPS. Figure 2.5 illustrates how each part of the node communicates with a remote node.

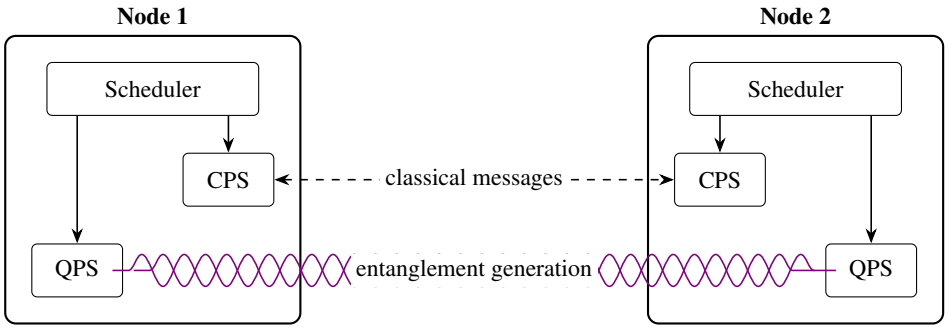


Figure 2.5: Two Qoala nodes communicate over classical messages exchanged between their CPS and entanglement generated through their QPS. The scheduler controls the CPS and QPS on each node.

2.3.4 Resource Management

Through their tasks, program instances acquire exclusive access to qubits. No other program instance can apply operations to these qubits. The memory manager tracks which program has allocated which qubits using a per-program virtual memory abstraction, analogous to virtual memory in classical operating systems, that maps the virtual qubits a program interacts with onto the physical qubits of the node.

A task does not request a specific physical qubit. It requests a virtual qubit of a given type, either a communication qubit or a memory qubit, and the memory manager allocates a specific physical qubit of that type. When a task releases a qubit, either explicitly or by measuring it, the memory manager deallocates the corresponding physical qubit and updates the mapping accordingly.

2.4 Deadlock

Before turning to how deadlocks arise in Qoala's setting, we first introduce the deadlock problem in general terms. Deadlocks have been studied extensively across several fields for decades. A deadlock occurs when a set of processes cannot proceed because each is waiting on another to take action, blocking one another in turn. Resolving the situation requires an external force [19] to intervene.

A deadlock *can* occur when the following four conditions are simultaneously met.

- **Hold-and-wait**: A process holds at least one resource and is waiting to acquire additional ones.
- **No preemption**: A resource cannot be taken from a process and must be released by its owner.
- **Mutual exclusion**: Resources cannot be shared between processes.

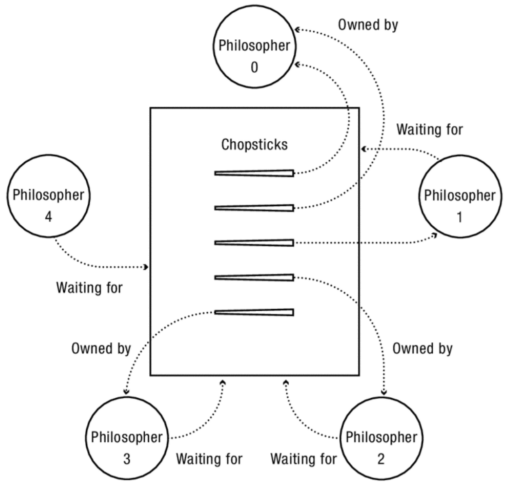


Figure 2.6: Dining Philosophers with 5 total chopsticks the philosophers can pick up. From [3]

- **Circular wait**: A cycle of waiting dependencies exists in the resource allocation graph.

A classic example of a deadlock is the Dining Philosophers (Figure 2.6). Five philosophers sit around a dinner table. Each needs two chopsticks to eat, but only five chopsticks are available. A philosopher can pick up only one chopstick at a time, and the five chopsticks are sufficient only if the philosophers take turns. Suppose each philosopher picks up one chopstick at the same moment. If none is willing to put theirs back down, the system has reached a deadlock. In this deadlock, (1) each philosopher holds one chopstick while waiting for another, (2) none will release theirs to let the others continue, (3) a chopstick cannot be shared, and (4) a cycle can be traced through each philosopher and the chopsticks they are waiting for [3].

A resource allocation graph (RAG) is used to visually represent the pending requests and allocations of a system. Reusable resources, which are the relevant type for this thesis, are assumed to be held with mutual exclusivity. An allocation is depicted as an arrow from the resource to the process, and a request as an arrow from the process to the resource or resource type. Resources can also be either single-unit or multi-unit. In a multi-unit system, resources are grouped by type, and a process requests a resource of a given type and is allocated a specific unit of that type [21]. The Dining Philosophers (Figure 2.6) is an example of a multi-unit system, but can also be represented as a single-unit system (see [3]).

There are many strategies for handling deadlocks, each based on different assumptions about the underlying system. The optimal strategy can for example depend on the chances of deadlock or the recoverability from one.

2.4.1 Detection

Deadlock detection strategies identify a deadlock only after it has occurred [12]. A detection algorithm is run at chosen points in time to determine whether the system has entered a deadlocked state.

In single-unit systems, a cycle in the RAG is a sufficient condition for deadlock. In multi-unit systems, which will be considered for the rest of this thesis, a knot is the sufficient condition while a cycle is necessary but not sufficient [21].

Coffman et al. [12] describe a detection procedure for reusable resources that does not rely on the resource graph directly. The procedure checks whether the currently allocated resources allow every process to run to completion in some order. If no such order exists, the system is deadlocked.

Once a deadlock has been detected, the system has to resolve it. It must first be decided which process or processes should release their resources. This decision can be guided by factors such as how many resources a process has acquired and how long it has been running or the cost of re-acquiring some resource type [12]. Terminating younger processes can shorten the overall time to completion. This also reduces the risk of starvation, since older processes are allowed to continue. Selected victims are then terminated and may resume from the most recent checkpoint, or, if no checkpoint exists, restart from the beginning.

2.4.2 Prevention

Prevention algorithms aim to eliminate one of the four necessary conditions for deadlock. One prevention method requires all resources a program will need during its lifetime to be acquired before execution begins. This can lead to poor utilization but eliminates any hold-and-wait situation [23].

Another method imposes an ordering on the resource types and requires processes to acquire resources in that order. For example, a process that needs both type A and type B must acquire all resources it might need of type A before requesting any of type B. This eliminates the circular wait condition.

2.4.3 Avoidance

Avoidance techniques aim to keep the system on a safe execution path while resources are dynamically allocated as processes request them. One such algorithm is the Banker's algorithm [14, 15].

The Banker's algorithm temporarily allocates the requested resources to the requesting process and then runs a safety check to verify that a safe state is preserved. The safety check operates similarly to the deadlock detection algorithm, except that it compares each process's remaining need rather than its currently outstanding requests. Each process must also specify the resources it might use over its runtime called the maximum claim [23, 19]. The remaining need is defined as the difference between a process's maximum claim and the resources it has already been allocated.

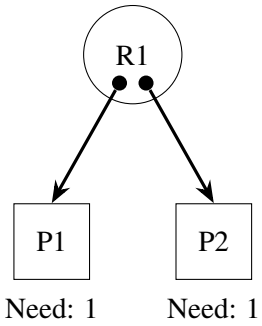


Figure 2.7: How the Banker's algorithm avoids deadlock. After we have temporarily allocated an R1 instance to P2, there is no process that is able to continue and allocating the resource therefore leads to an unsafe state. The resource instance is only given to P1.

Figure 2.7 illustrates how a request may be denied if the maximum claim of process P1 is known. A flagged unsafe state does not imply that a deadlock will occur if the request is granted, only that deadlock-free execution cannot be guaranteed. If P1 first releases all of its resources before requesting two units at once, P2 can be granted its last request and both processes can complete. The algorithm does not need to know exactly when allocations and deallocations will occur, only the maximum number of resources each process can hold at any given time.

The Banker's algorithm has a time complexity of $O(n^2 \cdot d)$ for n processes and d resource types [15]. Several variants have since been proposed under different assumptions about the underlying system, with Zohora et al. [44] reducing the time complexity to $O(n \cdot d)$. As discussed in Subsection 2.5.6, the time complexity of the deadlock handling algorithm is not the dominant cost in the setting considered in this thesis.

2.5 Related Work

Deadlocks, resource management, and scheduling have been studied extensively for decades across many fields, including operating systems, manufacturing, networks, and robotics. The domains in which deadlocks arise are diverse, and existing solutions are typically tailored to the particular domain in which they were developed.

Quantum computing is comparatively young, and the intersection between resource management and quantum systems is therefore narrower. The underlying principles of deadlock, however, are general enough that solutions developed in very different domains can still be compared meaningfully.

This section discusses how the literature addresses the deadlock problem. We cover deadlocks in quantum computer systems, real-time operating systems, and manufacturing systems.

2.5.1 Deadlocks in Computer Systems

Several implementations of deadlock handling have been proposed for multi-threaded computer programs. Dimmunix [24] allows deadlocks to occur and recovers from them, then uses patterns observed in previous deadlocks to avoid recurrences. Rx [32] similarly allows deadlocks to occur and relies on checkpointing to roll back to an earlier state. Gadara [37] performs both static and dynamic analysis, with the help of program annotations, to identify potential deadlocks, and optimizes for a maximally permissive policy in order to increase concurrency.

These approaches contribute relevant ideas to deadlock recovery and prevention. Their implementations, however, are tied to multi-threaded programs running on classical operating systems.

2.5.2 Distributed Systems

Work on deadlocks in distributed systems often focuses on developing algorithms that reduce the overhead and congestion of centralized or decentralized controllers [17, 26]. Such algorithms also face the risk of false positives, known as *phantom deadlocks*, in which the system rolls back even though no deadlock was actually present [26]. This risk applies equally when the distribution spans a network.

This thesis is concerned with deadlocks on a single network node. The deadlocks we consider can still arise as an indirect result of network communication through entanglement, but the resources involved in the deadlock are qubits on a single node.

Distributed databases are a subdomain in which deadlocks have been studied extensively [26, 23]. A central aspect of databases is the heavy use of checkpoints through transactions, which makes rollback to a previous state straightforward.

2.5.3 Manufacturing Systems and Sequential Resource Allocation

The deadlock problem is also relevant in sequential resource allocation systems used in automated manufacturing [33]. These systems can draw on knowledge of the entire production process to prevent and avoid deadlocks. When recovering from a deadlock requires manual intervention, detection and recovery strategies become expensive and are often unsuitable [33].

Flexible Manufacturing Systems (FMS) are a particularly well-studied subdomain in which deadlock avoidance, often using Petri-net models, has been extensively researched [34, 4, 39, 1, 40]. These models rely on detailed structural knowledge of the production process that is not available in the setting of this thesis, where program behavior may not be known before runtime.

2.5.4 Real Time Operating Systems

Real-time operating systems (RTOS) are typically concerned with completing tasks within a bounded execution time while minimizing overhead [35]. Software-level detection and

avoidance carry a runtime overhead that is generally considered too high for these systems, although hardware-level algorithms have been demonstrated to perform better in this setting [28]. Detection further introduces unpredictable runtimes through the recovery it triggers, and its effectiveness depends on how often deadlocks actually occur. Real-time schedulers frequently rely on priority and preemption to ensure that high-priority tasks complete within their deadlines [20].

Although Qoala adopts soft deadlines from this literature, the deadlock-handling assumptions of an RTOS do not transfer directly to the setting considered in this thesis. Qoala does not require bounded execution time, and recovery in this setting restarts the relative deadlines that were used to limit decoherence. Qoala also does not enforce deadlines, and deadlines only influence scheduling when they have been annotated on program blocks [36]. Deadlines nonetheless remain a useful primitive in Qoala, since they can also be inserted automatically by a compiler. The RTOS literature therefore contributes useful scheduling primitives, but it does not provide a deep understanding of how deadlock handling affects quantum network nodes.

2.5.5 Distributed Quantum Computing

Distributed Quantum Computing (DQC) aims to execute quantum circuits across multiple quantum computers, since the number of qubits available on a single QPU is limited [41, 5, 43].

Zhang et al. [42, 43] discuss deadlocks across QPUs and avoid them by compiling certain instructions, such as entanglement generation, into alternative virtual gates. The deadlocks they consider arise from mismatched expectations between the instructions executed on each node.

DQC concerns a single program whose execution is distributed across multiple QPUs, whereas Qoala targets separate programs that run on different nodes and communicate with one another over a network. These programs may not be known before runtime, and the deadlocks studied in this thesis are between such programs as they execute on a single node.

2.5.6 Time Complexity

Work on deadlocks in operating systems and networks often focuses on reducing the time complexity of the underlying algorithms [17, 44]. The time scale on which deadlock detection and avoidance run can substantially affect overall system performance in those settings. In our setting, where entanglement generation can take up to seconds, the time complexity of these algorithms becomes negligible.

2.5.7 Gap

When deadlocks are studied in a new domain, the literature typically focuses on how to solve the problem rather than on comparing different deadlock handling strategies. Cross-

domain comparisons through literature surveys are also limited, which makes it difficult to accurately predict which strategy will be best in an untested domain.

When such comparisons are made, the literature on classical systems tends to observe the following trade-offs [12, 23]. Prevention typically has a low runtime overhead but restricts concurrency, which can lead to underutilization and a longer makespan. Avoidance carries a higher runtime overhead while permitting more concurrent allocations, and certain Petri-net-based formulations such as Gadara [37] can be made maximally permissive. Detection and recovery also incur some runtime overhead and tend to be most suitable for systems in which deadlocks occur only rarely.

Although deadlock handling has been addressed and further optimized in many domains, new domains still appear in which a deeper understanding of the optimal strategy is needed.

A comparison of deadlock handling methods in quantum networks is currently missing from the literature. A clearer understanding of the impact of each class of algorithm is needed before optimization within this domain can be undertaken meaningfully.

Chapter 3

Method

A quantum network node running Qoala can execute multiple program instances that compete for a limited number of qubits. When two or more program instances each hold qubits while waiting for others to free up, the system deadlocks and no further progress can be made. This chapter describes how we approach deadlock handling on such a node.

We consider three strategies, each representing a different class of deadlock handling. Detection with recovery lets deadlocks occur and resolves them after the fact. Prevention removes one of the necessary conditions for deadlock up front. Avoidance keeps the system in a safe state by checking each allocation before granting it. We first describe how the four deadlock conditions of Section 3.1 map onto Qoala's resource model, then motivate our choice of algorithms and the shared inputs they operate on, discuss the time scales at which they run, and finally describe each strategy on its own. The integration of these algorithms inside Qoala is explained in Chapter 4.

3.1 Deadlocks in Qoala

Qoala already satisfies two of the four necessary conditions for deadlock, independent of any design choice. First, qubits are held with mutual exclusion since program instances acquire exclusive access through the memory manager (Section 2.2). Second, no resource preemption is possible. A running program's qubit state cannot be copied out and restored later, since reading the state collapses it on measurement. Classical operating systems can sidestep this by paging memory to disk, but no equivalent backup exists for qubit state. Hold-and-wait and circular wait remain the two conditions that any deadlock handling strategy on Qoala can act on.

Each qubit is modeled as a resource instance of a type. We consider up to two resource types, memory qubits and communication qubits, following the qubit roles introduced in Section 2.2. Because of the virtual memory abstraction in Qoala's memory manager, a program allocates a qubit of a given type rather than a specific physical qubit. The distinction between memory and communication qubits depends on the underlying memory topology and is not always meaningful. On NV centers the two roles are physically separate, while

on trapped ions any qubit in the trap can act as a communication qubit and the resource pool collapses to a single type. However, there are ways to improve on this model as we describe in **??**.

Figure 3.1 shows two deadlocked configurations on a single Qoala node, one for each topology class introduced in Section 2.2.

In Figure 3.1b, the node has one communication qubit and two memory qubits, as a simplified and minimal NV-center node. Program instance P_1 needs the communication qubit followed by one memory qubit, and P_2 needs both memory qubits followed by the communication qubit. The following task interleaving leads to deadlock.

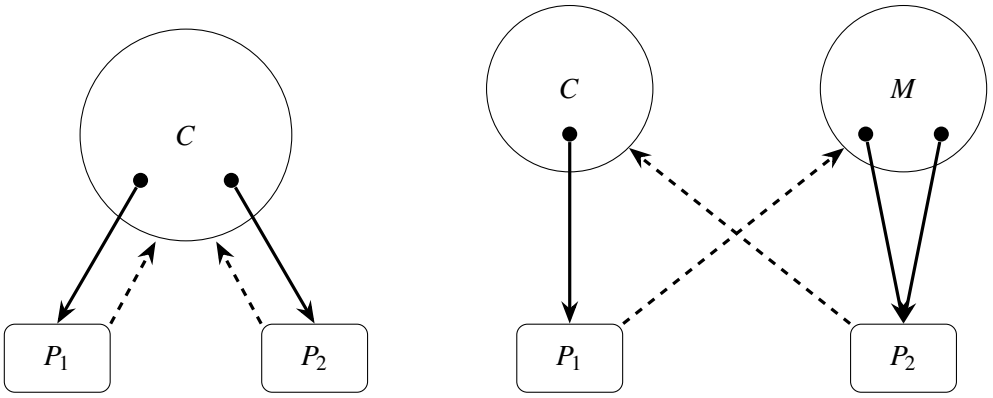
1. P_2 generates entanglement on the communication qubit and immediately swaps the state onto a memory qubit, freeing the communication qubit.
2. P_1 generates entanglement on the now-free communication qubit and holds it for its next quantum task.
3. P_2 allocates the remaining memory qubit for a local quantum routine and keeps it, so it now holds both memory qubits.
4. P_1 requests a memory qubit to swap its state onto, but both are held by P_2, so P_1 blocks.
5. P_2 requests the communication qubit to generate a third entangled pair, but it is held by P_1, so P_2 blocks.

Both program instances are now blocked on resources held by the other, and neither can proceed. In Figure 3.1a, the node exposes a single pool of four communication qubits, modeled after a possible trapped-ion node, and two program instances contend for it. P_1 and P_2 each hold a qubit and are waiting to allocate one more. Both qubits are allocated and each program instance is waiting on a qubit held by another, so the system is deadlocked.

3.2 Algorithms

We have implemented and evaluated three strategies, deadlock detection with recovery, prevention through request-release-all, and the Banker's algorithm [12, 19, 14]. These were chosen because they represent the three established classes of deadlock handling, namely detection, prevention, and avoidance. The three classes differ in the amount of concurrency they permit and the overhead of tracking the system allocations and de-allocations. There are extensions and optimizations of these algorithms, but these have mostly been left out in order to focus on the core principles of each strategy. We return to some of these extensions in the discussion of future work.

Each strategy operates on a different subset of inputs, described in detail in its own section. Detection and avoidance both track the allocation and request state of the running program



(a) Two communication qubits (C) shared between P_1 and P_2.

(b) One communication qubit (C) and two memory qubits (M) between P_1 and P_2.

Figure 3.1: Two deadlock configurations on a single Qoala node. Solid arrows from a resource instance to a program instance denote allocations and dashed arrows from a program instance to a resource denote pending requests. On the left, both communication qubits are allocated across P_1 and P_2, and each program instance is waiting on qubits held by another. On the right, P_1 holds the only communication qubit while waiting for a memory qubit that P_2 holds, and P_2 holds all memory qubits while waiting for the communication qubit.

instances, while prevention only inspects the maximum claim of each instance at admission. How these inputs are obtained from Qoala's task graph and memory manager is described in Chapter 4.

The strategies rely on a known maximum claim. Each program instance must have a finite maximum claim known before admission, which we compute through manual static analysis of each program, including unpredictable programs with loops and conditionals. Larger unpredictable programs make this analysis more complex, and automating it is left to future work. Restart does not strictly require this bound, but uses it on readmission to avoid more cyclic restarts.

3.3 Time Scales of Quantum Networking

The three time scales at which a quantum network node operates are introduced in Section 2.2. For deadlock handling to be viable on a real node, the algorithm should ideally complete within the time scale of local classical or quantum tasks, so that it does not stall waiting tasks that could otherwise run. The time complexity of deadlock detection and avoidance can substantially increase the runtime overhead. However, the limited number of programs and qubits on current hardware keeps that cost minimal, and we therefore do not treat it as a current area of optimization. Existing literature already covers this direction [44, 17]. In the Qoala simulator the deadlock handling algorithm runs alongside the simulated program and has no effect on its simulated duration, so the time cost of the algorithm is not directly visible in our results.

The three strategies have different degrees of this impact. Detection and avoidance both continuously run system checks and therefore contribute more to this overhead cost. Prevention only inspects the maximum claim of a program instance at instantiation and introduces less overhead than the other two. The results in Chapter 6 should be read with this context in mind.

3.4 Detection

The detection algorithm considers program instances that are blocked waiting on a resource while already holding other resources, since these are the only ones that can be part of a deadlock.

To detect deadlocks we use Algorithm 1 on all program instances. The algorithm follows Coffman et al. [12] with pseudocode inspired by [3, p. 188], extended to support multiple program instances and multiple resource types. We keep track of matrices of the available resources in the system, the allocated resources per program instance, and the resources each instance is currently waiting on. The algorithm marks each program instance as either completed or, if it is left uncompleted after the loop, as part of the deadlocked set. For n program instances and d resource types, the algorithm runs in $O(n^2 \cdot d)$ time.

Algorithm 1 Deadlock detection

```
1:  Initialize completed = [false, ...] and temporary copy of available resources
2:  while ∃ process i with completed[i] = false and request[i] ≤ available do
3:      available ← available + allocated[i]
4:      completed[i] ← true
5:  end while
6:  if completed has false entries then
7:      Non-completed processes are deadlocked
8:  else
9:      Not in deadlock
10: end if
```

3.4.1 Termination

After detection we resolve the deadlock by releasing resources. Our strategy terminates a program instance to make sufficient resources available. We base the choice of which instance to stop on the following factors.

- The program instance's runtime
- The number of allocated qubits
- How long its qubits have lived in memory

A Deadlock Impact Score is calculated from these factors. The program instance with the highest score is terminated and rescheduled by the system.

We aim to maximize program correctness and runtime, reduce lost work, and terminate as few processes as possible. The score therefore prefers to terminate a program instance that (1) has not been running for a long time, (2) has allocated a large number of qubits, and (3) has qubits largely affected by decoherence.

We only consider T_2 noise in our model, since T_2 is the shorter of the two noise time constants relevant to qubits in Qoala.

For a program instance i, we define the termination score S_i as

$$S_i = \frac{|q_i|}{|q|}\left(\alpha \exp\left(-\frac{r_i}{T_2}\right) + \beta \sum_{j \in q_i} \exp(-\delta_j)\right), \tag{3.1}$$

where

- q_i is the set of qubits currently allocated to instance i and q is the set of all qubits on the node, so $|q_i|/|q| \in [0,1]$.
- r_i is the elapsed runtime of instance i.
- T_2 is the dephasing time constant of the underlying hardware.
- $\delta_j = (T_2 - t_j)/T_2$ is the remaining fraction until qubit j has been in memory for T_2 time, where t_j is the time the qubit has been held. δ_j can be negative, in which case t_j has already passed T_2.
- $\alpha, \beta \geq 0$ are weights for the runtime and dephasing terms respectively.

The instance with the highest S_i is terminated, so a high score should mark an instance that is cheap to kill and frees many or already-noisy qubits.

The two exponential terms encode the two factors that drive this trade-off. The runtime term $\exp(-r_i/T_2)$ is close to one when the instance has just started and decays as it runs, so young instances are preferred victims. The dephasing term $\sum_j \exp(-\delta_j)$ grows as the qubits in q_i dephase, so instances holding already-noisy qubits become more attractive to kill. Both terms within the exponentials are normalized by T_2 to keep runtime and dephasing on a comparable scale. The exponential shape was chosen so that score differences amplify near the limits of each factor.

The score has several limit cases. If instance i holds no qubits, $|q_i| = 0$ and $S_i = 0$, since terminating it would free nothing. If T_2 is set to 0 or infinity to disable dephasing in simulation, the dephasing term is dropped and only the runtime term contributes. If a qubit has been held beyond its T_2, $\delta_j < 0$ and the dephasing exponential grows past one, since the qubit is treated as already fully decohered. A program instance holding such qubits is a desirable victim, since its remaining computation can no longer rely on those qubits and

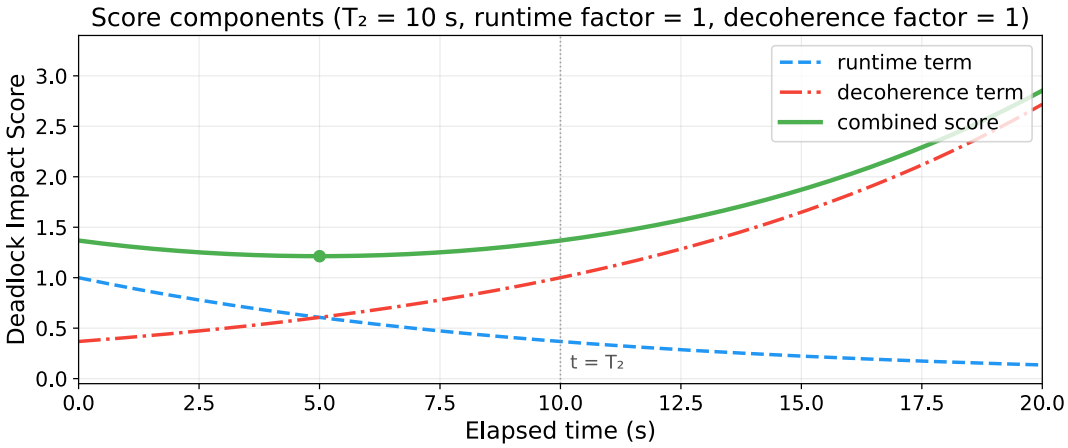


Figure 3.2: Example scoring of runtime and decoherence. T_2 is set to 10 seconds, and runtime and decoherence factors α and β are set to 1. The program instance has 1 out of 1 available qubits in memory from the start of its runtime.

terminating it frees them for new work. Because the dephasing exponential keeps growing once a qubit passes its T_2, the score has no upper bound. The decoherence term can then dominate regardless of how long an instance has run, so an instance holding old qubits stays the preferred victim even when it has done substantial work that a restart would discard. We return to this in the cyclic-restart discussion of Chapter 7. Concrete values of α and β for evaluation are described in Chapter 5.

See Figure 3.2 for an example of how a program instance might be scored over time. In the example we consider a program instance with one qubit, where the qubit is kept in memory from the start of the program instance's runtime. Most notably, the score is first reduced over time as the program instance runs, but increases as the qubit is exposed to decoherence.

3.4.2 Recovery

After a program instance has been selected and terminated, the system has to recover and ensure that progress can continue. The terminated instance releases its allocated qubits back to the pool, its remaining tasks are dropped from the task graph, and partner nodes in a distributed program are notified that the program has been aborted. The program is reinstantiated from the beginning. Recovery from a checkpoint has not been implemented and is left to future work. Mid-program checkpoint recovery is often infeasible, since the no-cloning theorem prevents copying an unknown quantum state and re-preparing the state from a classical description is typically equivalent to restarting the program from scratch. A checkpoint placed between blocks where no qubit is in memory avoids this constraint and has the potential to reduce the cost of restarting after a deadlock.

A naive restart can create a cycle where the same instance is terminated, restarts, and is selected again on the next deadlock. To avoid this, a restarted instance is admitted only once its full declared maximum is simultaneously available. Note that this does not allocate

all of the instance's qubits when it is rescheduled, only that enough qubits are free to lower the chance of another deadlock. Programs therefore need to define their maximum claim before running, similar to what prevention and avoidance require.

3.5 Prevention

To remove the hold-and-wait condition, we reserve all resources a program instance will need before any of its tasks are run. We refer to this strategy as request-release-all. The instance is admitted only if that many qubits can be reserved at once, and the reservation is released when the instance completes. The maximum claim of qubits must therefore be defined before execution. This strategy assumes that the maximum is finite and known before admission.

If not enough qubits are free, the program instance waits until a signal of freed resources is received, at which point its execution is reattempted. The strategy is conservative. It can deny admission even in cases where a deadlock would never actually have occurred, and it can therefore reduce concurrency and the overall throughput of the system. We chose reservation over the resource-ordering alternative described in Chapter 2, since ordering would require a global agreement on the type-ordering across programs that does not naturally exist in our setting.

3.6 Avoidance

Avoidance reuses Algorithm 1 from Section 3.4 but turns it into a safety check. The requested resources for a program instance are temporarily allocated and the algorithm is run, except that each process is compared against its *remaining need*, defined as its maximum claim minus what it has already been allocated, rather than its current pending request [14, 19]. If no completion order of programs exists for the modified state, the state is unsafe and the task has to wait until more resources are freed, at which point the safety check is run again.

We keep track of matrices of the available resources in the system, the allocated resources per program instance, their maximum claim, and their remaining need. The maximum claim covers the entire program rather than being based on regional maximums [27]. Extending the algorithm to use regional maximums has the potential to allow more permissiveness, but is out of scope for this thesis and is left to future work.

Chapter 4

Implementation

This chapter describes how the three strategies from Chapter 3 are integrated into Qoala. Detection, avoidance, and prevention all extend the resource-availability check that the QPU scheduler already runs before each task. The memory manager remains the interface through which qubits are allocated and freed. Restarts and terminations are coordinated by the node scheduler, which also notifies remote nodes for distributed programs. We refer back to Chapter 3 for the algorithms themselves and limit this chapter to the integration points in the Qoala simulator.

Figure 4.1 shows the internal components of a node in the simulator that our integration interacts with. Notably, the entanglement distributor is specific to the simulator. On real hardware, the netstack would interact with a heralding station rather than submitting requests to a centralized component. This distinction matters for the restart path of detection, where pending entanglement requests have to be cancelled in the simulator.

4.1 Detection

The QPU scheduler runs the detection algorithm of Section 3.4 before each task is started. Allocations are read from the memory manager's qubit mapping, and requests are read from the tasks in the task graph that have no precedence constraints. Running the check on a timed interval instead would reduce overhead with a large enough interval, but as discussed in Section 3.3 this overhead is not taken into account. If the check returns a non-empty deadlocked set, the QPU scheduler picks the program instance with the highest termination score and requests the node scheduler to restart it.

4.1.1 Recovery

The QPU scheduler does not restart the program instance itself. It signals the node scheduler, which then drops the instance's tasks from both processor schedulers, restarts it from its first block, and forwards a restart message to every remote node that hosts a partner instance.

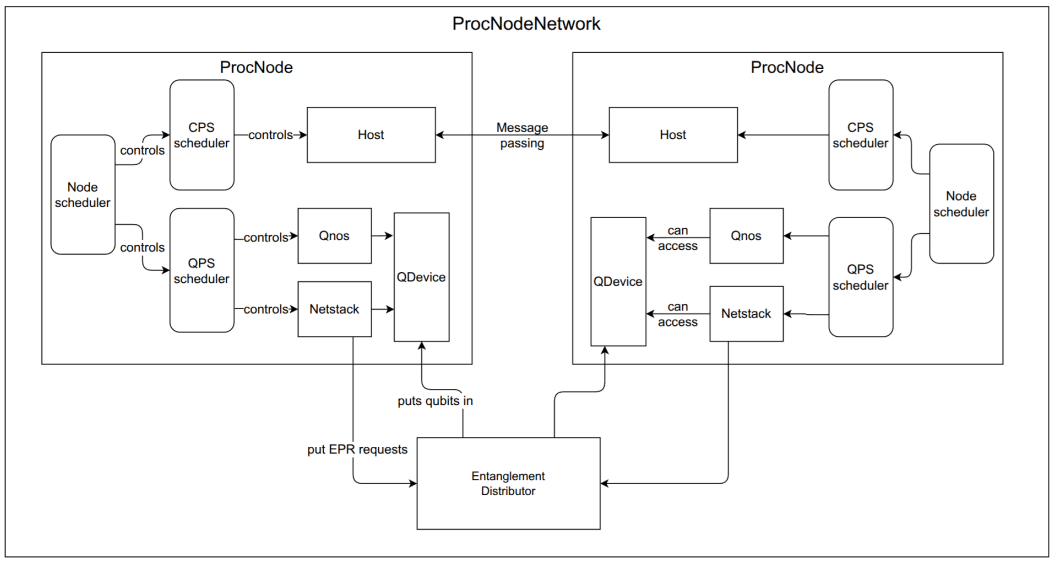


Figure 4.1: Diagram of the inner components of nodes in a network from [36]. The entanglement distributor is a simulator-specific abstraction over the heralding station that would exist in a physical implementation.

Two timing concerns can arise on the restart path, because a deadlock may be detected on a remote node while a task for the affected instance is running on the local QPU. The restart message arrives at the local node scheduler with no synchronization to that running task. First, the local QPU may be suspended inside an EPR task for the terminated instance. The qubit free is then deferred until the running task completes, so that the memory manager can release its mapping without double-freeing. Second, an entanglement request submitted before the restart can still match against a request from the restarted instance. The node scheduler therefore sends a cancel for any pending requests of the terminated instance to the entanglement distributor through the netstack. Figure 4.2 shows an example timeline.

The cyclic-restart mitigation described in Section 3.4 cannot rule out all repeated restarts of the same instance in practice. As a safety net, the node scheduler tracks how many times each program instance has been restarted. If any instance reaches 20 restarts, an error is raised and the Qoala simulator shuts down. Reaching this bound indicates that the deadlock-handling policy is failing to make progress, and stopping the simulation surfaces the failure rather than masking it as an unbounded restart loop.

4.2 Prevention

Prevention is implemented as an extension of the QPU scheduler's resource-availability check. A task for a program instance is only allowed to run once the full set of qubits the instance has declared in its program metadata can be reserved simultaneously from the memory manager. If not all qubits are available, the task is held until resources are freed

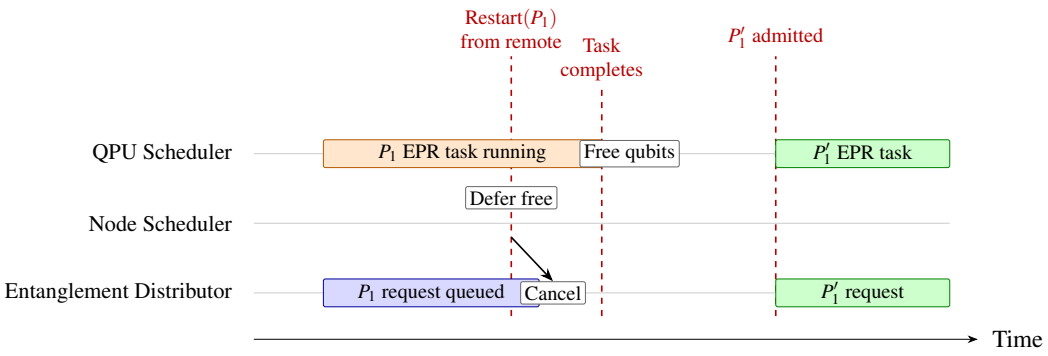


Figure 4.2: Timeline of a remote-triggered restart that arrives while an EPR attempt for the victim program instance P_1 is in progress on the local node. Because the local QPU task is still executing, the qubit free is deferred. The node scheduler cancels the pending request in the entanglement distributor so that it cannot match against the new request submitted by the restarted instance P_1'. The deferred free runs once the running task completes.

and the check is rerun. Once the last block of the program has run, all held qubits are freed. Prevention assumes that the maximum claim of qubits used over the lifetime of a program instance is declared in the program metadata.

4.3 Avoidance

Avoidance extends the QPU scheduler's resource-availability check with the Banker safety check from Section 3.6. For each task that has no precedence constraints, the scheduler asks whether granting that task's allocation would still leave the system in a safe state. Current allocations are read from the memory manager, in the same way as for detection, and the maximum claim of each program instance is taken from its program metadata. Unsafe tasks have to wait until resources are freed.

We evaluate these strategies and compare them across classical and quantum performance metrics in Chapter 5.

Chapter 5

Evaluation

In this chapter we detail how the implementation was evaluated through the Qoala simulator. We specify evaluation metrics, network configurations, tested applications, and connect them to our research questions.

5.1 Deadlock Handling

We evaluate three deadlock handling strategies, deadlock detection with restart, avoidance, and prevention. We label them Restart, Avoid, and Prevent. We also include two baselines, Enough Qubits and Linear, for minimal and maximal permissiveness. Enough Qubits gives each node more qubits than every program instance running on it will ever need at the same time, so no deadlock can occur. Linear imposes a precedence constraint that runs program instances strictly one after the other, so no two instances are interleaved. We also include Raise, which shuts down the simulation at the time of deadlock, allowing us to identify when deadlocks occur.

5.2 Metrics

We use two classical duration metrics and one quantum success metric.

The first classical metric is makespan, the total duration from the start of the first program instance to the completion of the last instance in a workload. We also include a normalized makespan over the total number of entanglement generations in a workload so that workloads of different sizes can be compared. For example, three instances of a program that generates two entanglements each gives a normalization factor of $2 \cdot 3 = 6$.

The second classical metric is QPU execution time, the average execution time of a program instance from the start of its first quantum task to the end of its last. Classical operations before the first allocation and after the last release are excluded. For a program instance that has been terminated and restarted, we report two values, one from the first start and one from the last restart.

The quantum metric is success probability, the fraction of successful measurement outcomes across all program instances in a configuration. Each outcome is binary, true or false, and the application-specific definition of a successful outcome is given in Section 5.4. We report the mean success probability across iterations together with the standard error of the mean as the uncertainty measure (see Section 5.6).

We want to execute our programs in the shortest time possible while still preserving successful outcomes. Because of the need for less noise in programs like BQC [9], the success probability of applications might be prioritized over the makespan.

The two metrics are intertwined because qubit decoherence over time affects success while strategy choices affect duration. Consider Restart and Avoid. Avoid delays program instances deemed unsafe, which leaves their qubits in memory longer, resulting in more decoherence. Restart instead lets an instance run until a deadlock occurs, then terminates and re-runs it, which lengthens the makespan but resets the time a qubit is kept in memory. Under Avoid, a longer makespan generally comes with more decoherence and therefore a lower success probability. Restart breaks this link, because each restart resets a qubit's time in memory, so Restart can show both a higher makespan and a higher success probability than Avoid.

5.3 Configuration

A base configuration was used for all scenarios. To isolate the effect of deadlock handling strategies on success, decoherence is the only source of noise in the simulations. Entanglement generation is treated as deterministic, the entangled state is taken to be ideal, and gate fidelities are set to one. A successful outcome therefore depends only on how long qubits are held in memory, which in turn depends on the scheduling decisions made to handle deadlocks.

The configuration of the network and simulation is defined in Table 5.1.

Scenario-specific configurations that override values in Table 5.1 are stated in Section 5.5. Every configuration was simulated 1000 times.

To isolate the impact of deadlock handling strategies, only one node in each simulation was allowed to deadlock. We refer to this node as the tested node. Every other node ran exactly one program instance so that nondeterministic scheduling on remote nodes did not perturb the tested node.

5.4 Evaluated Applications

We evaluate four existing applications (BQC, GHZ, Ping-Pong, and Teleport) and four applications designed for this thesis (Fluctuating Needs, Deadlock Loop, Late Lock, and Mem-Comm Allocation). To test the effect of decoherence of these synthetic applications, they were designed to rotate qubits to the following states: $|+\rangle$, $|-\rangle$, $|i\rangle$, $|-i\rangle$. This was done either through two local routines rotating a qubit stepwise to a desired outcome of

Table 5.1: Default configuration for the network and nodes.
[a] This value was intended to be 0.026 ms [31]. Since the same gate duration is applied across all configurations, the difference does not affect the comparative results, and it is in any case negligible relative to the time scale of entanglement generation.

Configuration	Value
Node	
`T1`	∞
`T2`	10 s [31]
`Single Qubit Gate Durations`	0.26[a] ms
`Two Qubit Gate Durations`	0.11 ms [31]
`Single Qubit Gate Fidelity`	1.0
`Two Qubit Gate Fidelity`	1.0
Link	
`Expected Gen. Time`	1s [31]
Latencies	
`Host / Qnos Instr.`	1 µs
`Host Peer Latency`	100 µs
`Internal Scheduling Latency`	0.1 µs
Scheduling	
`Scheduling`	Nondeterministic
`Deadlines`	No deadlines set
`Critical Sections`	No critical sections used
`Network Schedule`	Time bins 10% longer than entanglement time
NetSquid	
`Quantum State Representation`	Density Matrix
Deadlock Handling Strategy	
`Termination Factor` α	1.0
`Termination Factor` β	1.0

$|0\rangle$ or $|1\rangle$, or through a Remote State Preparation (RSP)-like [7] protocol for entanglement. Simply put, the remote node measures an entangled qubit and sends the outcome which the node uses to rotate its qubit to an expected state. For each application we describe its purpose, the request-release pattern on the tested node, the maximum claim, and the criterion that determines whether an instance is successful. See Appendix A for more details on the application structures.

5.4.1 BQC

Blind Quantum Computing (BQC) lets a client delegate a measurement-based computation to a server without the server learning what is being computed [9]. The server generates a sequence of entangled qubits with the client, applies a fixed pattern of Controlled Phase (`CPHASE`) gates across them, and then performs a sequence of measurement rounds. Before each measurement the client sends a classical angle that selects the measurement basis. The server is the tested node in our simulations. We evaluate two variants of the protocol that differ in the size of the entangled state.

BQC2q The server generates two EPR pairs, applies one `CPHASE` between them, and runs two measurement rounds. The application runs a verification round where one qubit has a deterministic measurement outcome the client predicts. An instance is successful when the server's measurement of this qubit matches the prediction. Since this only fails through dephasing, the metric reflects how long a strategy keeps the qubit in memory before measuring it. The server's maximum claim is 2 qubits and the client's maximum claim is 1 qubit.

BQC10q The server generates ten EPR pairs and applies `CPHASE` gates in a brickwork pattern across them, then measures the qubits in the order 0, 5, 1, 6, 2, 7, 3, 8, 4, 9. This corresponds to a `CNOT` gate on the server. The server's maximum claim is 10 qubits and the client's maximum claim is 2 qubits, since the client prepares each input state on a second qubit before teleporting it. An instance is successful when the server's measurements of qubits 4 and 9 are equal meaning that the `CNOT` was correctly applied.

5.4.2 GHZ

The GHZ protocol creates a three-party GHZ state shared between Alice, Bob, and Charlie, and verifies it by measuring the state on each node. Bob acts as the hub. It first generates an EPR pair with Alice, then a second EPR pair with Charlie. Bob entangles its two qubits, measures one of them, and sends the outcome to Charlie as a correction bit. Charlie applies the correction, fuses the received qubit with a locally prepared qubit, measures one of them, and sends its outcome back to Bob as a second correction bit. At the end, all three nodes measure their remaining qubit. Alice's maximum claim is 1 qubit, while Bob and Charlie each have a maximum claim of 2 qubits. Bob is the tested node in our simulations. An instance is successful when the parity of the three measurement outcomes is even, which confirms that the qubits retained the correlation expected of an intact GHZ state rather than losing it to decoherence.

5.4.3 Ping-Pong

Ping-Pong teleports a state from a sender to a receiver and immediately back to the sender. The sender prepares its qubit in one of six initial states ($|+\rangle$, $|-\rangle$, $|0\rangle$, $|1\rangle$, $|i\rangle$, $|-i\rangle$), chosen uniformly at random per instance. It then generates an EPR pair with the receiver, performs a Bell-state measurement on the two qubits, and sends the two correction bits to the receiver. The receiver applies the corrections to its qubit and then generates a second EPR pair with

the sender, performs a Bell-state measurement, and sends its correction bits back. The sender applies the second set of corrections to its remaining qubit and measures it. Both the sender and the receiver have a maximum claim of 2 qubits and the program generates two EPR pairs in total. The sender is the tested node in our simulations. The sender's final measurement is performed in the basis matching the initially prepared state, so a fully coherent qubit yields outcome 0. An instance is successful when this outcome is 0.

5.4.4 Teleport

Teleport transfers a state from a sender to a receiver using one EPR pair and classical corrections. The sender prepares its qubit in one of six initial states, chosen uniformly at random per instance. It then generates an EPR pair with the receiver, performs a Bell-state measurement on the two qubits, and sends the correction bits to the receiver. The receiver applies the corrections to its qubit and measures it. The sender has a maximum claim of 2 qubits and the receiver has a maximum claim of 1 qubit. A program instance generates one EPR pair. The sender is the tested node in our simulations. The receiver's measurement is performed in the basis matching the prepared state, so a fully coherent qubit yields outcome 0. An instance is successful when this outcome is 0.

The four applications below were designed for this thesis to expose deadlock conditions that the existing applications do not cover. Fluctuating Needs tests non-monotone resource use. Deadlock Loop produces deadlocks at a tunable frequency. Late Lock delays the deadlock until late in a program instance's runtime. Mem-Comm Allocation produces a deadlock pattern on a node that separates communication and memory qubits.

5.4.5 Fluctuating Needs

Fluctuating Needs is a program designed to acquire and release qubits before reaching its peak claim. Its number of held qubits rises, falls, and rises again, so that strategies relying on monotone resource use are tested under non-monotone behavior.

The program runs the following sequence on the tested node.

1. Request 1 qubit
2. Request 1 qubit
3. Release 2 qubits
4. Request 2 qubits
5. Request 1 qubit
6. Release 3 qubits

We use three variants that differ in how each request is fulfilled. The *ql* variant uses local allocations for every request. The *qc* variant generates every request through entanglement

with a partner node. The *mix* variant generates requests 2 and 5 through entanglement and uses local allocations for the rest.

The maximum claim is 3 qubits. Each qubit is rotated by $\pi/2$ around X at request and by another $\pi/2$ at release, so a fully coherent qubit ends in $|1\rangle$. Where a request is fulfilled through entanglement, the generation is followed by a classical-communication task that receives a correction bit from the partner and a local routine that applies this correction, bringing the entangled qubit into the known state on which the $\pi/2$ rotations act. Each instance produces five such measurement outcomes. An outcome is successful when its value is 1.

5.4.6 Deadlock Loop

Deadlock Loop is a program designed to deliberately produce deadlocks at a tunable frequency on a memory-constrained node. The frequency is controlled by a single parameter without changing the number of instances or the number of available qubits, isolating the effect of deadlock frequency from these two factors. The program runs $N = 20$ iterations of a base sequence and inserts an additional allocation every k iterations to drive contention into the deadlock regime. The base iteration generates and releases a single EPR pair, which keeps the per-iteration runtime stable across frequencies so that changes in makespan between configurations can be mostly attributed to the deadlock handling strategy rather than to the extra work introduced by raising the frequency.

Each iteration runs the following sequence on the tested node.

1. Generate an EPR pair with the partner node.
2. Release the EPR qubit by measurement.
3. Prepare a local qubit in $|+\rangle$.
4. Increment a counter; if the counter exceeds k, run an extra EPR generation followed by a release before continuing.
5. Rotate qubit to $|0\rangle$ and measure.

The loop terminates after N iterations. The value of k is derived from a frequency parameter f as $k = \lfloor N/(f+1) \rfloor$, so increasing f increases how often the extra allocation runs and therefore the chance of a deadlock between concurrent instances. The starting count is chosen uniformly at random in $[0,k]$ per instance, so the extra allocations of different instances do not all fire at the same iteration. Note that a starting count closer to k might trigger another iteration.

The maximum claim is 3 qubits and the tested node is the one running the loop above. The success probability of the program is defined by the percentage of iterations that resulted in 0 on measurement. The held qubit was predicted to decohere when the extra EPR generation was executed, and even more if it would be deemed unsafe to execute.

5.4.7 Late Lock

Late Lock is a program designed to delay the deadlock condition until late in a program instance's runtime. This isolates the cost of recovering from a deadlock after substantial work has been done, in particular the cost of discarding already-generated entanglement under the restart strategy.

The program runs six allocations on the tested node, split into two phases.

1. Phase 1 (four rounds): generate one EPR pair, measure the qubit, release. No qubit is held between rounds.
2. Phase 2: Generate a fifth pair which the receiver measures to send a correction bit. The sender rotates its qubit to $|+\rangle$ based on the correction. This is then done for a sixth EPR pair. Finally, the sender rotates these back to $|+\rangle$ and measures.

The program's success is defined by each of the two Phase 2 measurements that is 0, corresponding to a fully coherent state. The maximum claim is 2 qubits.

5.4.8 Mem-Comm Allocation

Mem-Comm Allocation is a pair of programs designed to produce a deadlock between two interleaving instances on a node that separates communication and memory qubits. One program first generates entanglement, receives a correction bit, then requests a memory qubit, and finally measures the two qubits. The other program performs these steps in the reverse order, requesting the memory qubit to put in $|+\rangle$ before generating entanglement. When the two programs run together on a node with a limited number of communication qubits, each instance holds the resource the other needs. The program uses one communication and one memory qubit per instance with a maximum claim of 2 qubits. A measurement is successful when its outcome is 0, which corresponds to a fully coherent $|+\rangle$ state at release.

The selection and design of these applications was driven by two criteria. First, every application that holds two or more qubits across separate requests can satisfy the hold-and-wait condition for deadlock, alongside the mutual-exclusion and no-preemption conditions that are already inherent to Qoala and its qubit memory. Second, the applications cover a range of allocation patterns and evaluate the hypotheses defined in our scenarios. They do not cover all possible applications, but they represent a wide range of possible patterns.

5.5 Scenarios

A *workload* is the set of program instances that run concurrently on the tested node, defined by which applications run and how many instances of each. A *scenario* pairs a workload with a node and network configuration and varies one factor to test a hypothesis. Each subsection below defines one scenario. It states a hypothesis about how the strategies should

compare under one experimental factor, gives the reasoning behind it, and describes how that factor is varied to test and falsify the prediction.

Across the scenarios we predicted in general a single ordering pattern to emerge. Restart would have the lowest makespan when deadlocks are rare and easily recoverable, since it pays no cost in their absence. As deadlocks become more frequent or more costly to recover from, Restart's makespan increases above Avoid and Prevent. Prevent was predicted to serialize contended instances for a higher success probability at the cost of makespan, while Avoid should leave room for concurrency at the cost of longer qubit hold times and the reduced success probability that follows. Similarly, we predicted the baselines would bracket the strategies. Linear would have the highest success probabilities but the highest makespan, and Enough Qubits around the lowest success probabilities but the lowest makespan.

In the following sections, we list several scenarios connected to our research questions which we will share results of and discuss later. Per-Application Scaling, Network Schedule, Late Deadlocks, and Deadlock Frequency address RQ1 by varying which contention regime is exercised. Per-Application Scaling, Late Deadlocks, Entanglement Generation Time, Longer Bin Length, and Topology Variation address RQ2 by varying the number of qubits, program instances, program sizes, or network and node configurations. Termination Factors addresses RQ3. Results for each scenario are reported in Chapter 6.

5.5.1 Per-Application Scaling

As stated in our prior general expectations, we predicted Restart to yield the lowest makespan when contention was low, since Avoid and Prevent block tasks that would otherwise run without deadlocking. As contention rises with more instances, we predicted the cost of failed restarts to outweigh this advantage and Prevent's makespan to grow steepest, since Prevent serializes contended instances.

To evaluate this we ran BQC2q, GHZ, Ping-Pong, Teleport, and the three Fluctuating Needs variants individually, sweeping the number of instances from 1 to 9 and the number of qubits on the tested node from 2 to 7. Fluctuating Needs requires up to 3 qubits and was therefore run from 3 to 12 qubits instead. Each strategy was simulated for every combination, with both baselines included.

We predicted the strategy ordering to hold under heterogeneous contention, where instances with different request-release patterns compete on the same node. Heterogeneity can change which instances reach a deadlock first but not the underlying trade-off between Restart, Avoid, and Prevent. We therefore predicted higher variance in this case because of the nondeterministically chosen tasks.

To evaluate this we ran a mixed workload of 2 BQC2q, 1 GHZ, and 1 Ping-Pong, duplicated one and two times on a node with 2 to 7 qubits, where four instances corresponds to a single duplication. We refer to this workload as the "Mixed" workload hereafter.

5.5.2 Without a Network Schedule

A network schedule aligns entanglement attempts in time bins, which forces interleaved execution and possibly raises the chance of deadlocks. Without one, we mimic a network schedule able to generate entanglement on-demand. We predicted Avoid and Prevent to benefit the most with reduced makespan because every available time to generate entanglement would be filled. Meanwhile, a program instance restarted after deadlock would need to regenerate entanglement, predicted to lead to a higher makespan than the other strategies. For success, we predicted the success probability for prevention to be slightly reduced from linear execution, because prevention would allow interleaving entanglements, resulting in decoherence across entanglements. We predicted a similar pattern for avoidance and restart, with their higher concurrency amplifying these differences from linear execution further.

To evaluate this we re-ran the Per-Application Scaling scenarios including Late Lock both with and without a network schedule.

5.5.3 Late Deadlocks

We predicted Restart to yield a worse makespan than Avoid and Prevent when the deadlock occurs late in a program instance's runtime, since Restart discards all already-generated entanglement and re-runs the program from the first block. Without a network schedule the effect would be somewhat reduced, since less forced concurrency through time bins reduces the chance of repeated deadlocks after a restart.

To evaluate this we used two workloads. The first uses Late Lock, which reaches its deadlock condition only after the fifth allocation, after four prior EPR generations have completed. The second uses BQC10q, which can deadlock after each entanglement generation. Late Lock was run with 2 to 9 instances on a node with 2 to 7 qubits, while BQC10q was run with 1 to 4 instances on a node with 15 and 20 qubits.

5.5.4 Deadlock Frequency

We predicted the best-performing strategy to shift from Restart toward Avoid and Prevent as the chance of deadlock increased, since the cost of failed restarts grows with frequency while the cost of waiting under Avoid and Prevent does not.

To isolate the effect of frequency from the number of instances and qubits, we used Deadlock Loop with 10 instances and varied the frequency parameter f from 0 to 6 on a node with 6 qubits. This would result in k values of 2, 3, 4, 5, 6, 10, and 20.

5.5.5 Entanglement Generation Duration

The time to generate entanglement is based on factors such as the distance between nodes and the underlying hardware. It is therefore relevant to evaluate the strategies across different magnitudes of durations.

We predicted that longer entanglement generation durations would widen both the cost of failed restarts under Restart and the cost of holding qubits under Avoid, while preserving

the strategy ordering for makespan and success. This prediction would be refuted if the ordering changed, or if Avoid showed no penalty as the duration grew.

To evaluate this we ran GHZ with three expected entanglement generation durations of 5 s, 1 s, and 0.01 s on a 3-qubit node with 4 instances. We kept the bin length 10% longer than each entanglement duration.

5.5.6 Longer Bin Length

The length of each time bin determines how many entanglement generations can be fulfilled. What is the strategies' impact when time bins can fit all entanglement generations of programs?

We predicted longer time bins to reduce the makespan penalty of Restart and Prevent in workloads where contention resolves within a single bin, since two entanglement generations then fit inside one bin and a deadlock can clear without crossing a bin boundary. Because entanglement generations could fit within one time bin, the success probability was predicted to be similar for all strategies.

To evaluate this we ran BQC2q, GHZ, Ping-Pong, and Teleport on a 3-qubit node with 4 instances, using a network schedule with a bin length 2.1 times longer than the entanglement generation duration.

5.5.7 Termination Factors

For the deadlock impact score introduced in Section 3.4, we predicted the runtime factor alone to optimize for makespan, since young instances are preferred victims and re-runs are cheap. We predicted the decoherence factor alone to optimize for success probability, since instances holding the noisiest qubits are preferred victims and the freed qubits return with full coherence. We predicted the combination of both factors to fall between the two single-factor cases.

To evaluate this we ran three factor configurations on Fluctuating Needs *qc* variant, BQC2q, and Late Lock from the scenarios described earlier. This was run on a node with 4 instances competing over 3 qubits for BQC2q and Late Lock and 2 instances competing over 4 qubits for Fluctuating Needs *qc*. The difference comes from the latter having a maximum claim of 3 qubits.

We label the different factor combinations we evaluated with the following:

- RF, runtime factor only ($\alpha = 1$, $\beta = 0$).
- DF, decoherence factor only ($\alpha = 0$, $\beta = 1$).
- RF+DF, both factors at equal weight ($\alpha = 1$, $\beta = 1$).

5.5.8 Topology Variation

We predicted deadlocks to be more prevalent on a topology separating qubits into two resource types because there are more cases where deadlocks occur even though there are qubits available, but of the wrong resource type. This would result in more restarts, increasing Restart's makespan above Avoid. With the need for all network applications to use entanglement, Prevent was also predicted to be penalized because of the need for a communication qubit over the duration of the program. Avoidance was thereby predicted to utilize this situation better. Yet, the success probability was still predicted to perform better for Prevention because of the serialized execution whereas Avoid might hold on to qubits while its next task is unsafe.

To evaluate this we ran Mem-Comm Allocation on a node with 1 to 4 memory qubits and 1 communication qubit with 4 instances in total (2 of each program variation). This was the only scenario in which the node topology differed from the default Uniform.

5.6 Statistics of Outcomes

The success metric is defined per application as described in Section 5.4. Each instance of a program produces one or more binary outcomes, where 1 marks success and 0 marks failure. We report the mean success probability across iterations together with the standard error of the mean.

We do not report the standard deviation because the discrete nature of the per-iteration mean makes it uninformative. For a workload that produces X outcomes per iteration, the per-iteration mean takes one of only $X+1$ values, evenly spaced between 0 and 1. For example, two outcomes per iteration give possible means $\mu_i \in \{0, 0.5, 1\}$. With a single-outcome workload the per-iteration mean is either 0 or 1, which inflates the standard deviation and makes percentiles uninformative. With 1000 iterations per configuration, the standard error of the mean is small and provides a more meaningful uncertainty estimate.

5.7 Simulation Hardware

All simulations with a set configuration were run for 1000 iterations. The machine has 80 Intel Xeon Gold cores at 3.9 GHz with 192 GB of RAM.

Chapter 6

Results

We now present the results from the evaluations, showing how the different deadlock handling strategies affect our tracked metrics and how they compare against our baselines. Each section below presents the results for one scenario defined in Section 5.5 and revisits the prediction made for that scenario. We discuss some individual results as they arise, while the complete discussion for this thesis is found in the next chapter.

6.1 Per-Application Scaling

Across most tested workloads with high contention between processes, prevention usually has a higher makespan. Figure 6.1 shows how makespan is significantly higher for prevention, between 44% and 171% ± 2.4% above restart and avoidance across these workloads. Restart and avoidance have very similar makespans across most workloads, within 1.4% ± 1.5% of each other in each tested case. This similarity holds even when a restarted program instance has already generated entanglement that it would need to regenerate. Looking at the timeline of BQC in Figure 6.2, we can see for Restart that the restarted program instance reruns its first entanglement at the next possible time bin, ultimately reaching the same makespan as Avoid. However, there are exceptions such as Late Lock and GHZ. The makespans of Restart and Avoid are still lower than Linear and higher than the Enough Qubits baseline. As predicted, the Linear baseline regularly has a higher makespan, while the Enough Qubits baseline has a lower makespan than the deadlock handling strategies.

For some workloads, however, prevention has a higher makespan than the Linear baseline. This can happen when a nondeterministically chosen local routine is allocated qubits and blocks the entanglement generation for the next time bin. The linear baseline, by contrast, schedules the next program instance in the same order as time bins. However, because prevention still runs instances concurrently when there are enough qubits, its makespan is drastically lower than linear execution as the number of qubits increases.

The deadlock handling strategy has no meaningful impact on the success probability for most of these workloads. The number of instances is instead the most impactful factor. Figure 6.3 shows the success probability across deadlock handling strategies for six different

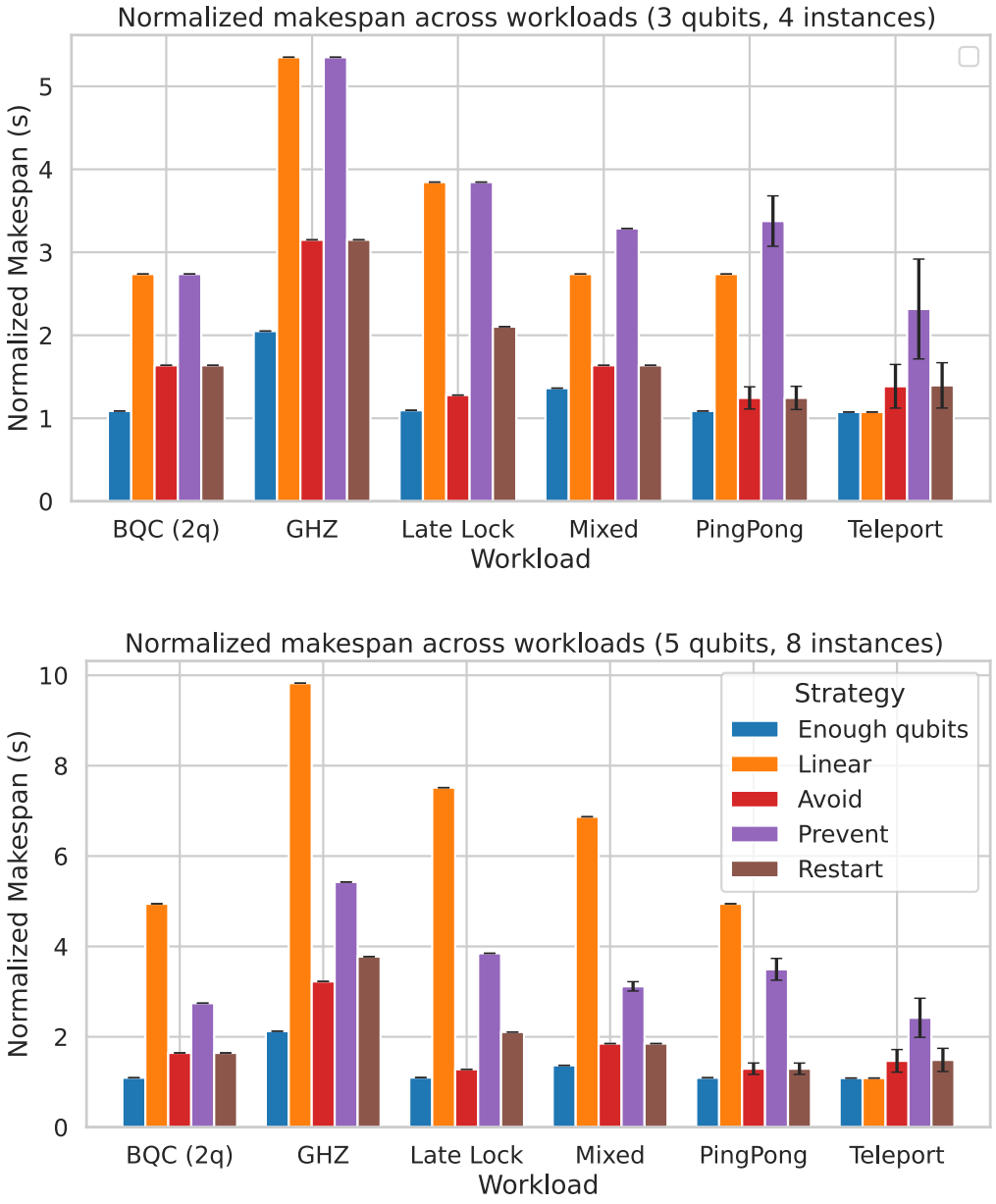


Figure 6.1: Makespan for 6 different workloads. Each workload has 3 or 5 qubits and 4 or 8 program instances. The first workload contains 1 or 2 of each of the listed programs (2 of BQC, PingPong, GHZ) while the rest has duplicates of the same program.

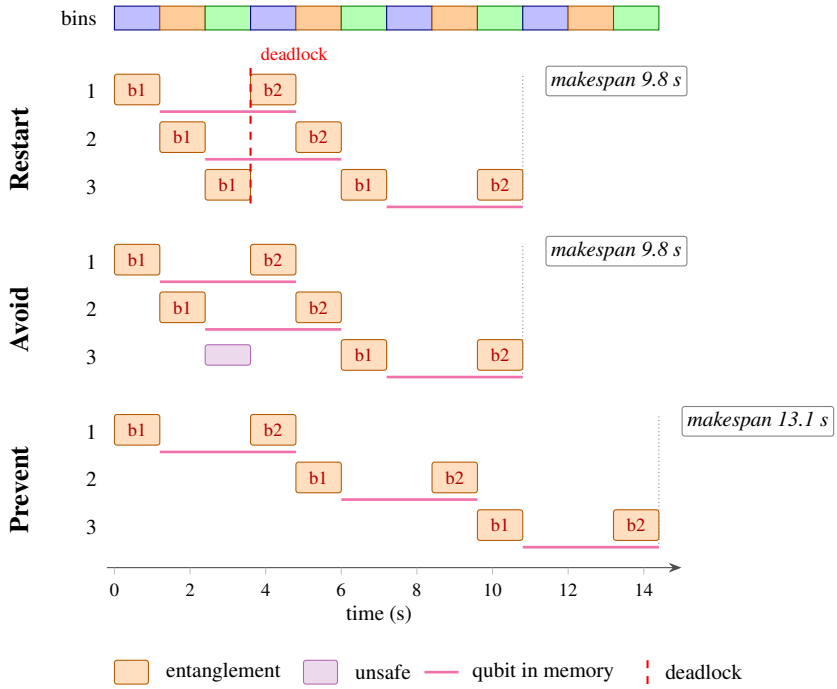
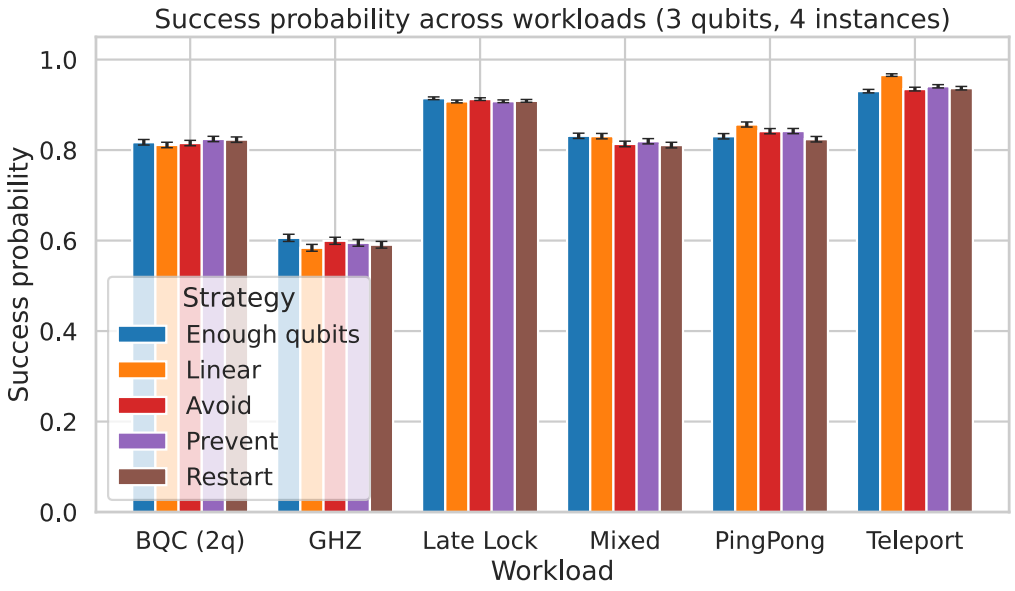


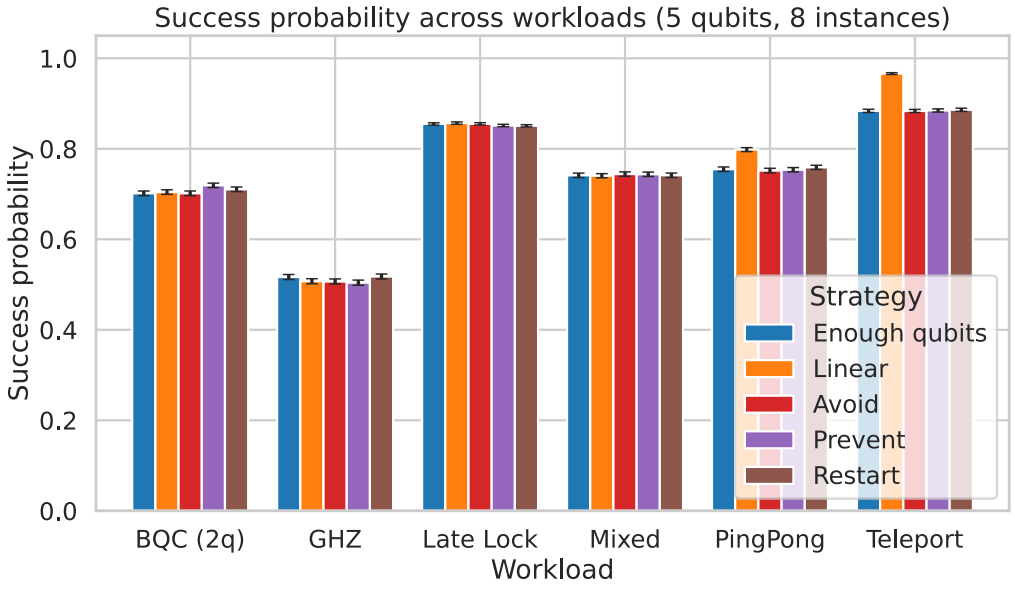
Figure 6.2: Schematic timeline of BQC with 3 qubits and 3 instances under the three deadlock-handling strategies. Each row is a program instance and each block represents a request routine. BQC executes two blocks (b1 and b2). There are also local routines and classical blocks, but they have been omitted because of their relatively small duration. The line below blocks represents the time a qubit is in memory and dephases. At the time of deadlock, instance 3 has acquired the last qubit and at that point each program instance needs another qubit to continue. They are deadlocked.

workloads. Only linear execution is shown to have an increased success probability, for PingPong and Teleport, which start with local routines. This advantage is 1.5 ± 1.6 and 3.9 ± 1.3 percentage points for PingPong and 2.5 ± 0.9 and 8.0 ± 0.8 percentage points for Teleport over the best deadlock handling strategy, while for GHZ and BQC the strategies stay within 1.6 ± 2.1 percentage points of each other.

The consistent minimal differences in success imply that applications were in most cases able to free a qubit from one time bin to the next, made possible because competing programs had freed their held qubits beforehand. Increasing the number of instances also increases the duration between analogous time bins, correspondingly increasing the time a qubit spends in memory. This is consistently the largest factor for a program instance's success.



(a)



(b)

Figure 6.3: Success for 6 different workloads. Each workload has 3 or 5 qubits and 4 or 8 program instances.

6.2 Fluctuating Needs

There are still cases where increasing the contention changes the success probability among strategies. Figure 6.4 shows a different trend for the *mix* variant of Fluctuating Needs. For this variant, avoidance keeps a higher success probability than the other strategies as the number of instances increases. This occurs when a local routine that allocates a qubit is scheduled closer to its next entanglement as a result of it being deemed unsafe to execute prior. Prevention, on the other hand, is able to schedule this local routine immediately after the entanglement request. This is possible because each time bin is 10% longer than the entanglement duration.

On the *mix* variant, Restart does not complete for all iterations. At three, four, and five instances all 1000 iterations hit the 20-restart limit, so no makespan or success is reported for Restart there. At six instances, 41 iterations fail the same way while the rest complete, though restarts remain frequent. The decoherence term in the Deadlock Impact Score selects an instance whose qubits have decohered the most. Both short-lived and long-lived instances were repeatedly terminated and restarted, together exhausting the 20-restart budget before any instance could complete.

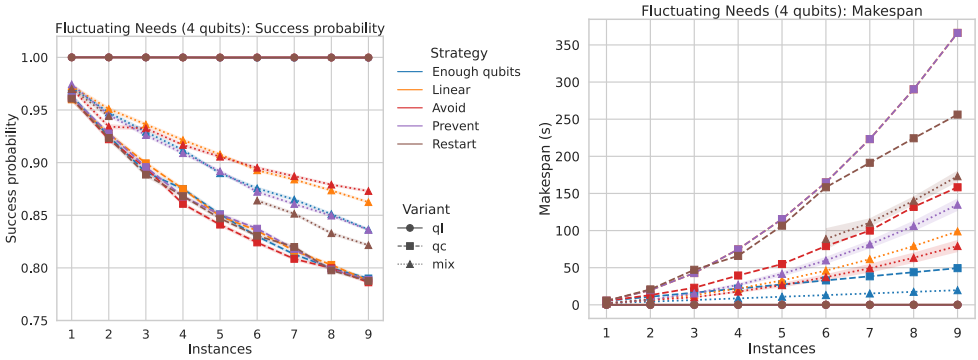
A possible future direction could be to recast the impact score as a cost that grows with runtime, rather than letting the dephasing term exponentially grow once a qubit has been held past its T_2. The longer an instance has been running, the more expensive terminating it would become, which would discourage the repeated selection that drives the restart loop. There is also a discussion of how many qubits are acceptable to decohere for the benefit of a reduced makespan. If the answer is none, the readmission rule could also require an acquire-all and release-all method like prevention.

For the *qc* variant, avoidance has only a minimal effect on the success probability, but the mechanism behind it is worth noting. The variant reaches a point where several program instances are able to allocate a qubit but are then unable to schedule entanglement in their allocated time bin because doing so is deemed unsafe. That time bin is skipped while the already held qubit remains in memory and decoheres, as illustrated in Figure 6.5.

The *ql* variant has a very low makespan below 0.1 s and a 100% success probability. Here, Restart is the strategy with the highest makespan, while the other strategies remain similar. The independency from the time bins makes this case similar to entanglement requests without a network schedule. Indeed, the higher makespan is also visible in that case as will be presented in Section 6.3. Still, the general low makespan prevents any significant decoherence of all qubits.

6.3 Without Network Schedule

Running with and without a network schedule can meaningfully impact the timing of entanglement generation. A network schedule can have significant effects on the makespan on a node with few qubits and many running instances. Figure 6.7 shows how running Late Lock with a network schedule causes an especially high increase in makespan for preven-



(a) Fluctuating Needs Success

(b) Fluctuating Needs Makespan

Figure 6.4: Makespan and success of Fluctuating Needs with 4 qubits. It shows each strategy with baselines for three variants. Only request routines (qc), two request routines with the rest local (mix), and only local routines (ql). Prevention for the *qc* in makespan variant follows the same line as the Linear baseline. The *ql* variant has a makespan below 0.1s and 100% success probability across all instances.

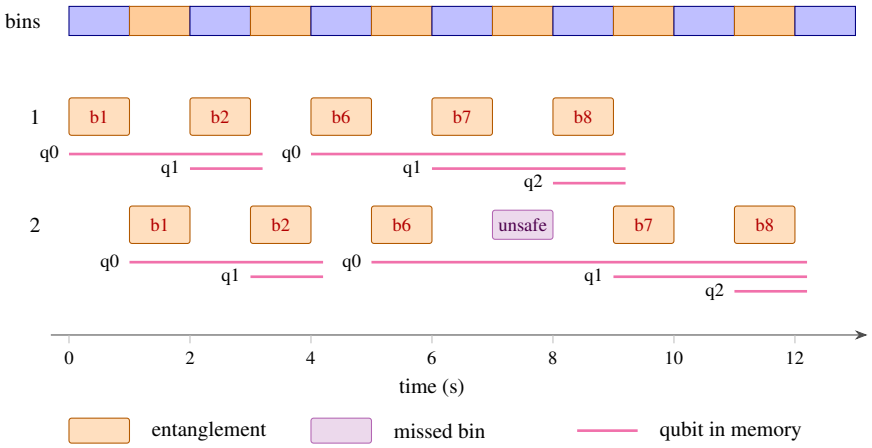


Figure 6.5: Simplified timeline of the *qc* variant of Fluctuating Needs with 2 instances and 4 qubits. Each row is a program instance and each illustrated block is an entanglement request routine, following the program structure with blocks b1, b2, b6, b7, and b8. Local routines are omitted because their time scale is small compared to an entanglement and they are scheduled before the time bin ends. The line below the blocks marks the time a qubit is in memory and dephases. For instance 2 the second qubit of request 4 is unsafe in its next slot, so that time bin is missed and b7 is scheduled to a later bin while the qubit from b6 stays in memory and decoheres. The longer a qubit is held over a time bin, the more noise it accumulates.

tion compared to without it. Restart is also heavily impacted by the network schedule, but its makespan is also high without one. We describe this further in Section 6.4.

Several effects appear when running without a network schedule. First, there are more workloads that do not result in a deadlock. This comes from the fact that fewer program instances run concurrently when no network schedule enforces scheduled entanglement. Second, the makespan of avoidance and prevention is not determined by the number of qubits but only by the number of instances. Both strategies manage to run entanglement sequentially without being delayed by unsafe allocations or because fewer instances run concurrently. Third, we see that Prevent in Figure 6.8 has a higher success probability than Restart and Avoid without a network schedule, especially for scenarios with many instances and many qubits. These cases have the highest concurrency. With a network schedule, the success probability becomes less dependent on the number of qubits.

Figure 6.6a and Figure 6.6b show the success and normalized makespan across workloads. Similar to Late Lock, Restart is the only strategy with a higher makespan and standard deviation. The other strategies are able to generate entanglement as soon as it is available. This makespan penalty for Restart is application dependent. In Late Lock, Restart is penalized both with and without a network schedule, whereas BQC2q and Mixed only show a higher makespan without one. We also see a similar trend to Late Lock for success, where Prevent is consistently higher than Restart and Avoid. Across these applications, prevention leads by 6 ± 1.3 to 14 ± 1.5 percentage points for PingPong, GHZ, and BQC2q, while Teleport is unaffected. For the other applications, the success probabilities for Restart and Avoid are mostly similar, but Avoid has an increased success probability compared to Restart for BQC2q and GHZ.

Subsection 5.5.2 predicted that Avoid and Prevent would benefit the most without a network schedule. The prediction would be refuted if neither strategy gained without a schedule. It held only in part.

Without a network schedule, program instances compete for entanglement slots freely and deadlocks typically occur less frequently. Prevention especially benefits from this because it no longer has to hold qubits across time bins. Restart, by contrast, pays the full cost of regenerating all prior entanglements when a deadlock does occur. The makespan penalty is not visible for PingPong and Teleport since they consistently did not trigger any deadlocks in this scenario. Where the network schedule delayed entanglement to the next time bin and let available local routines run first, its absence meant that entanglement was prioritized earlier, before another instance acquired a qubit through a local routine, resulting in no deadlocks.

The success probabilities also reveal how the strategies differ. Avoidance consistently reaches lower success probabilities than Prevention because of the increased concurrency. Restart suffers from the same effect. Most notably, in the cases where a restart is needed, the success probability dips below avoidance, in contrast to where it does not. This implies that even though the readmission of tasks requires all qubits to be available, it still leads to a high concurrency after restarts that can lower the success probability. Prevention, on the

other hand, is often able to run EPR generations back-to-back. The number of qubits also has a much larger impact on success without a network schedule.

With more qubits, a greater number of program instances can run concurrently. If one instance holds a qubit across the entanglement generation of another instance, the qubit is exposed to additional decoherence from the interleaving tasks. Prevention and the linear baseline are the least affected by this, since they reduce this concurrency. Reducing the number of qubits forces more sequential execution and therefore shorter effective hold times, improving success. This effect is already seen in the baseline of enough qubits, but is also seen throughout deadlock handling strategies, where prevention is able to keep the highest success probability.

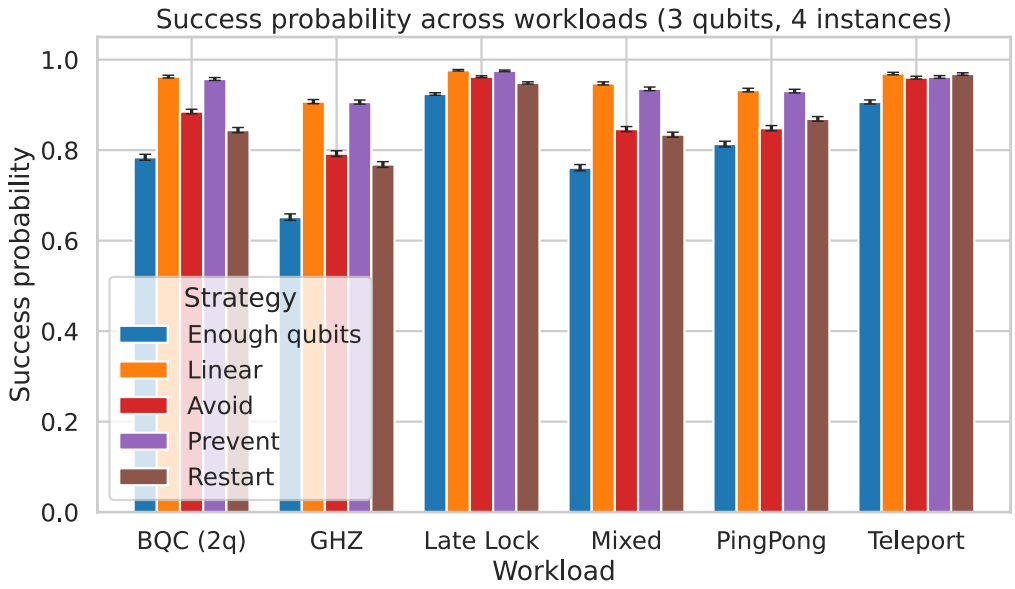
6.4 Late Deadlocks

In workloads where the deadlock happens late in a program instance's runtime, restarting the application is costly. Figure 6.7 shows how Late Lock has a heavily increased makespan under restart reaching from 59% up to 280% over avoidance when a deadlock occurs. For the configurations that do not lead to deadlocks, there is no difference in makespan.

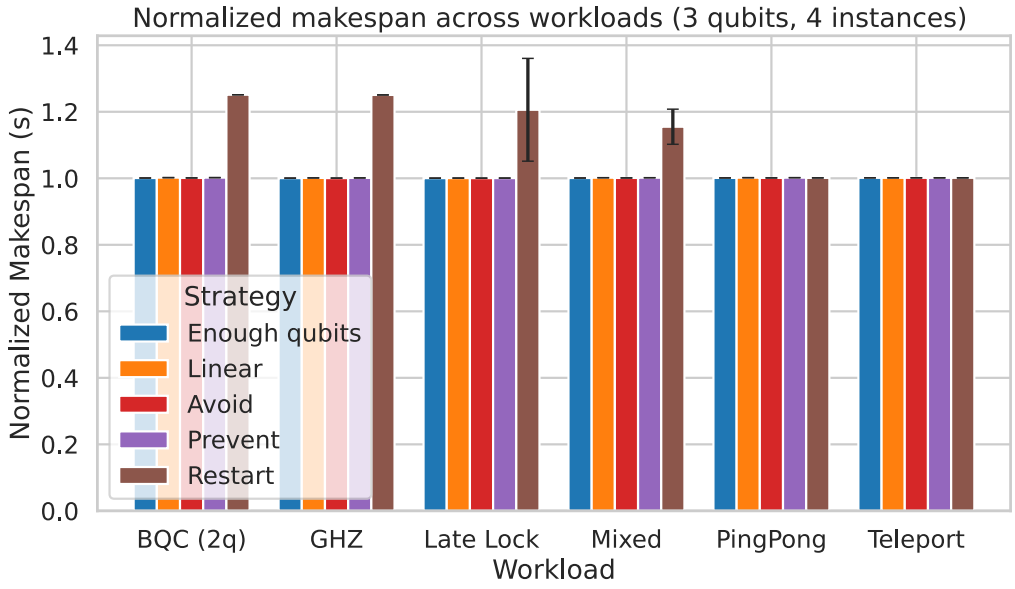
A BQC instance allocating 10 qubits through 10 entanglement generations can also be costly to restart. Figure 6.9 shows the increase in makespan as more instances are added. Where previous smaller workloads have a makespan for Restart closer to Avoid, here it also surpasses prevention.

We predicted in Subsection 5.5.3 that Restart would yield a worse makespan than Avoid and Prevent when the deadlock occurs late in a program instance's runtime, with the penalty growing in the amount of entanglement work discarded and amplifying further under a network schedule. This prediction would be refuted if Restart matched Avoid and Prevent on makespan for late deadlocks. We did not observe this, and the prediction held.

The penalty is driven by the discarded entanglement work. For BQC10q, each restart forces up to ten entanglement generations to be redone before any new measurement can take place, which is why Restart's makespan surpasses even Prevention as instances are added. A large factor here is that each allocated qubit is kept in memory until the last entanglement generation. The decoherence factor for the Deadlock Impact Score increases so rapidly that the longest running instance might be restarted, substantially increasing the makespan. Indeed, the cost of a restart scales with the amount of entanglement already generated before the deadlock occurs, showing that the makespan of Restart does not scale well with larger applications. This also brings up a discussion of whether the Deadlock Impact Score necessarily must always penalize programs where a few qubits have been highly affected by decoherence. We will get back to this in the next chapter.

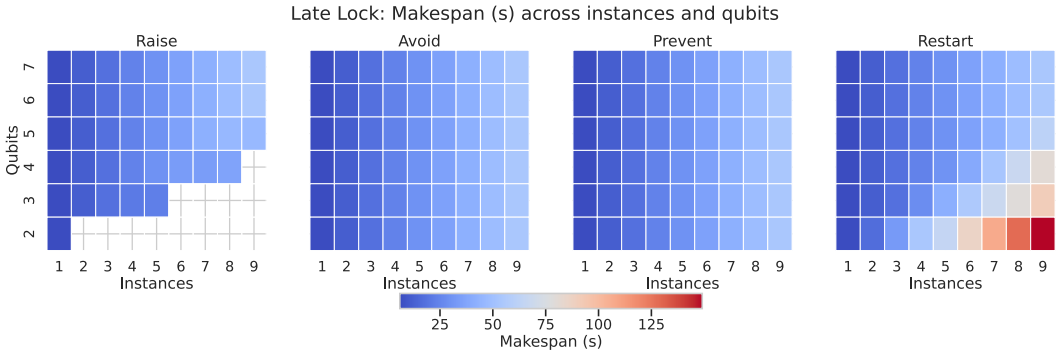


(a) Success probability.

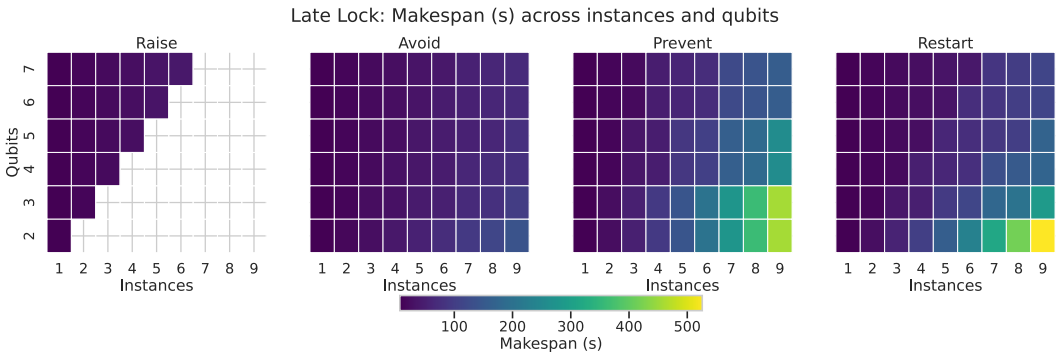


(b) Normalized makespan.

Figure 6.6: Success and normalized makespan across workloads without network schedule.



(a) Without a network schedule.



(b) With a network schedule.

Figure 6.7: Makespan of Late Lock without and with netschedule. Note that the two heatmap rows have different value ranges. Where Raise does not have values present are the cases where deadlocks occurred.

6.5 Deadlock Frequency

The Deadlock Loop workload increases the frequency of entanglement generations and deadlocks without increasing the number of instances. The makespan of a workload with 10 instances of Deadlock Loop is shown in Figure 6.10. A clear increase in both standard deviation and makespan compared to Avoid is shown for Restart as the frequency increases, with Restart's standard deviation rising from near zero to 114 s. Prevent and Linear are not visible as their makespan starts similarly increasing from 1056 s and 2101 s respectively.

Similar to previous results, the success probabilities are within 0.2 ± 0.1 percentage points from each other across all frequencies. The success probability decreases the same for all strategies. This is attributed to the same factor that makes all success probabilities from Section 6.1 very similar.

Subsection 5.5.4 predicted that the best-performing strategy would shift from Restart toward Avoid and then Prevent as the chance of deadlock increased, because the cost of failed restarts should grow with frequency while the cost of waiting under Avoid and Prevent should not. We also predicted that Avoid would hold decohering qubits in memory for

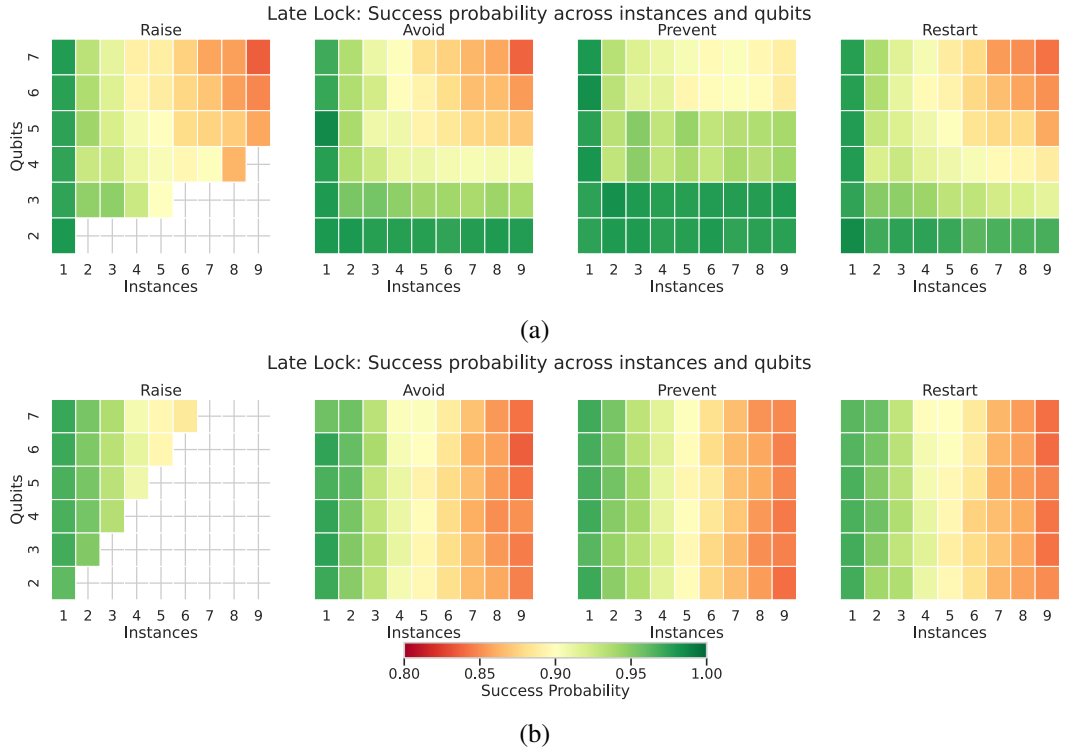


Figure 6.8: Success of Late Lock without and with netschedule.

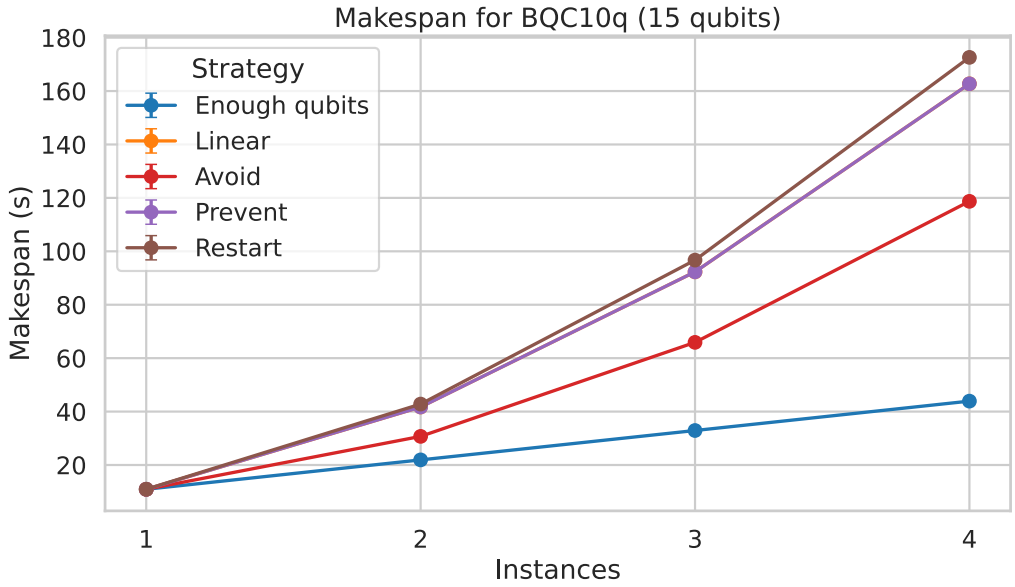


Figure 6.9: Makespan of BQC using 10 qubits executed on a node with 15 qubits increasing instances from 1 to 4. Each qubit, created through entanglement, is kept in memory until the last local routine. A network schedule is used. Linear is overlapped by Prevent.

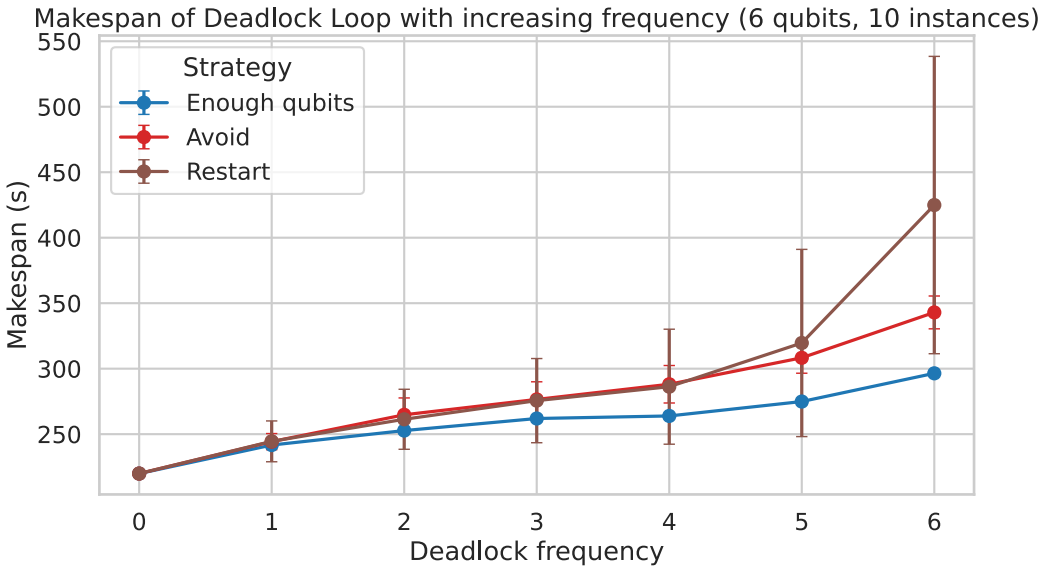


Figure 6.10: Makespan and success of 10 Deadlock Loop instances with a 6 qubit node. The frequency of entanglement generations increases. Linear and Prevent have consistently a higher makespan than what the figure is limited to. For success, Enough Qubits, Prevent, and Restart all mostly overlap.

longer at high frequencies. The prediction held only in part.

Restart's makespan rises with frequency but never reaches the level of Prevent, so the predicted shift toward Prevent was refuted. Why Restart does not exceed Prevent as shown for BQC10q can be attributed to the scenario's structure having a high number of running instances, but also to the Deadlock Impact Score often prioritizing the termination of shorter running instances. The high variance is a product of how many deadlocks occurred and which instances were terminated.

6.6 Entanglement Generation Duration

Figure 6.11 shows how the entanglement generation duration affects the makespan of GHZ. For most strategies it has no noticeable effect on the relative makespan, but for Restart a five second entanglement duration increases the relative makespan more than for the other strategies. Avoid shows no comparable penalty at the durations tested. The success probability was also heavily reduced by the order-of-magnitude increase in entanglement duration, falling from 0.99 at 0.01 s to 0.59 at 1 s and 0.47 at 5 s.

Subsection 5.5.5 predicted that longer entanglement generation durations would widen the penalties under both Restart and Avoid while preserving the ordering. The prediction held for Restart but was refuted for Avoid, which showed no penalty at the durations tested.

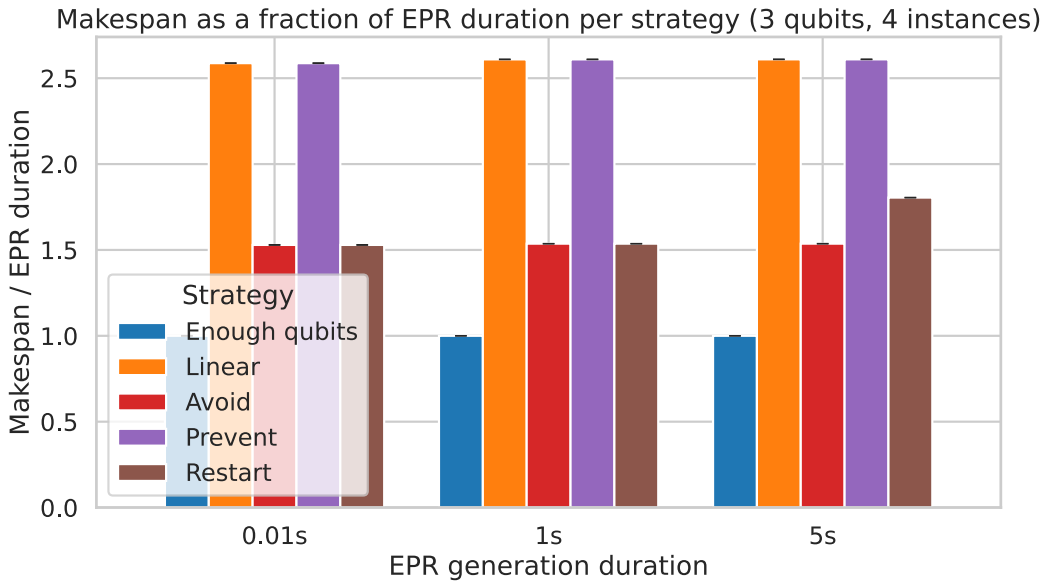


Figure 6.11: Makespan as a fraction of the entanglement duration for GHZ across three different entanglement durations. In order to compare between generation durations, we calculate the makespan as a fraction of the entanglement duration.

As described above, only Restart's relative makespan grows, and only at the 5 s duration. This deviation comes from the deadlock impact score, which at long entanglement durations judges the first running instance degraded enough to terminate over newly scheduled ones. The terminated instance must then restart completely, increasing the duration. This heavy reduction in success probability also gives reason to try varying the duration in smaller steps.

6.7 Longer Bin Length

If we increase the bin length to fit all entanglement generations we see a somewhat similar result for success probability in Figure 6.12a with two main exceptions. First, GHZ has a clear reduction as it is not able to schedule its entanglements within a single time bin for Prevent, Avoid, and Restart, incurring decoherence across the other time bins. Second, we see a reduced success probability for Avoid in PingPong. This follows from avoidance scheduling local routines that allocate qubits before their allocated time bin more often than Prevent and Restart. The effect is also reflected in the QPU execution time.

We see a significant increase in Avoid's QPU execution time for PingPong in Figure 6.12b. Even though the total makespan is less than for Prevent, the concurrency combined with initialized but further unsafe allocations makes the average duration of each instance longer. However, this is not visible for other applications. In the figure, we also distinguish between the time from the last restart and first added task.

Subsection 5.5.6 predicted that longer time bins would reduce the penalty of Restart and Prevent in cases where entanglement demand resolves within a single bin, reducing the frequency of a hold-and-wait condition. The prediction was neither supported nor cleanly refuted, and the results are similar as before. The success rate stays largely the same between strategies except for PingPong as explained.

6.8 Topology Variation

For 1 communication qubit with several memory qubits we see a close success probability between strategies, within less than 0.9 ± 0.5 percentage points for 4 instances at around 97%. However, linear execution increases this to 99% ± 0.1%.

The makespan across increasing memory qubits is plotted in Figure 6.13. While prevention keeps the same makespan, restart and avoidance show equal reductions in their makespan as the number of memory qubits increases. Notably, the standard deviation of the makespan is highest for prevention, but also high when there is only 1 memory qubit for Avoid and Restart of 2.2 s. Meanwhile, the linear baseline has a significantly lower makespan than the deadlock handling strategies for 1 and 2 memory qubits. Enough Qubits baseline has enough communication and memory qubits to generate entanglement at each time bin.

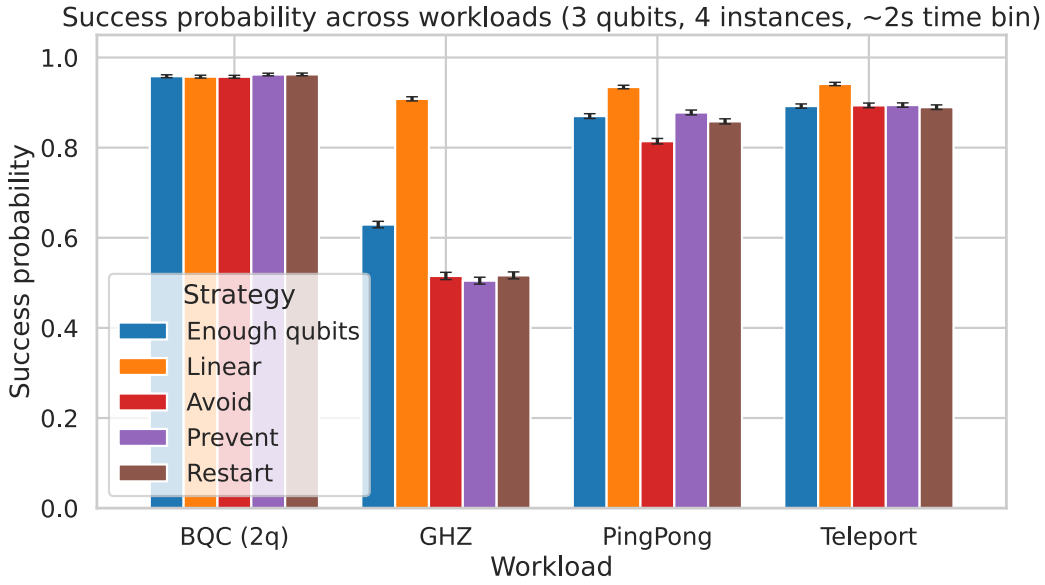
Subsection 5.5.8 predicted that separating qubits into communication and memory roles would make deadlocks more prevalent, raising Restart's makespan above avoidance and penalizing Prevent through its need to hold a communication qubit for the duration of the program. We also predicted that Avoid would exploit this situation better while Prevent kept the highest success probability. The prediction held only in part, since Restart and avoidance showed the same makespan rather than Restart rising above it.

The serialization penalty for Prevention is visible, as it is the only strategy whose makespan does not improve with more communication or memory qubits. However, restart and avoidance have the same makespan.

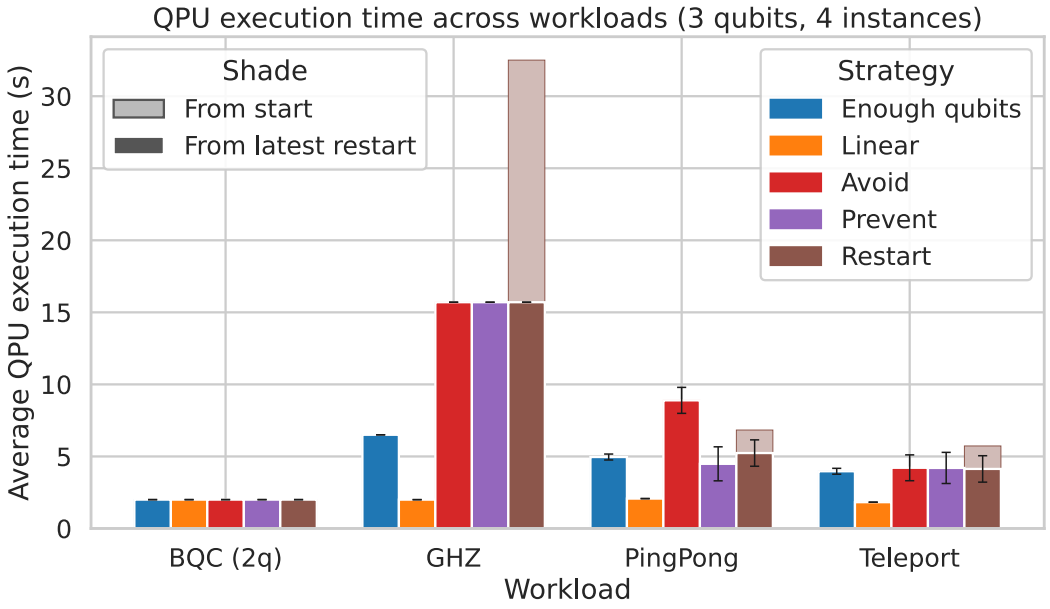
The standard deviation is explained by the initial local routine on two of the program instances. When the first local routine does not match the current time bin, prevention cannot allocate enough qubits for that time bin. This nondeterministically chosen task can therefore increase the makespan. This is also the case for 1 memory qubit where avoidance cannot continue or more likely for a restart to happen.

6.9 Termination Factors

Figure 6.14 shows how the coefficients in Equation 3.1 influence the makespan and success for different applications both with and without a network schedule. There is a slight increase in success for BQC2q and Fluctuating Needs *qc* using the decoherence factor (DF). With a network schedule, DF has a slightly higher makespan. Without it, RF shows a slightly reduced makespan. Meanwhile, RF+DF follows either the makespan or success



(a) Success probability using bin length to fit 2 entanglement generations.



(b) Average QPU execution time for a longer bin length for different workloads. Restart includes the time from start and from the last restart.

Figure 6.12: Success probability and QPU execution time for longer time bins.

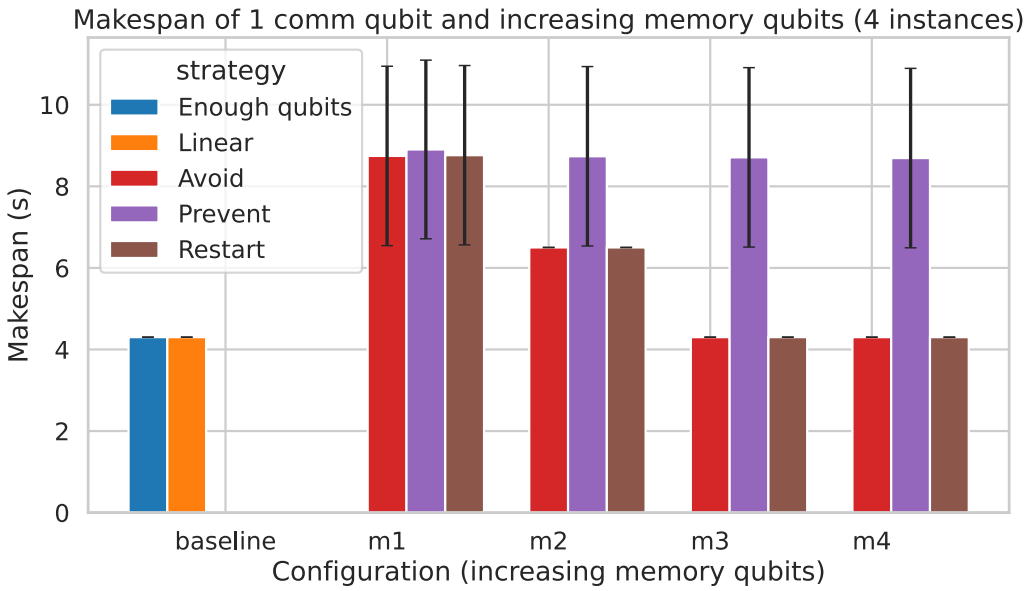


Figure 6.13: Makespan of different topologies ranging from 1 to 4 memory (mem) qubits and 1 communication qubit with 4 instances.

of one of the other factors. As seen before in Section 6.3, the variation in makespan and success depends on whether a network schedule is used.

Subsection 5.5.7 predicted that RF would optimize for makespan, DF would optimize for success probability, and the combination RF+DF would fall between the two single-factor cases. The prediction held only in part.

The slight success increase under DF, together with its higher makespan under a network schedule, is consistent with the predicted DF trade-off, where terminating instances with degraded qubits frees them for fresh use at the cost of additional restart work. The combination RF+DF lands closer to one or the other single factor depending on the configuration, rather than a clean midpoint, because the time at which a deadlock is reached determines which factor dominates the impact score. Whether the choice of coefficients has an effect is dependent on whether a network schedule is used and which metric is of interest.

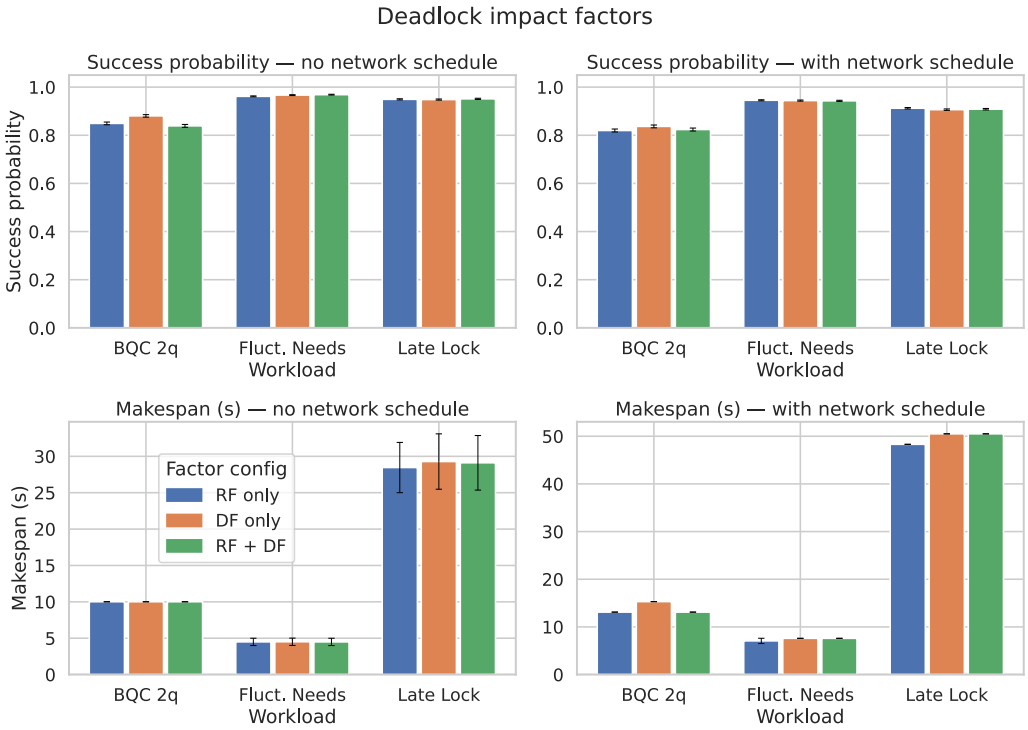


Figure 6.14: Comparison of different deadlock impact scoring across different workloads and configurations. Impact scoring includes runtime factor only (RF), dephasing (DF), and both equally (RF+DF).

Chapter 7

Discussion

We discuss the overall results and how they help us answer the research questions. We have presented results from various applications and configurations, and these configurations change the results in various ways. Our strategies should first be compared with the two baselines.

7.1 Baselines

We predicted that the two baselines would bracket the strategies, with Linear giving the highest makespan and Enough Qubits the lowest. This held only for Enough Qubits, which consistently reached the lowest makespan across workloads. Linear, on the other hand, was not always the upper bound. Prevention sometimes exceeded it, and for some topologies Linear ran with a noticeably lower makespan than the strategies. As previously mentioned, this was caused by a nondeterministically chosen task blocking entanglement generation for a time bin.

The success side of the prediction was refuted. Enough Qubits did not reach a meaningfully lower success probability than the others, as the success probabilities stayed close across both baselines and strategies, typically within about one percentage point with a network schedule. Only Linear showed a clear advantage, but only for applications allocating a qubit with a local routine and keeping it until an entanglement generation.

Enough Qubits relies on the idea that hardware has enough qubits such that resources are never scarce. This is however unrealistic for near-term quantum computers. The Linear baseline is on the other hand fully capable of preventing deadlocks. Still, its inability to interleave programs makes it scale poorly as the contention increases.

7.2 Answering Research Questions

From the results and discussions presented, we try to clearly answer the research questions.

7.2.1 RQ1: Strategies for low- and high-contention

Across low- and high-contention workloads with a network schedule enabling concurrency, Avoid regularly exhibits a makespan similar to Restart and considerably faster than Prevent. We see often that when Restart does not reach a deadlock, Avoid is similarly able to allocate without finding unsafe allocations. When a deadlock does occur for Restart, Avoid correctly defers allocations. This leads to a similar makespan between these two strategies. This might be because reaching an unsafe state in our workloads leads to a deadlock more often than not. However, this assumption breaks down when the restart occurs late in an instance's runtime. Prevention, by contrast, is slower in both low- and high-contention situations.

The choice of strategy has little effect on the success probability for most workloads, where the number of instances is instead the dominant factor. With a network schedule, avoidance is therefore the better choice across the workloads we tested.

We note that the reasoning for the best strategy without a network schedule changes considerably. The makespans of Prevent and Avoid are often identical, and the increased success probability for Prevent makes it more desirable. Meanwhile, the increased makespan of restarting entanglement is not reduced to the same level of the other strategies. Since Qoala assumes the existence of a network controller scheduling time bins [36], we stress that the best strategy should depend on one, but the difference with and without one shows the relevancy of including it. It highlights how enabling a network to consistently use each possibility for entanglement can change which strategy performs best.

7.2.2 RQ2: Scaling to network and node

Avoidance scales much better than prevention and restart. As the size of programs grow with more entanglements, so does the contention, and as we have already seen, avoidance consistently maintains the lowest makespan among the strategies. Restart, by contrast, scales poorly with program size, as each restart discards more entanglement work the larger an application becomes. This is additionally worsened by the increased variance depending on which instance is terminated. Similarly, since Prevent allocates all qubits at instantiation, increasing the number of qubits it needs at one time noticeably decreases the possible concurrency, in turn increasing the makespan.

We also evaluated scenarios changing the length of time bins to fit several entanglement generations, increasing the duration of each generation and changing the network topology. In both cases, we saw no meaningful difference between avoidance, prevention, and restarting. Again, success probability was often similar between deadlock handling strategies, with prevention often having the highest makespan. This shows how our results hold across changes to the hardware, distance between nodes, and specific changes to the network schedule. This has to be seen in relation to our results without a network schedule, which are meaningfully different.

7.2.3 RQ3: Deadlock Impact Score

We show that the coefficients of the Deadlock Impact Score can be tuned to prioritize either the success probability or the makespan, and that combining the decoherence and runtime factors strikes a balance between the two. With the importance of reduced noise, only including the decoherence term might be beneficial without a network schedule. However, with a network schedule where the success probabilities are mostly similar, including the runtime factor, or using it exclusively, seems to be a better option. The continued risk of cyclic restarts suggests that the runtime factor should be weighted more heavily, or that the equation should be changed such that passing the T_2 time does not exponentially increase the score. The score should perhaps also reflect that restarting a long-running instance is more costly than restarting one in which only a single qubit, out of potentially several, has been considerably affected by noise.

7.3 Limitations and Untested Cases

The evaluations are conducted under specific assumptions that limit generalization. We designed and evaluated scenarios based on our expectations. Other configurations and applications fall outside these expectations, for example reducing the entanglement generation duration while inversely extending each time bin. This can be further tested with a large application benchmark and randomly generated configuration, but the number of possible testable combinations makes this difficult and complex to evaluate. A better solution is in that case to evaluate on a fixed set of configuration constraints, for example based on the current hardware, and define the best strategy at that point. For this thesis, that could for example be by using the existing topology presented by [8].

We also stress that other metrics can be relevant to measure. For example, utilization of resources with queue length is a relevant one to see the impact of each strategy. These two should be paired up because high utilization alone does not necessarily imply optimal resource management or higher success probabilities. We have prioritized measuring the success probability and makespan as it was expected to be the more important factor when choosing a strategy. Similar to early classical operating systems, having a high utilization was beneficial in order to use all the available hardware [3], which can also be relevant for quantum systems. However, trading successful execution for higher utilization might not be a worthy trade-off for reliability.

There is also room to further evaluate the effect of different factors of deadlock impact score or variations of the formula terms. This could for example be done by converting the score to a cost as discussed previously. It is also possible to evaluate different coefficients. However, since the factors were tested on two extremes, any variations of these factors are expected to fall within the presented results.

Deadlock conditions can also span the network, but this thesis does not address that case. For example, two communicating nodes might each be able to schedule only one program instance that requires entanglement generation. If these instances do not share a session, no node is able to continue. This can sometimes be solved with time bins, but deadlock

handling such as prevention and avoidance complicates the situation further. This can happen, for example, because a nondeterministically chosen instance allocates all its possibly needed resources, or because other program instances cannot run while their tasks are unsafe. This opens the discussion to look into synchronization of safe executable tasks across nodes.

These limitations do not undermine the results of this thesis, but rather point to several different directions for future work and enhancements.

Chapter 8

Conclusion

We conclude our work by presenting future work and summarizing our contributions and results, before reflecting on the possible disciplinary and societal impact.

8.1 Future Work

Our research is merely a step in exploring ways to solve deadlocks in multiprogramming for quantum computing. There are several opportunities for future work to enhance our approach, extend its capabilities, and explore implications of different techniques.

8.1.1 Enhanced Deadlock Strategies

There are several enhancements to the implemented deadlock strategies that can be explored. Restarting programs is a costly operation, especially when programs need to reschedule several entanglement generations. These restarts are also distributed over a network, where communication to restart adds to the cost, especially in cases like GHZ and BQC where several nodes need to communicate. Allowing for rollback to a previous state after beginning can heavily reduce the effect of restarts on larger applications. However, this requires synchronization between nodes to ensure that the latest possible state can be found, one where no qubit state is lost.

There are also techniques to allow for more concurrency using avoidance [27]. By tracking the requests and releases of qubits through program analysis, resources could be allocated where the current avoidance system would not. Indeed, this requires more complex analysis for unpredictable programs that include loops and conditionals.

8.1.2 Hardware

The Qoala runtime, and with it our contributions, has not yet been realized on quantum or classical hardware. Simultaneously, simulations cannot capture all parts of the system even though they are based on configurations from hardware demonstrated in research. Future

work still needs to realize and demonstrate these methods on real hardware. How the hardware and system evolve can change which of the presented results are relevant, as we have demonstrated. The scenarios indicate the impact in isolated environments because evaluating all possible combinations of configurations becomes infeasible. Understanding where the hardware is headed becomes vital to determining the strategy to adopt.

8.2 Summary

In this thesis, we started by describing the need for reliable applications running on quantum network nodes. We then provided background on the existing literature and explained the gap this thesis fits into. A methodology, implementation, and evaluation plan were then presented before relevant results were shown and discussed.

Three strategies, prevention, avoidance, and restarting, were designed on the basis of the literature to allow running applications that were previously not possible without using linear precedence constraints or having enough qubits for all executed tasks. Prevention through allocate-all, release-all removes the hold-and-wait condition. Avoidance preserves more concurrency than prevention and avoids the makespan penalty of restart for late deadlocks. Restart enables maximum concurrency until a deadlock occurs, after which the entire program has to be rescheduled leading to a considerable increase in makespan when deadlocks happen later in a program's execution.

We show that success probabilities are often similar across workloads and strategies with a network schedule, typically within about one percentage point, but that avoidance in particular ends up with a reduced makespan. Although restarting entanglement is costly, it still resulted in a lower makespan than prevention for most scenarios. Because of the similar results for success probability, the consistent lower makespan of avoidance suggests that it is the better option with a network schedule.

Without a network schedule, however, the success probability varied more between strategies, with prevention consistently performing better. In terms of makespan, prevention had a completion time similar to avoidance while restarting entanglement became more costly in comparison. Because successful execution is an important factor to execute quantum applications, prevention is in this case the better option.

This ordering of strategies was mostly similar across other configurations of the networks and the node hardware.

We also show how our Deadlock Impact Score is able to prioritize the reduction of either runtime or decoherence by balancing coefficients. The increased success probability from prioritizing decoherence is only shown without a network schedule. With a network schedule, which Qoala assumes, success probabilities are roughly equal and prioritizing the runtime factor is therefore preferred.

The impact of our work opens the door to run applications in Qoala that were previously not possible without linear execution or expecting enough qubits to be available. On a broader scale, our work contributes to the evaluation of how deadlocks impact multiprogramming

in quantum computing in general, especially for quantum internet applications requiring entanglement.

Our work also extends to the entire field of computer science. Indeed, there are many similarities between the current state of quantum computing and the early days of classical computing. Deadlocks have been extensively studied in classical computing. However, there is not a universal solution. Although our work focuses on quantum computing, the implemented techniques can be adapted to classical computing and domains that have similar constraints of degrading resources and tasks of significantly different time scales.

With the investments and research in quantum computing, we expect our work to become more and more relevant as quantum hardware improves and unreliable concurrently running applications become a bottleneck for the quantum internet.

Bibliography

[1] I. B. Abdallah and H. A. ElMaraghy. Deadlock prevention and avoidance in FMS: A Petri net based approach. *International Journal of Advanced Manufacturing Technology*, 14(10):704–715, 1998. ISSN 02683768. doi: 10.1007/BF01438223/METRICS. URL https://link.springer.com/article/10.1007/BF01438223.

[2] Antonio Abelem, Don Towsley, and Gayane Vardoyan. Quantum internet: The future of internetworking. *Minicursos do XXXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 48–90, 4 2023. doi: 10.5753/sbc.5033.7.2. URL http://arxiv.org/abs/2305.00598http://dx.doi.org/10.5753/sbc.5033.7.2.

[3] Thomas Anderson and Mike Dahlin. *Operating systems: Principles and practice, volume ii: Concurrency*. Recursive Books, Ltd. http://recursivebooks. com, 2015. ISBN 978-0985673543. URL https://www.kea.nu/files/textbooks/ospp/osppv2.pdf#page=169.

[4] Zbignew A. Banaszak and Bruce H. Krogh. Deadlock avoidance in flexible manufacturing systems with concurrently competing process flows. *IEEE Transactions on Robotics and Automation*, 6:724–734, 1990. ISSN 1042296X. doi: 10.1109/70.63273. URL https://ieeexplore.ieee.org/abstract/document/63273.

[5] David Barral, F. Javier Cardama, Guillermo Díaz-Camacho, Daniel Faílde, Iago F. Llovo, Mariamo Mussa-Juane, Jorge Vázquez-Pérez, Juan Villasuso, César Piñeiro, Natalia Costas, Juan C. Pichel, Tomás F. Pena, and Andrés Gómez. Review of distributed quantum computing: From single qpu to high performance quantum computing. *Computer Science Review*, 57:100747, 8 2025. ISSN 1574-0137. doi: 10.1016/J.COSREV.2025.100747. URL https://www-sciencedirect-com.tudelft.idm.oclc.org/science/article/pii/S1574013725000231?ref=pdf_download&fr=RR-2&rr=9f8fb69ea96cf5e0#page=6.46.

[6] Charles H. Bennett and Gilles Brassard. Quantum cryptography: Public key distribution and coin tossing. *Theoretical Computer Science*, 560:7–11, 12 2014. ISSN

0304-3975. doi: 10.1016/J.TCS.2014.05.025. URL https://www-sciencedirect-com.tudelft.idm.oclc.org/science/article/pii/S0304397514004241.

[7] Charles H. Bennett, David P. DiVincenzo, Peter W. Shor, John A. Smolin, Barbara M. Terhal, and William K. Wootters. Remote state preparation. *Physical Review Letters*, 87:077902, 7 2001. ISSN 0031-9007. doi: 10.1103/PhysRevLett.87.077902.

[8] C. E. Bradley, J. Randall, M. H. Abobeih, R. C. Berrevoets, M. J. Degen, M. A. Bakker, M. Markham, D. J. Twitchen, and T. H. Taminiau. A ten-qubit solid-state spin register with quantum memory up to one minute. *Physical Review X*, 9:031045, 9 2019. ISSN 21603308. doi: 10.1103/PhysRevX.9.031045. URL https://journals.aps.org/prx/abstract/10.1103/PhysRevX.9.031045.

[9] Anne Broadbent, Joseph Fitzsimons, and Elham Kashefi. Universal blind quantum computation. *Proceedings - Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 517–526, 2009. ISSN 02725428. doi: 10.1109/FOCS.2009.36. URL https://ieeexplore.ieee.org/abstract/document/5438603.

[10] C. D. Bruzewicz, R. McConnell, J. Stuart, J. M. Sage, and J. Chiaverini. Dual-species, multi-qubit logic primitives for ca+/sr+ trapped-ion crystals. *npj Quantum Information 2019 5:1*, 5:102–, 11 2019. ISSN 2056-6387. doi: 10.1038/s41534-019-0218-z. URL https://www.nature.com/articles/s41534-019-0218-z.

[11] Andrew M. Childs. Secure assisted quantum computation. *Quantum Information and Computation*, 5, 7 2005. doi: 10.26421/QIC5.6. URL http://arxiv.org/abs/quant-ph/0111046http://dx.doi.org/10.26421/QIC5.6.

[12] Edward G Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3:67–78, 1971.

[13] Axel Dahlberg, Matthew Skrzypczyk, Tim Coopmans, Leon Wubben, Filip Rozpdek, Matteo Pompili, Arian Stolk, Przemyslaw Pawelczak, Robert Knegjens, Julio De Oliveira Filho, Ronald Hanson, and Stephanie Wehner. A link layer protocol for quantum networks. *SIGCOMM 2019 - Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication*, pages 159–173, 8 2019. doi: 10.1145/3341302.3342070;TOPIC:TOPIC:CONFERENCE-COLLECTIONS. URL https://dl-acm-org.tudelft.idm.oclc.org/doi/pdf/10.1145/3341302.3342070.

[14] Edsger Dijkstra. Een algorithme ter voorkoming van de dodelijke omarming. URL https://www.cs.utexas.edu/~EWD/ewd01xx/EWD108.PDF.

[15] Edsger Wybe Dijkstra. *Selected writings on computing: a personal perspective*. Springer-Verlag, 1982. ISBN 0387906525.

[16] C. Delle Donne, M. Iuliano, B. van der Vecht, G. M. Ferreira, H. Jirovská, T. J.W. van der Steenhoven, A. Dahlberg, M. Skrzypczyk, D. Fioretto, M. Teller, P. Filippov, A. R.P. Montblanch, J. Fischer, H. B. van Ommen, N. Demetriou, D. Leichtle,

L. Music, H. Ollivier, I. te Raa, W. Kozlowski, T. H. Taminiau, P. Pawełczak, T. E. Northup, R. Hanson, and S. Wehner. An operating system for executing applications on quantum network nodes. *Nature 2025 639:8054*, 639:321–328, 3 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-08704-w. URL https://www.nature.com/articles/s41586-025-08704-w.

[17] Ahmed K Elmagarmid. A survey of distributed deadlock detection algorithms. 1986.

[18] Scarlett Gauthier, Gayane Vardoyan, and Stephanie Wehner. An architecture for control of entanglement generation switches in quantum networks. *IEEE Transactions on Quantum Engineering*, 4:1–17, 2023. ISSN 26891808. doi: 10.1109/TQE.2023.3320047. URL https://ieeexplore-ieee-org.tudelft.idm.oclc.org/abstract/document/10265162.

[19] A Nico Habermann. Prevention of system deadlocks. *Communications of the ACM*, 12:373–ff, 1969.

[20] Prasanna Hambarde, Rachit Varma, and Shivani Jha. The survey of real time operating system: Rtos. 2014. doi: 10.1109/ICESC.2014.15. URL https://d1wqtxts1xzle7.cloudfront.net/102349756/F1156-English-IranArze-libre.pdf?1684389828=&response-content-disposition=inline%3B+filename%3DThe_Survey_of_Real_Time_Operating_System.pdf&Expires=1778273175&Signature=HQ3pqi-5hJ05lZnSZnS~GmSeP7Rvnex0dP4a6IVh43UXORFxyolcamGFyOoU-nRBcGyFDovHqYfEu7pnULmaaSvIJEdOdXTPz0Dg08RIBenp4JQOE~tkM5pDsXYT-S38D1YPjA9muK8mgxmRrvs6s5xS5Ov2RXNIX36jQe95UC74oqtgrk6-PY3IsJcWGiUYTPyc~uoNO85YvycwAdRqRXgxloMg-pRINtGOITIad4p98JSIo-oNmqZmatO76mj~0vxsKLlCv5mxGQvJm9l0O8rIZhx.

[21] Richard C Holt. Some deadlock properties of computer systems. *ACM Computing Surveys (CSUR)*, 4:179–196, 1972.

[22] I. V. Inlek, C. Crocker, M. Lichtman, K. Sosnova, and C. Monroe. Multispecies trapped-ion node for quantum networking. *Physical Review Letters*, 118:250502, 6 2017. ISSN 10797114. doi: 10.1103/PhysRevLett.118.250502. URL https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.118.250502.

[23] Sreekaanth S Isloor and T Anthony Marsland. The deadlock problem: An overview. *Computer*, 13:58–78, 1980. URL https://webdocs.cs.ualberta.ca/~tony/OldPapers/dead.pdf.

[24] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. page 295, 2008. URL https://www.usenix.org/legacy/events/osdi08/tech/full_papers/jula/jula.pdf.

[25] Barkaodl Kamel and Imed Ben Abdallah. A deadlock prevention method for a class of fms. 1995. URL https://ieeexplore-ieee-org.tudelft.idm.oclc.org/stamp/stamp.jsp?tp=&arnumber=538436.

[26] Edgar Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, 19, 1 1987. URL https://ics.uci.edu/~cs223/papers/p303-knapp.pdf#page=13.08.

[27] S-D Lang. An extended banker's algorithm for deadlock avoidance. *IEEE Transactions on software engineering*, 25:428–432, 2002.

[28] John Lee. *Hardware/Software Deadlock Avoidance for Multiprocessor Multiresource System-on-a-Chip*. PhD thesis, 12 2004. URL https://www.researchgate.net/publication/27522135_HardwareSoftware_Deadlock_Avoidance_for_Multiprocessor_Multiresource_System-on-a-Chip.

[29] Hoi-Kwong Lo and Yi Zhao. Quantum cryptography. *arXiv preprint arXiv:0803.2507*, 2008.

[30] Michael A. Nielsen and Isaac L. Chuang. Quantum computation and quantum information: 10th anniversary edition. *Quantum Computation and Quantum Information*, 12 2010. doi: 10.1017/CBO9780511976667.

[31] Samuel Oslovich, Bart van der Vecht, and Stephanie Wehner. Compilation strategies for quantum network programs using qoala. pages 731–741, 5 2025. doi: 10.1109/qce65121.2025.00084. URL https://arxiv.org/pdf/2505.06162.

[32] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies - a safe method to survive software failures. *Proceedings of the 20th ACM Symposium on Operating Systems Principles, SOSP 2005*, pages 235–248, 2005. doi: 10.1145/1095810.1095833;CSUBTYPE:STRING:CONFERENCE. URL https://dl-acm-org.tudelft.idm.oclc.org/doi/pdf/10.1145/1095810.1095833.

[33] Spiridon A. Reveliotis, Mark A. Lawley, and Placid M. Ferreira. Polynomial-complexity deadlock avoidance policies for sequential resource allocation systems. *IEEE Transactions on Automatic Control*, 42:1344–1357, 1997. ISSN 00189286. doi: 10.1109/9.633824. URL https://ieeexplore.ieee.org/abstract/document/633824.

[34] Paul J. Schweitzer. Deadlock avoidance in store-and-forward networks-i: Store-and-forward deadlock. *IEEE Transactions on Communications*, 28:345–354, 1980. ISSN 00906778. doi: 10.1109/TCOM.1980.1094666. URL https://ieeexplore.ieee.org/abstract/document/1094666.

[35] John A Stankovic and R Rajkumar. Real-time operating systems. 28, 2004. URL https://www.cse.wustl.edu/~lu/cse467s/papers/rtos.pdf.

[36] Bart van der Vecht, Atak Talay Yücel, Hana Jirovská, and Stephanie Wehner. Qoala: an application execution environment for quantum internet nodes. *arXiv preprint arXiv:2502.17296*, 2 2025. URL http://arxiv.org/abs/2502.17296.

[37] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott Mahlke. Design and implementation gadara: Dynamic deadlock avoidance for multithreaded programs. In *USENIX Association 8th USENIX Symposium on Operating Systems*, 2008. URL https://www.usenix.org/legacy/event/osdi08/tech/full_papers/wang/wang.pdf.

[38] Stephanie Wehner, David Elkouss, and Ronald Hanson. Quantum internet: A vision for the road ahead. *Science*, 362, 10 2018. ISSN 10959203. doi: 10.1126/SCIENCE.AAM9288;PAGE:STRING:ARTICLE/CHAPTER. URL /doi/pdf/10.1126/science.aam9288?download=true.

[39] Ke Yi Xing, Bao Sheng Hu, and Hao Xun Chen. Deadlock avoidance policy for petri-net modeling of flexible manufacturing systems with shared resources. *IEEE Transactions on Automatic Control*, 41:289–295, 1996. ISSN 00189286. doi: 10.1109/9.481550. URL https://ieeexplore.ieee.org/abstract/document/481550.

[40] Gang Xu and Zhiming Wu. Deadlock avoidance based on banker's algorithm for fms. *European Control Conference, ECC 2003*, pages 2370–2377, 4 2003. doi: 10.23919/ECC.2003.7085321. URL https://ieeexplore.ieee.org/abstract/document/7085321.

[41] Anocha Yimsiriwattana and Samuel J. Lomonaco Jr. Distributed quantum computing: a distributed shor algorithm. *https://doi-org.tudelft.idm.oclc.org/10.1117/12.546504*, 5436:360–372, 8 2004. ISSN 0277786X. doi: 10.1117/12.546504. URL https://www-spiedigitallibrary-org.tudelft.idm.oclc.org/conference-proceedings-of-spie/5436/0000/Distributed-quantum-computing-a-distributed-Shor-algorithm/10.1117/12.546504.fullhttps://www-spiedigitallibrary-org.tudelft.idm.oclc.org/conference-proceedings-of-spie/5436/0000/Distributed-quantum-computing-a-distributed-Shor-algorithm/10.1117/12.546504.short.

[42] Hezi Zhang, Yiran Xu, Haotian Hu, Keyi Yin, Hassan Shapourian, Cisco Quantum Lab San Jose, Usa Jiapeng Zhao, Usa Ramana Rao Kompella, Usa Reza Nejabati, and Usa Yufei Ding. Optimizing quantum communication for quantum data centers with reconfigurable networks. *Proceedings of The 52nd IEEE/ACM International Symposium on Computer Architecture ('ISCA 2025')*, 1, 2024.

[43] Sen Zhang, Lingjun Xiong, Yipei Liu, Brian L Mark, Lei Yang, Zebo Yang, and Weiwen Jiang. An end-to-end distributed dantum circuit simulator. 2025.

[44] Most Fatematuz Zohora, Fahiba Farhin, and M Shamim Kaiser. Dbdaa: A real-time approach to dynamic banker's deadlock avoidance algorithm with optimized time complexity. *PloS one*, 19:e0310807, 2024.

Appendix A

Application Diagrams

This section contains circuit diagrams of the evaluated applications within the Qoala simulator. Each lane shows the sequential steps of one program per node. Each program has lines representing qubits affected by initialization, EPR creation, gates, and measurements. Tall *Create EPR* boxes span the two nodes that share an entangled pair, and double lines carry classical bits. Rotation boxes labelled R_x or R_y rotate the qubit about the X or Y axis, an H box is a Hadamard gate, a *Corr* box applies a measurement-dependent correction received from the partner, and a small label under a measurement box gives the measurement basis. Shaded *Meas* boxes are the measured outcomes that define the success metric. These diagrams are inspired by or taken from diagrams found in [36]. The motivation behind the applications is described in Section 5.4.

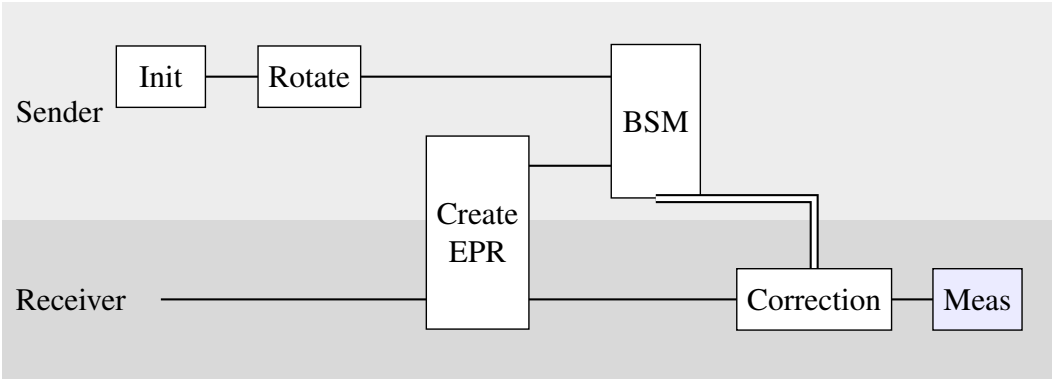


Figure A.1: Teleport. The sender prepares a qubit, generates an EPR pair with the receiver, performs a Bell-state measurement (BSM), and sends the correction bits. The receiver applies the corrections and takes the single shaded success measurement, in the basis matching the prepared state.

Fluctuating Needs comes in three variants that share the same resource pattern but differ in how each request is fulfilled. In all three the number of held qubits rises to 2, falls to 0, and rises again to its peak of 3, the non-monotone pattern shown by the count below each lane. Every held qubit is rotated by R_x at request time and by a second R_x at release time,

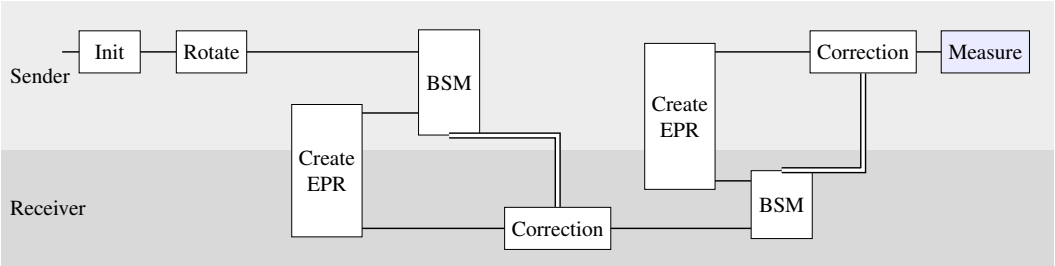


Figure A.2: Ping-Pong. The state is teleported from the sender to the receiver and immediately back, using two EPR pairs and two rounds of Bell-state measurement and correction. The sender takes the single shaded success measurement on the returned qubit, in the basis matching the prepared state.

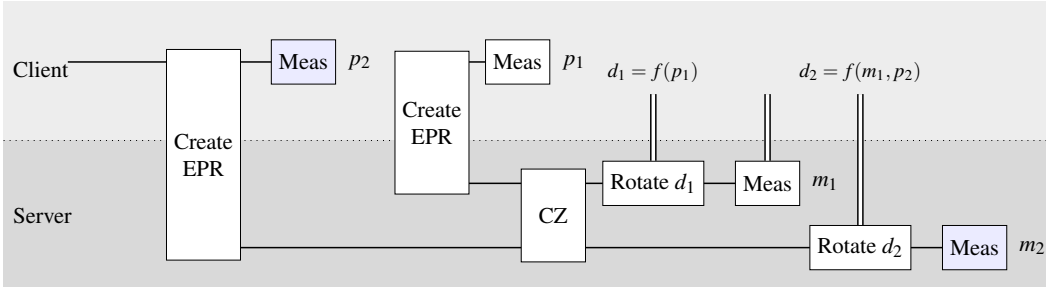


Figure A.3: BQC (two-qubit variant). The client generates two EPR pairs and measures its qubit of each. The server keeps both qubits, applies a `CPHASE`, and runs two measurement rounds, each taken in the basis set by the preceding client-supplied rotation angle. Classical messages cross the lane divider. Success compares the two shaded outcomes, the client's first measurement and the server's second.

so the two $\pi/2$ rotations sum to π and a fully coherent qubit reaches $|1\rangle$. Each release then measures in the Z basis, and the five shaded measurements are the success measurements, with outcome 1 expected. Each qubit's wire runs from its request to its release, so the wire length shows how long the qubit decohered while held.

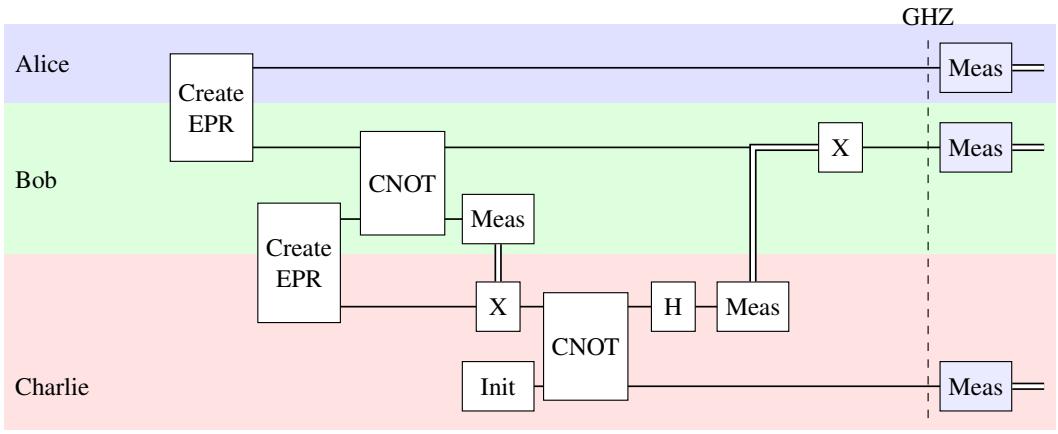


Figure A.4: GHZ. Bob (the hub) generates an EPR pair with Alice and another with Charlie, entangles its qubits with a CNOT, measures one, and sends a correction bit to Charlie. Charlie corrects, fuses a local qubit, measures, and returns a correction bit to Bob. All three nodes then measure their remaining qubit in the X basis; success is the even parity of these three shaded outcomes.

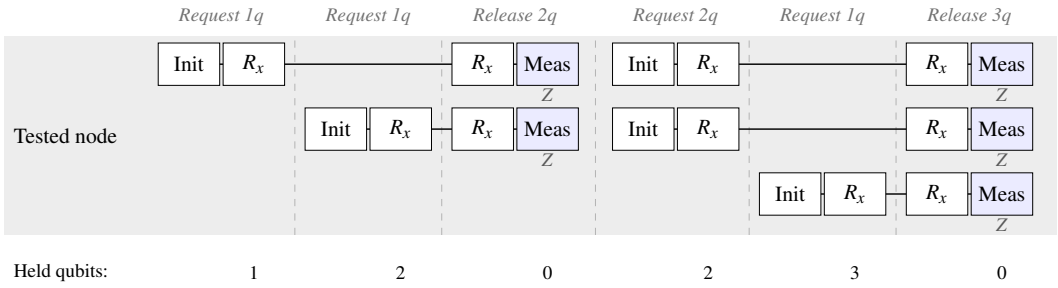


Figure A.5: Fluctuating Needs, *ql* variant. The program runs entirely on the tested node with no entanglement partner. Every request is a local allocation, so each held qubit is prepared by *Init* followed by the first R_x. A release applies the second R_x and the shaded Z-basis measurement, giving five success measurements per instance.

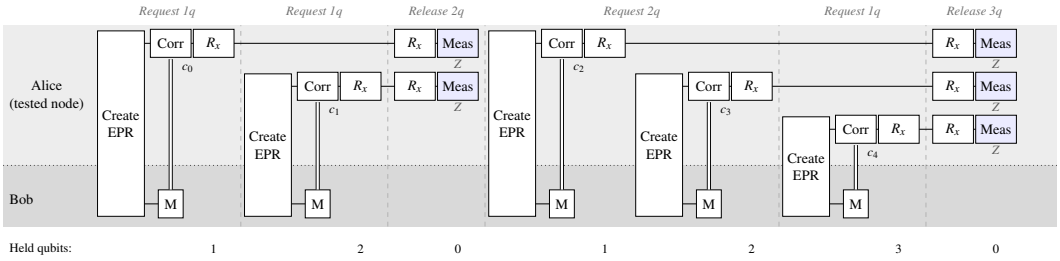


Figure A.6: Fluctuating Needs, *qc* variant. Every request is a Create EPR with Bob, who measures his half and returns the correction bit c_i. Alice applies that correction (*Corr*) and the first R_x, bringing the entangled qubit into the known state on which the rotations act. A release applies the second R_x and the shaded Z-basis measurement. Because each held qubit was entangled rather than locally prepared, its state is exposed to the entanglement-generation latency before the rotations.

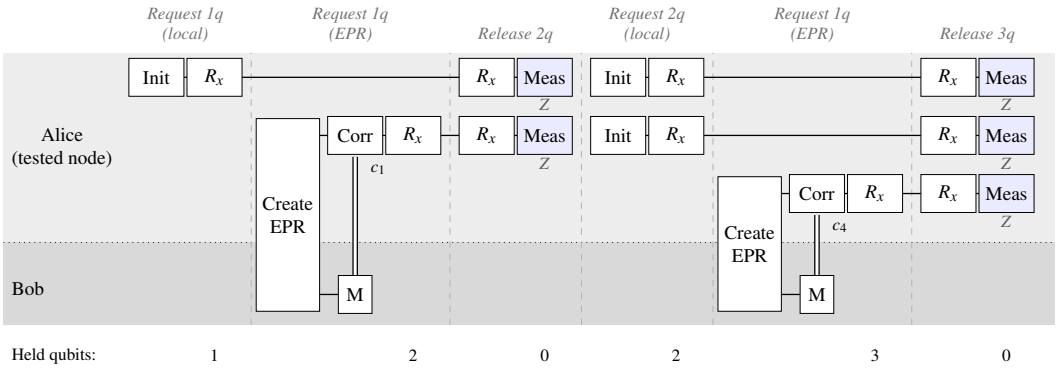


Figure A.7: Fluctuating Needs, *mix* variant. Only the second and fifth requests use entanglement (Create EPR, *Corr*, then the first R_x); the remaining requests are local allocations (*Init* then the first R_x). As in the other variants, a release applies the second R_x and the shaded Z-basis measurement, giving five success measurements per instance.

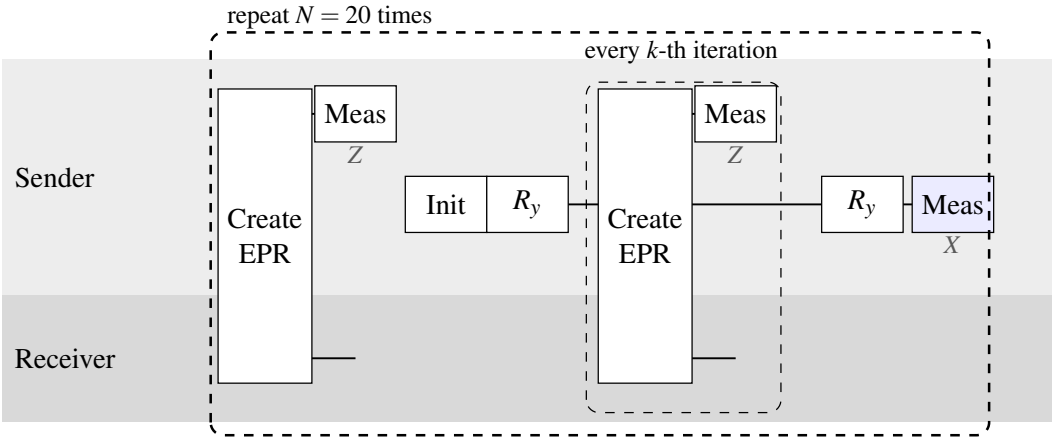


Figure A.8: Deadlock Loop. Each of the $N = 20$ iterations generates and releases an EPR pair and prepares a local qubit. Every k-th iteration runs an additional EPR generation while the local qubit is still held, raising the qubit demand and driving contention into the deadlock regime. The frequency parameter f sets $k = \lfloor N/(f+1) \rfloor$. Each EPR qubit is released by a plain Z-basis measurement. The local qubit is prepared in $|+\rangle$ (*Init* then R_y) and released by an X-basis measurement (R_y then *Meas*, expect 0, shaded); its wire spans the extra allocation, so its length shows how long it decohered while held.

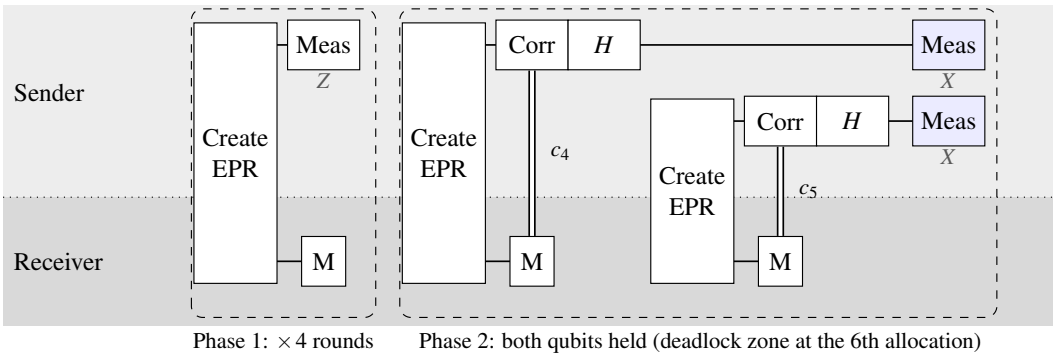


Figure A.9: Late Lock. Phase 1 generates and releases four EPR pairs one at a time, measuring each immediately in the Z basis and holding no qubit between rounds. Phase 2 generates a fifth and a sixth EPR pair and holds both qubits simultaneously; for each, the receiver returns a correction bit (c_4, c_5) that Alice applies (*Corr*) before an H that brings the qubit into $|+\rangle$. On a two-qubit node a deadlock can occur at the sixth allocation when two concurrent instances each already hold one qubit. Both held qubits are released by the two shaded X-basis measurements (expect 0); the long wires from their Create EPR boxes show how long they decohere, which grows under a deadlock.

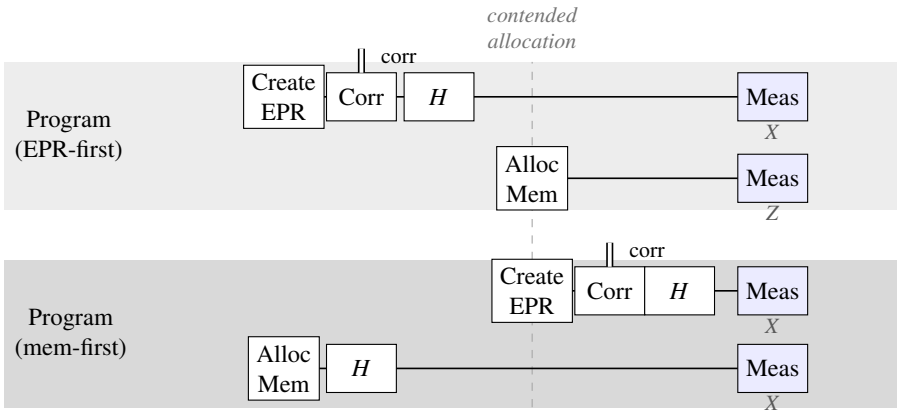


Figure A.10: Mem-Comm Allocation, shown for a node with a single communication qubit. One program acquires the communication qubit (Create EPR) before requesting a memory qubit, while the other requests them in the reverse order. Each instance then holds the resource the other needs at the contended allocation, producing a circular wait. The communication qubit is brought into $|+\rangle$ by the partner's correction (*Corr*) and an H, and the mem-first program also prepares its memory qubit in $|+\rangle$ with an H. Each program ends in two shaded success measurements (expect 0): the communication qubit in the X basis, and the memory qubit in the X basis for the mem-first program or the Z basis for the EPR-first program. The held qubit's wire spans the wait, so its length shows how long it decohered while blocked.

Appendix B

Declaration of AI usage

The recent years have seen a surge in use of Large Language Models (LLMs), here referred to as Artificial Intelligence (AI), to generate parts of or complete works of assessments. These models are non-deterministic statistical models doing auto-complete on a large scale and using them as part of scientific research has to be done under careful consideration. I am therefore inclined to be transparent about the extent to which AI was used during this project.

At the beginning of the project, I avoided using AI tools. Research and initial implementation of algorithms were done without the help of AI. This meant relying on existing tools such as the TU Delft Repository and Google Scholar to find relevant literature. I found this to be my preferred solution to be able to validate and improve on my ability to research. Especially as these tools might not be as ubiquitous if the current subsidies for AI inference are not continued. This decision were also taken at a point where I was continuously seeing hallucinations at a scale I was uncomfortable with.

However, as the thesis progressed and these tools became better and better, AI was used for four main use cases where I saw it fit. First, AI was used to generate diagrams in LaTeX. Spending time to deeply understand the makings of complex diagrams with tools like TiKz was not something I saw as relevant for my thesis work or my future career. Second, AI was used to assist with developing code for running evaluations. An evaluation suite was created to run evaluations with the Qoala simulator. Because of the amount of work required to extensively tune and configure each evaluation to my needs, I found it best to be able to use AI to increase my productivity. Third, AI was used to assist with writing the code to load in simulation results and plot them. Lastly, AI was in the end used for spell-checking and the flow between paragraphs.

The arguments I present, the text I have written, the algorithms I have created and implemented are all my own work and have been carefully assessed.