A System for the Development and Execution of Hybrid Classical/Quantum Programs

Ayush Tambde

5th Year

Stratford College

BTYSTE 2022 Stand No. 2302

December 2021

1 Abstract

Quantum computers exploit the effects occurring on a microscopic level to perform certain types of computation more efficiently than their classical counterparts. Programs that consist of both classical and quantum instructions are known as hybrid programs. They are prepared on a client device, and the quantum portion is sent for execution on a quantum computer. The device that receives these instructions is classical, and behaves as a controller for the quantum system. Oftentimes, classical code/data is sent to the controller, and it is preferable to execute those instructions closer to the quantum hardware, due to the currently low coherence times for qubits. One of the most efficient ways to accomplish this is to have a shared memory buffer between the client and controller that can be used to send code and data.

However, there are no known tools which facilitate the use of such shared memory for classical/quantum communication. To solve this problem the author has designed and built an operating system (OS) kernel, a programming language named Q++, and a quantum computing simulator. The OS enables the efficient sharing of information using Direct Media Access (DMA) buffers, provides system routines for user-space applications to do so, and coordinates processes wishing to read/write to shared memory. The Q++ programming language provides an intuitive interface with system calls, makes the compiled output adhere to certain custom conventions, and allows for the development of hybrid programs to run on the OS. The quantum simulator itself is a simulated Peripheral Component Interface (PCI) device running on the open-source classical computing emulator QEMU.

As of December 2021, the OS kernel supports process multi-tasking, memory paging, a simple UNIX-style file-system, shared memory management and quantum process management. The author has tested multiple programs on this system, and in all cases the output is as expected. Some of these tests are described in detail later on in the paper. However, there are still certain bugs in the kernel and language, and there is ample scope for optimization in the simulator. It is a goal in the future to make this system open-source.

Contents

1 Abstract **1**

2 Introduction **5**

3 The Operating System **7**
- 3.1 Bootloader . . . 8
- 3.2 Interrupts . . . 8
  - 3.2.1 System Calls . . . 9
- 3.3 Memory Management . . . 9
  - 3.3.1 Paging Implementation . . . 10
  - 3.3.2 Page Frame Allocation . . . 10
  - 3.3.3 Kernel Heap Allocation . . . 11
- 3.4 Processes . . . 11
  - 3.4.1 Loading . . . 12
  - 3.4.2 Scheduling . . . 12
  - 3.4.3 Multi-tasking . . . 13
  - 3.4.4 Quantum Processes . . . 13
- 3.5 Drivers . . . 14
  - 3.5.1 Screen . . . 14
  - 3.5.2 Keyboard . . . 15
  - 3.5.3 Hard Drive . . . 15
  - 3.5.4 Quantum Simulator . . . 15
- 3.6 File System . . . 17
- 3.7 Quantum Programs . . . 17

4 Programming Language Design **18**
- 4.1 Variables . . . 19
- 4.2 Operations . . . 20
  - 4.2.1 Arithmetic . . . 20
  - 4.2.2 Relational . . . 20
  - 4.2.3 Boolean . . . 20
- 4.3 Functions . . . 20
- 4.4 Conditionals . . . 21
- 4.5 Loops . . . 22
  - 4.5.1 For Loop . . . 22
  - 4.5.2 While Loop . . . 22
- 4.6 Quantum Instructions . . . 23
- 4.7 Keywords . . . 24

5 Programming Language Implementation **25**
- 5.1 Lexical Analyser . . . 25
- 5.2 Parser . . . 26
- 5.3 Reference Tracker . . . 26
- 5.4 Compiler . . . 26

5.4.1 Quantum Instructions 27

6 Quantum Simulator **28**
6.1 Classical Controller Simulator 30

7 Testing **32**
7.1 Test 1 32
7.2 Test 2 33
7.3 Test 3 34
7.4 Test 4 35

8 Results **38**
8.1 Test 1 Results 38
8.2 Test 2 Results 38
8.3 Test 3 Results 38
8.4 Test 4 Results 38

9 Summary **38**

10 Acknowledgements **39**

11 References **39**

12 Appendices **40**
12.1 Q++ Lexical Specification in Regex 40
12.2 Q++ EBNF Specification of Syntax 40

2 Introduction

Quantum Computers were first hypothesized in 1982 by Richard Feynman in a landmark paper[1], in which he stated that it is inefficient to simulate quantum mechanics using classical computers, since the memory required to store quantum states increases exponentially with the size of the system. The solution to this, he proposed, was the quantum computer.

Whereas classical computers compute using bits, a discrete, two-state unit of information, the fundamental building-block of quantum computers is the quantum-bit, or qubit. In addition to the two binary basis states, qubits can also be in superposition, in which case their state is well-defined yet their behaviour can be random. Qubits can also be entangled, in which the measurement results of two qubits are correlated.

The field has advanced much since Feynman's time, and is no longer limited to quantum simulations − algorithms have been devised which yield significant improvements over their classical counterparts. In 1994, Peter Shor discovered an algorithm that gives an exponential speedup for prime factorization[2], a "hard" problem for classical computers. In 1999, Lov K. Grover published his work on a quantum search algorithm that yields a quadratic speedup over the best possible classical unordered search[3].

As the language of classical computing is derived from Boolean algebra, the language of quantum computing derives from quantum mechanics − Linear algebra, the mathematics that describe vector spaces are surprisingly good at describing atoms. All quantum phenomena are said to occur in **Hilbert Space**[4], an infinite-dimensional complex vector space equipped with an inner product, a variant of the dot product.

The two basis states of computation are described mathematically in terms of vectors:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

These states are by convention said to be in the Z-basis, as the eigenvectors of the Pauli-Z (σ_z) operator:

$$\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

$$\sigma_z|0\rangle = 1|0\rangle$$
$$\sigma_z|1\rangle = -1|1\rangle$$

Larger systems are defined as tensor products of the two basis states:

$$|x_0x_1x_2...x_i\rangle = |x_0\rangle \otimes |x_1\rangle \otimes |x_2\rangle \otimes ...|x_i\rangle, x_n \in \{0, 1\}$$

The state of quantum systems can be manipulated through operators, which assume a matrix representation with respect to a basis.

Common quantum gates include:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

$$X = \sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$Y = \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

$$Z = \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

A purely quantum program consists of quantum gates and measurement instructions. The standard method to prepare a program for quantum computers is for a client to write instructions to a file and send this to a quantum computer. The quantum device itself is generally controlled by specialized, classical hardware. This controller receives the file and executes the instructions on a quantum device, processes the measurement results and returns them to the client.

In many cases it may be more efficient to execute code on the classical controller itself, rather than having to go through the inconvenience of an interaction with the client, which can be costly in terms of qubit coherence times, ie the time it takes for a qubit to lose its information. In this case, a method to send classical code or data to the controller is required. An efficient solution is to make use of a shared memory buffer between the client and controller, where the classical code and data can be shared freely.

However, there are no known tools that manage such shared memory for classical/quantum communication. For this reason, the author has designed and built an operating system kernel and a programming language compiler, tested on a quantum simulator the author wrote. The OS provides abstractions to read-/write to shared memory via Direct Media Access (DMA), allows sharing of classical code by introducing Quantum-and-Adjacent-Classical-Instruction (QACI) files, manages access through concurrency mechanisms such as the mutex, and provides the standard functions of an OS kernel, such as multi-tasking, process scheduling, memory management and paging, devices drivers, and a file system.

The programming language Q++ presents an abstraction of shared-memory system functions by incorporating them into its syntax, tracks references for sharing, and allows quantum code to be embedded within classical code, thus providing an intuitive development environment for writing hybrid programs. It is the result of rewriting the compiler for the author's Higher-Level-Oracle-Description-Language (HODL), which was originally developed for writing oracles for quantum circuits[5].

The simulator is written as an emulated Peripheral Component Interface (PCI) device, by extending the open-source classical computing emulator QEMU[6]. It is implemented as a standard statevector simulator, whereby the quantum state is represented as a vector, and all operations correspond to matrix multiplication.

As of December 2021, the author has finished writing the OS kernel from scratch in C and x86 assembly, although there are still features such as Translation-Lookaside-Buffers (TLB) which need to be implemented, and there is still scope for more advanced exception handling. The programming language compiler has also been written from in C/C++ and is functional. There are however, certain bugs, and ample scope for optimization.

While the author has tested this system locally on a simulator, without any changes it will not in its current form work on real devices. The first change that would need to be made is the controller software to adhere to the conventions introduced by the system. The second change would require a kernel driver for a network PCI device that supports DMA. The third change would require a dis-assembler within the kernel when sending machine code to the controller. Currently there is no need for such a tool since all everything runs locally, but if the controller is of a different CPU architecture then a dis-assembler and is essential.

3 The Operating System

The Operating System is the main component in this project. In this paper, we use the terms kernel and OS interchangeably, although to be more pedantic the kernel is defined as the executable which is in complete control of the computer hardware and software, whereas the OS encompasses the kernel and also includes tools and features such as system libraries, shell, etc. The defining features of the kernel are its native support for quantum operations and processes, and abstraction of communication with the quantum device via system calls. The kernel provides most functions that are expected, supporting software multi-tasking, memory management and protection through paging, interaction with hardware peripherals, and abstractions such as file-systems, while also providing functions to send classical code and data to the shared memory buffer.

The kernel is a 32-bit executable designed for the Intel x86 architecture. It is monolithic, meaning all its functionality is in one large executable, in stark contrast to a microkernel[8], where the kernel is defined as the minimum amount of software that can run in kernel-mode, and the rest, such as the file-system and device drivers, run on top as applications. The advantage of this is modularity is that certain components can be swapped in and out as if they are applications, rather than having to update the kernel itself. The author's decision for a monolithic kernel comes from reasoning that the development time of a microkernel is much longer, and the particular style has no benefit to the goal of the project.

3.1 Bootloader

Traditionally, the role of the bootloader is to load the kernel into memory[8], and conventionally takes up a sector of space (512 bytes), ending with a particular boot signature (0xAA55) to indicate a bootable disk to the BIOS, which loads the loader itself into memory at 0x7C00.

The structure of the bootloader is as follows:

1. Set stack pointer to just before where bootloader is loaded
2. Read from disk and load 64 sectors into the memory region starting at 0x15000
3. Disable interrupts
4. Load Global Descriptor Table. This is a structure that describes how memory is mapped. Prior to setting up paging, a "flat" memory model is implemented, where there are two overlapping memory segments of code and data, ie there are no restrictions when accessing memory
5. Enable paging by setting the least significant bit of the **CR0** register and mapping the first 4MB to itself.
6. Enter protected-mode (32-bit mode)
7. Create paging structures
8. Re-enable interrupts
9. Jump to 0x15000 (kernel start address)

3.2 Interrupts

During boot time the BIOS provides a series of useful interrupts. However, these are in 16-bit mode and to make use of them after entering protected mode would require one to return to a limited environment. One of the first things the kernel does after being loaded is to set up an Interrupt Descriptor Table (IDT), a table which the CPU uses to find interrupt handlers[8]. While a large number

of interrupts are unused, all hardware interrupts and many interrupt requests (IRQ) have corresponding handlers. Hardware interrupts include division-by-zero, page faults, general protection faults, etc. IRQs include system calls, timer, keystroke and quantum interrupts. Quantum interrupts are fired when DMA transfers have finished and when the quantum device has finished computing. Most handlers consists of assembly code which calls a high-level C routine. After an IRQ it is required to send a command to the Programmable Interrupt Controller (PIC) to allow it to accept further IRQs[7].

3.2.1 System Calls

Since all applications run in user-mode, this places serious restrictions regarding what a program is capable of. For example, it is almost impossible to print to screen without any help from the OS. System calls (syscalls) exist for this very reason. Interrupt 0x80 (deriving from UNIX) is used for syscalls. Parameters are passed in registers and the interrupt is invoked through the "INT" instruction. The EAX register is generally used for the function index in a syscall table, the ECX register for an optional sub-function index, while the EBX/EDX registers are general-purpose. Common classical interrupts include print functions, keyboard input, reading/writing to storage, etc. Quantum system calls (q-syscalls) use the interrupt 0x40, for DMA read/write to shared memory, compiling functions into a QACI file, allocating quantum instruction buffers, and to send commands to the classical controller.

3.3 Memory Management

Early computers had no mechanism for memory management or protection. This meant that processes running on the computer could access and even overwrite each others' memory. This was hardly desirable, since any malicious actor could exploit this and take control of the machine by overwriting other programs or even the OS itself. Thus, operating systems eventually grew to facilitate simple forms of managing memory[7].

One of the earliest forms of memory management was segmentation. The basic principle was that each process had a memory region devoted to code, data, stack, heap, etc. and that a process could only access memory within its segments. Segments could increase in size as required. However, with segmentation arises a problem known as external fragmentation, whereby the segment cannot grow due to lack of free memory after it, despite ample free space which may unfortunately not be contiguous. Due to this flaw, segmentation was largely phased out in favour of more modern approaches. The OS kernel does not use segmentation for this reason, but rather a technique known as paging.

The idea of paging is to divide all of addressable memory into a series of chunks, known as pages, all of which are generally 4KB in size[6]. These pages are said to represent virtual memory, that is, the memory management unit

(MMU) of the processor translates virtual addresses in each page to a corresponding physical address. This largely bypasses the problem of external fragmentation, since now if we wish to grow our heap for instance, all that is required is a simple mapping to a memory region which is contiguous to the heap. However, paging can lead to internal fragmentation, whereby even to simply allocate a couple of bytes, the kernel wastes 4KB.

3.3.1 Paging Implementation

Basic paging is enabled at boot time, and is done so that the kernel start address can be moved to any part of memory in the future. The first 4MB are identity-mapped, meaning the virtual addresses equal the physical ones. Setting up paging requires two data-structures, the page directory and page table. The page directory consists of 1024 entries, each of which point to a page table. A page table also consists of 1024 entries, each of which map 4KB. The kernel identity-maps the first 12MB and moves the page structures to the 8MB mark, where 4MB are allocated simply to store page structures.

3.3.2 Page Frame Allocation

Paging requires an efficient page frame allocation method, since it is important to be able to allocate physical memory for each virtual page. The kernel uses buddy allocation[9], a method that allocates memory in powers of two, and is described as follows:

1. Create a doubly-linked list, l. Each element in the list, indexed by i, represents a list containing all memory blocks of size 2^{i+12} that are free. At initialization, there are 20 elements and thus 20 lists. The most memory that can be allocated is 4GB and the least is 4KB. All lists are empty except for the first one, which has a single element that represents 4GB
2. Let m represent $\log_2$(size of memory to be allocated)
3. Iterate over all elements, i, from index $m - 12$ to 20, of l
4. If $l[i]$ is not empty and $i + 12 = m$ return a memory block from $l[i]$ and terminate
5. If $l[i]$ is not empty but $i + 12 > m$ then split the block $l[i][0]$
6. To split, change the size of the block $l[i][0]$ to half the original, and create a new block corresponding to the second block. In this algorithm, no changes are made to the original memory block, but rather the corresponding list entry
7. Add the second block to $l[i - 1]$

8. Recursively split block $l[i][0]$ until it optimally approximates the memory required and return the block. If the allocator is being used for the first time to allocate a page, it will result in all lists being filled since each recursive split will result in a block added to the previous list

9. Remove what was once block $l[i][0]$ from l

3.3.3 Kernel Heap Allocation

Dynamic memory allocation is essential within the kernel for creating linked lists, etc. The kernel heap starts at the 4MB mark and spans 4MB. The allocation method is a doubly-linked list allocator, whereby each memory region available is preceded by a header that consists of a pointer to the next memory block, a pointer to the previous block, block size, and a "used" flag.

The allocator is written as follows:

1. Create a doubly-linked linked list with a single element corresponding to one block of total heap memory

2. Iterate through the list until we encounter a block that is free

3. Change size of block to bytes required

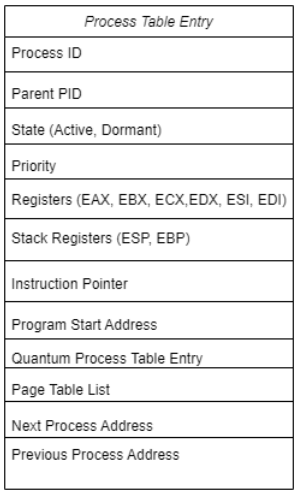
4. Mark the block as used

5. Insert another block which is of remaining size to the list

3.4 Processes

The kernel follows the standard process model common to UNIX-like systems[7]. The abstraction of all applications running (or asleep) into "processes" makes it easier to manage, schedule and serve them as required. Each process is described by an entry in the process table, the structure of which is seen in Figure 1. Each classical process also contains a pointer to a corresponding quantum process. This is because the kernel treats quantum processes independently. The advantage is that the kernel can schedule and manage quantum processes the same way it does classical processes.

Figure 1: Process Table Entry Structure



3.4.1 Loading

The kernel can load executable binaries in the Executable and Linking Format (ELF)[8]. Such a file consists of a series of headers which instruct the kernel how and where to load the program in memory, followed by the code and data segments. The kernel reads the header and identifies the file by checking the first byte (0x7F), followed by three bytes corresponding to "ELF" in ASCII. To load the program, the kernel iterates over each header, determines the page where the program is in, maps each page to a physical page frame, and copies the code and data segments from the file buffer to the page. A process table entry is created corresponding to the newly loaded process.

3.4.2 Scheduling

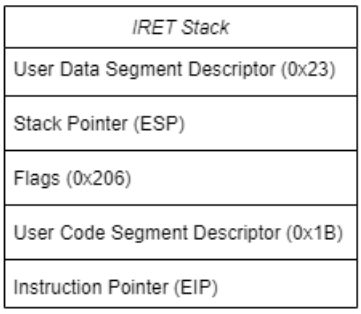
Scheduling is an important aspect of multi-tasking operating systems, as it is essential that every program gets a fair share of computing time[7]. Schedulers are generally grouped into *preemptive* and *non-preemptive*. The former switch processes after a fixed amount of time, whereas the latter only switch if the process voluntarily yields or blocks. Within the kernel, classical processes are governed by a preemptive round-robin scheduler, meaning after a fixed amount of time the kernel checks the process table to find the next process to run. Since the process table is implemented as a doubly-linked list, the inherent cyclic nature ensures that each halted process will eventually resume. The quantum processes on the other hand, use non-preemptive scheduling since it is not possible to halt a quantum program and resume it later − the much coveted

quantum RAM is but a dream at this moment in time. Each quantum process is scheduled on a first-come, first-served basis, unless another process has a higher priority.

3.4.3 Multi-tasking

The kernel is capable of giving the illusion of parallelism by switching between active processes when the timer interrupt (IRQ 0) fires. By default, the frequency of the timer is 16-18 Hz. The hardware uses the IDT to locate the interrupt handler and calls it. The handler, which is written purely in assembly for efficiency, saves the IRET stack frame − a particular structure that is pushed onto the stack by the hardware during each interrupt − to the user stack, as well as information such as registers. It proceeds to call the scheduler, which returns the information for the next process, which includes the stack. Since the next process has its IRET stack frame[8] saved, all that is left is to set this stack as the current stack, restore registers, and perform an IRET instruction to jump straight to the next process. Each process is initialized with a default IRET stack.

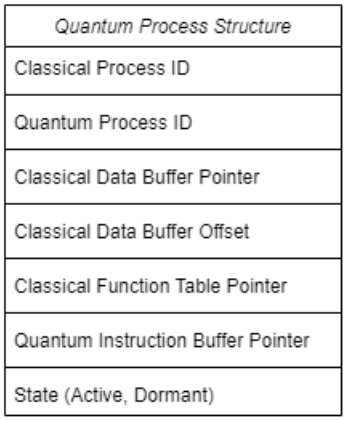
Figure 2: IRET Stack Structure



3.4.4 Quantum Processes

Each process that uses quantum operations has in its table entry a pointer to a corresponding quantum process structure.

Figure 3: Quantum Process Structure



The quantum process table is used for scheduling quantum tasks locally, ie for processes vying to enter the critical region in which they interact with the shared buffer. For example, if there are several processes that wish to run quantum programs, then it is more optimal to schedule the shortest program first. The existence of a table for quantum processes enables the OS to do so.

Since the shared buffer is global to all processes, it is important that access to it is tightly controlled. To do this the kernel maintains a mutex[7], a sort of lock on the device when a process enters the critical region. The shared buffer is then cleared when the process exits and the mutex is unlocked.

3.5 Drivers

The kernel is equipped with device drivers for the screen, keyboard, hard-drive, and quantum simulator.

3.5.1 Screen

The kernel prints everything through VGA text-mode, a somewhat antiquated display mode that resembles the style of monitors in the '70s and '80s. However, for testing purposes, this is sufficient and any attempt to implement a windowing system and GUI would not only take a tremendous amount of time but would be completely unnecessary to the overall aim of the project.

The video buffer begins at address 0xB8000, and is designed such that anything placed beyond this buffer is printed to screen. Printing a character requires two bytes of information: the ASCII code, and the color. The color byte is by

default green on black (0x02), Thus the kernel print function can print strings by iterating over each character until the zero byte is encountered, and copying them alongside a color byte, starting at the current offset in the video buffer. To print 32-bit hexadecimal values, however, the kernel uses a separate function. In this case, each hexadecimal digit is extracted and a corresponding character is fetched and printed to screen. Printing bitstrings is similar, although one has to convert the nybble to binary instead.

3.5.2 Keyboard

Contrary to what one might think, keyboards do not in actuality send ASCII or UTF characters upon each keystroke. Rather, once IRQ 1 fires, the kernel reads the 0x60 port to find it has received a code which must be mapped to its corresponding character[8]. This mapping is done by implementing a simple modulo-based hash table. During initialization, the kernel fills up this hash table with the appropriate mappings. In fact, there are two tables, the second one being used if the shift key is held.

3.5.3 Hard Drive

The acronym ATA stands for Advanced-Technology-Attachment, a hard-drive standard introduced in 1994 that still persists in various forms to this day[8]. It was notable for removing disk controller chips, which were conventionally on the motherboard, and attaching them directly to the drives themselves. Today, ATA has been improved and extended, and now known as Serial ATA (SATA). However, the kernel uses the old ATA format, since at this moment, faster disk-reads yield no benefit to the overarching goal of the kernel.

Prior to any form of reading or writing to disk, it is necessary to get the drive to identify itself. This involves writing the byte 0x0 to each port from 0x1F2-0x1F5, and the ATA_IDENTIFY (0xEC) command to port 0x1F7, and polling the drive to see if it is ready.

ATA drives generally use Logical Block Addressing (LBA) to address specific locations within the disk. LBA enumerates each block logically, ie the first block is 0, the second is 1, etc. as opposed to Cylinder-head-sector (CHS) addressing. For ATA writes it becomes a matter of writing the LBA, sector count, and the ATA_WRITE (0x30) command to the appropriate ports and waiting. Following this, it is possible to write data to the data port (0x1F0). The method for ATA reads is similar, although but with the ATA_READ (0x20) command and reading from the data port instead of writing to it.T

3.5.4 Quantum Simulator

Since the quantum simulator behaves as a PCI device, controlling it involves making use of Memory-Mapped I/O (MMIO), a form of communication that

is non-reliant on ports but rather certain memory addresses[8]. These memory addresses are identified and mapped during kernel initialization through a process known as bus enumeration, and involves iterating over each PCI device on the bus and read their information. At this stage the kernel also needs to enable PCI-busmastering for Direct Media Access (DMA) to work properly, which allows the quantum device to essentially become the "master" of the bus and perform read/write operations without explicit directions from the CPU.

Figure 4: PCI MMIO Read/Write

```
uint32_t mmio_read32(uint32_t base, uint32_t offset) {
       uint32_t* addr = (uint32_t*)(base+offset);
       return *addr;


}

void mmio_write32(uint32_t data, uint32_t base, uint32_t offset) {
       uint32_t* addr = (uint32_t*)(base+offset);
       *addr = data;

}
```

DMA is performed by writing the appropriate commands to the MMIO address space, alongside the address and size of the buffer we wish to transfer.

Figure 5: QC Simulator DMA Code

```
void qc_dma_read(uint8_t* buffer, size_t len) {
       mmio_write32(QC_DMA, QC_BASE, QC_DMA_SRC);
       mmio_write32((uint32_t)buffer, QC_BASE, QC_DMA_DST);
       mmio_write32(len, QC_BASE, QC_DMA_CNT);
       mmio_write32(QC_CMD_READ, QC_BASE, QC_DMA_CMD);


}

void qc_dma_write(uint8_t* buffer, size_t len) {
       mmio_write32((uint32_t)buffer, QC_BASE, QC_DMA_SRC);
       mmio_write32(QC_DMA, QC_BASE, QC_DMA_DST);
       mmio_write32(len, QC_BASE, QC_DMA_CNT);
       mmio_write32(QC_CMD_WRITE, QC_BASE, QC_DMA_CMD);

}
```

3.6 File System

The kernel's file-system also derives from UNIX[10]. Each file is described by an abstract structure known as an inode. The first block of the hard-drive (4KB) consists of a bitmap to indicate which inodes are free. The second block consists of a bitmap to indicate which blocks are free. The third contains all inodes. This means that the limitation in terms of number of files is $\approx \frac{4096}{sizeof(inode)}$. To create a file, we simply check which inodes and blocks are free using their respective bitmaps, allocate them, and write file data to those particular blocks.

Directories are different, however, although they are also described using inodes. We simply set the directory flag in the inode and allocate blocks for initial directory entries. We then fill those blocks with structures corresponding to directory entries. These structures consist of a filename and an inode number.

Reading a file consists of recursively loading the files in each directory in the file path − which are separated by a forward slash − into memory, freeing memory occupied by previous directory entries, and halting when the file desired is loaded.

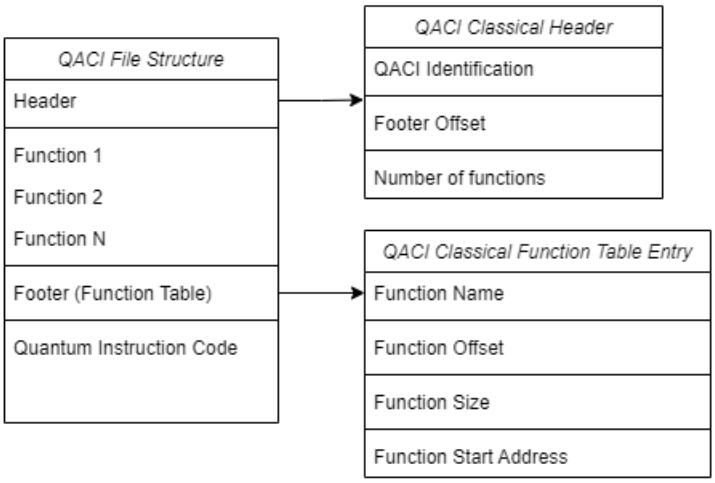
3.7 Quantum Programs

The defining feature of this kernel is its ability to work with quantum programs and facilitate classical/quantum communication. It does this through the driver for interacting with the PCI device simulator and DMA buffer and by providing system calls for writing such data to buffer.

The most fundamental system call is the QUANT function, which creates a quantum process and allocates a page worth of quantum instruction memory for the user program, which can then write quantum instructions to this buffer and invoke the QRUN syscall, which performs a DMA transfer of data from this buffer region to the shared memory, starting at the 1KB mark. The reason for this is that the kernel adheres to a format created by the author for sending classical and quantum code together to the controller, known as Quantum-and-Adjacent-Classical-Instructions (QACI).

The general structure of this file format is that the first quarter consists of classical code, and the remainder consists of quantum instruction code. The classical code is divided up into a header, followed by each function in memory, and ends in a footer. The header consists of five identification bytes, corresponding to "QACI" in ASCII, followed by the offset into the file at which the footer starts, followed by the number of classical functions present in the file. The footer consists of a function table. Each entry in the function table consists of the function name, the offset into the file they can be found at, their size, terminating with their load address.

Figure 6: Quantum Process Structure



The SENDQ syscall is responsible for writing to the classical region of the QACI file. It accepts as parameters the name of the function in ASCII, followed by the function start address. It copies the function into memory, halting when it encounters the byte sequence 0xC0DE, inserted by the compiler at the end of each function. It also performs a replacement while copying the machine code: the kernel's standard calling convention for a quantum gate is to call the value stored in the ESI register. However, for quantum functions, the controller requires a call to be made to the value stored at 0xA0000000. This convention is addressed by the compiler by moving 0xA0000000 into the EDI register. This means when sending over code, the byte sequence 0xFFD6 ("call esi") must be replaced by 0xFF17 ("call [edi]"). The SENDQ function also updates the header, and creates a function table entry in the footer.

The ASYNCQ syscall takes as parameters a function address, followed by the parameters on the stack. When the command is sent for quantum execution, the IRET stack is updated to jump straight to the function. The function exits when the completion IRQ is raised by the controller and the stack is fixed.

The QRUN syscall initiates the DMA transfer, sends the execution command to the controller, and for testing purposes, prints the value returned by the controller to screen.

4 Programming Language Design

The Q++ programming language serves two goals:

- Encapsulate the OS functions dealing with the quantum device and shared memory by providing abstracted interfaces with them
- To ensure that this way of writing hybrid programs is natural and intuitive

The language implementation is different to the author's previous Higher-Level-Oracle-Description-Language (HODL), the compiler retaining only a handful of code files from it. However, whilst differing in implementation, Q++ is heavily influenced by the design of HODL[5].

Q++ is a compiled, statically typed, imperative, and hybrid classical/quantum programming language. A program is divided up into functions. Expressions in Q++ consist of variable declarations and statements to manipulate that data. All expressions must be within functions. All expressions must terminate in a semicolon. In this regard, Q++ is heavily influenced by the classical programming language C alongside the author's quantum language HODL.

4.1 Variables

Data in Q++ is declared through an assignment operation. The structure of a variable declaration consists of a data-type, followed by the variable identifier as the left operand of the assignment operator (=). The right operand consists of a value given to the variable. The type system takes after C++, and thus the language supports four basic variable types:

- Integers (int)
- Floating-point numbers (float)
- Byte buffers (char)
- Strings (string)

Figure 7: Q++ Variables

```
function main() {
   int a = 2;
   float b = 0.5;
   char buff[256];
   string msg = "hello world";

   return;

}
```

4.2 Operations

Operations in Q++ can be either arithmetic, relational, or Boolean, and all operands must be either floats or integers. The following are supported:

4.2.1 Arithmetic

- Addition (+), Addition Assignment (+=)
- Subtraction (-), Subtraction Assignment (-=)
- Multiplication (*), Multiplication Assignment (*=)
- Division (/), Division Assignment (/=)

4.2.2 Relational

- Greater than (>)
- Less than (<)
- Greater than or equal to (>=)
- Less than or equal to (<=)
- Equal to (==)
- Not equal to (!=)

4.2.3 Boolean

- Logical AND (&)
- Logical OR (|)
- Logical NOT (!)

4.3 Functions

Functions are defined using the keyword "function", optionally preceded by the function return type, and followed by the function name. Parameters are declared in parentheses, separated by commas. The function body then lies within curly braces. During a function call, the function name is used, followed by the parameters in correct order, within parentheses and separated by commas. To execute the function non-locally, ie on the classical controller, the keyword "nloc" is used preceding the function call.

Figure 8: Q++ Functions

```
link printh;
# function returns an integer
int function foo(int var) {
   int new_var = var + 4;
    return new_var;

}

# main does function does not return anything
function main() {
   printh(foo(2));
   return;
}
```

4.4 Conditionals

Q++ supports two forms of classical conditionals:

- If-statements
- Else-statements

To define an if-statement, use the keyword "if", followed by an expression in parentheses. This is the conditional, ie the condition under which the following code will execute. The body of the conditional follows, and is enclosed within curly braces. Else-statements must succeed an if-statement, and are declared using the "else" keyword, followed by code enclosed within curly braces.

Figure 9: Q++ Conditionals

```
link print;

function main() {
   # check if var is less than 4 and if so
   # print "Yes" else print "No"

   int var = 2;
   if(var < 4) {
      print("Yes");
    }
   else {
      print("No");
    }
    return;
}
```

4.5 Loops

Q++ supports classical "for" and "while" loops.

4.5.1 For Loop

For loops are declared using the keyword "for", followed by three expressions in parentheses and separated by commas. The first expression is to be executed before the loop starts. The second expression is the condition under which the loop runs. The third expression is to be executed on each iteration of the loop. The loop body is then specified in curly braces.

4.5.2 While Loop

While loops are declared using the keyword "while", followed by an expression in parentheses. This is the condition under which the loop runs, and is followed by the loop body in curly braces.

Figure 10: Q++ Loops

```
function main() {
   for(int i = 0, i < 100, i+=1) {
     # loop body
    }

    int i = 0;
    while(i < 10) {
      # loop body
    }

   return;

}
```

4.6 Quantum Instructions

Q++ supports the following assembly instructions as intrinsic functions:

- Hadamard Gate – *H(qubit index)*
- X Gate – *X(qubit index)*
- Y Gate – *Y(qubit index)*
- Z Gate – *Z(qubit index)*
- Phase of $\frac{\pi}{2}$ Gate – *S(qubit index)*
- Phase of $\frac{\pi}{4}$ Gate – *T(qubit index)*
- Arbitrary Phase Gate – *P(qubit index, phase angle)*
- Controller-X Gate – *CX(control qubit index, target qubit index)*
- Arbitrary X-Rotation – *RX(qubit index, rotation angle)*
- Arbitrary Y-Rotation – *RY(qubit index, rotation angle)*
- Arbitrary Z-Rotation – *RZ(qubit index, rotation angle)*
- Measurement – *MEASURE(qubit index, optional result pointer)*
- Measure All – *MEASURE_ALL(optional result pointer)*

Figure 11: Q++ Quantum Instructions

```
function main() {
    H(0);
    P(0, pi);
    CX(0,1);
    return;
}
```

4.7 Keywords

Q++ recognizes the following keywords and macros:

- function
- int
- float
- char
- string
- return
- link
- nloc
- qrun
- async
- pi (π)
- addr(var)
- copy(var1, var2)

The keyword "return" is used to terminate a function and can optionally be followed by a value to be returned. This value is moved into the EAX register as is convention, local variables and parameters are popped off the stack, and the function returns.

The keyword "link" is used to specify external functions not defined in the current source file. The syntax is to use the keyword, followed by the function name.

The keyword "nloc" stands for non-local, and is used to execute functions on the controller. The syntax is to use the keyword followed by a normal function

call. The call does not happen locally, but rather on the classical controller for the QC.

The keyword “qrun” stands for “run quantum”, and is used to tell the controller to execute the quantum instructions placed in the shared buffer.

The keyword “async” stands for asynchronous, and is used to tell the kernel to execute another function while the process is waiting on the quantum device. The syntax is to use the keyword followed by a function call.

The macro “addr” takes as parameter a variable name and replaces itself with the address of the variable on the stack. The macro “copy” is essentially a MOVE instruction which transfers the contents of the second parameter into the first.

Figure 12: Q++ Keywords

```
int function foo() {
   return 5;
}
int function bar() {
   return 4;
}
function main() {
   RX(0, pi);
    # execute foo on controller
    async bar();
    nloc foo();
    # tell controller to execute RX and foo
    qrun;
    return;
}
```

5 Programming Language Implementation

The compiler for Q++ is written in the C++ programming language. It translates Q++ source code to x86 32-bit Intel syntax assembly code.

5.1 Lexical Analyser

The first stage in the compiler is the lexical analysis stage. It is here where comments are removed, and characters read in through the source code file are grouped together and assigned tokens. This is done through iterating over each character, storing them in a proxy, and flush the proxy once a comma, semicolon

or whitespace is encountered. The tokens are then looked up from a hash-table. An ordered map consisting of token-value pairs is returned.

5.2 Parser

The parser accepts as input the ordered map produced by the lexer, and proceeds to create an Abstract Syntax Tree (AST) representation of the program, a structure consisting of binary trees alongside useful metadata. Operations are parsed by creating a node corresponding to the operator, with pointers to two child nodes corresponding to the left and right operands respectively. The parser produces an AST list.

5.3 Reference Tracker

In order to send functions to shared memory, it is also becomes necessary to send all functions or data the function depends on to ensure correct execution. Therefore, after parsing, the reference tracker is called and is responsible for creating a hash table with the function name as key and another hash table, consisting of dependencies, as the value. A hash table is used since search is performed efficiently in constant, O(1) time. The reference tracker recursively iterates through the ASTs and calculates all dependencies.

5.4 Compiler

The final stage is the actual compilation, where the program in AST representation is translated to a low-level assembly language file.

During compilation, it is necessary to keep track of CPU resources such as the registers. Note that this does not include run-time resources. The compiler maintains a structure which keeps track of what registers are free during the course of compilation. Allocating registers while writing the compiler is analogous to dynamic memory allocation, in the sense that every register must be deallocated.

Local variables are kept track of through symbol tables. Each new scope (function bodies, if statements, loops) creates a new symbol table, which is represented as a hash table with the variable name as key and stack position as value. Each symbol table also stores a pointer to its parent table. If a symbol is not resolved using a current symbol table, it is recursively searched for through its parent.

The compiler recursively iterates over all nodes in the AST, halting only when an identifier or numeric constant is encountered. When this happens, if it happens to be an *lvalue*, as in, on the left-hand side of the expression, then it is copied into a free register, the name of which is returned. If it isn't, then

either the value (if it is known) or the stack offset is returned.

Function calls are compiled by pushing all registers onto the stack, followed by the function parameters in reverse order. The function name is then called using the instruction "call", followed by fixing up the stack and popping the values pushed onto it.

The process through which conditionals are compiled to assembly language relies on labels. For a standard if-statement, the conditional expression is compiled first, corresponding to a "CMP" instruction, followed by a "JMP" instruction to a label if the condition is false. The body of the the conditional is then compiled to lie between "JMP" and the label.

The compilation of loops similarly relies on jumps and labels. For a for loop, the first expression supplied in parentheses is compiled first. Following that, a label is generated, and under that the second expression is compiled next. This results in some form of a "JMP" instruction to the break label, which marks where the loop ends. Between the "JMP" and break label, the body of the loop is compiled, alongside the third expression.

Compiling while loops involves creating a label first, followed by compiling the expression in parentheses, and further followed by a particular form of "JMP" instruction to the break label. The body of the loop lies sandwiched between the "JMP" and break label.

5.4.1 Quantum Instructions

The structure of quantum instructions is heavily abstracted out through quantum gate calls. Furthermore, the OS provides a library of user-space system functions to link against the compiled output. These functions are written in assembly and generally constitute of both normal system calls and q-syscalls. Quantum instructions are implemented as writing bytes to a special buffer allocated by the OS. The QUANT system function, invoked by the program load routine, returns the start address of this buffer, and moves it into the EBX register. The actual writing to the buffer is done by the QGATE system function which accepts three arguments, the first being the quantum opcode, the second the qubit index, and the third is optional to either specify an angle for parameterized gates or the target qubit for controlled-not. This data is written to memory and the current offset into the buffer is incremented. The compiler calling convention for quantum gates is to load the address of the QGATE function into the ESI register, push the parameters, call ESI, and restore the stack.

The compiler ensures that each function definition terminates with the byte sequence 0xC0DE. This is purposefully defined to be non-executable even if within an executable page and is why the "return" statement is necessary, oth-

erwise an "Invalid Opcode" exception is thrown if the processor attempts to execute it. This sequence instructs the OS when to stop copying the function over to the QACI buffer.

The "nloc" keyword is implemented by checking the function reference map and fetching all dependencies. The addresses of each dependency is pushed onto the stack, alongside the function name in ASCII, and the SENDQ system function is invoked for each.

The implementation of the "async" keyword involves pushing the function address onto the stack and calling the ASYNCQ system function.

To implement the "qrun" keyword, the start address of the quantum instruction buffer is pushed onto the stack and a call is made to the QRUN system function, which initiates the DMA transfer to shared memory.

6 Quantum Simulator

The quantum simulator is an emulated Peripheral Component Interface (PCI) device running on the classical computing emulator QEMU. It is a heavily modified version of the PCI device template which is provided within the QEMU source code. The reasons for it being a PCI device are relatively simple: it can be easily configured to enable DMA and thus implement shared memory, and the emulator creates a new thread to run the PCI device, thus providing more realistic device behaviour through parallelism.

The quantum simulator is a standard statevector simulator. Each quantum state is represented as an 2^n dimensional vector, where n is the number of qubits. The default quantum state is the $|0\rangle^{\otimes n}$ state, where $\otimes$ represents the tensor product. During initialization, the simulator creates the default state by fetching the $|0\rangle$ state and applying the tensor product with itself n times.

The algorithm for calculating the tensor product, c, between two vector, a and b, is as follows:

1. Initialize c to zero components
2. Iterate over each component, i, in a
3. Iterate over each component, j, in b
4. Perform complex multiplication with i and j as the operands and store the result in k
5. Add k as a component of c

Such a state can be manipulated through linear operators, commonly represented as matrices (gates) with respect to a basis. In this case, everything is in the Z-basis. Each matrix is implemented as a vector of pointers to row-vectors. Each operator is initialized to act on a single qubit, except for the controlled-not operator which is by default a two-qubit quantum gate.

To apply a quantum gate to a state greater than a single qubit, it is necessary to tensor the single-qubit version with identity matrices[4]. For example if we have a five-qubit device, and we wish to apply the X gate to the fourth qubit, then the resultant matrix X_4 will have the resulting decomposition:

$$X_4 = I \otimes I \otimes I \otimes X \otimes I$$

where I and X are single-qubit gates.

The algorithm for calculating the tensor product, c, between two matrices, a and b, is as follows:

1. Initialize c to hold zero components
2. Get the transpose, $b^\dagger$, of the matrix b. This involves inverting the rows and columns, and is for making computation easier since in this algorithm, operations between row-vectors and column vectors are desired, but matrices are represented as vectors of row-vectors.
3. Iterate over each row-vector, i, in a
4. Iterate over each row-vector, j, in $b^\dagger$
5. Calculate vector tensor product, k, between i and j
6. Add k as a row-vector to c

The algorithm for preparing a quantum gate applied to a single qubit in a multi-qubit system, is as follows:

1. Initialize a variable i to zero
2. Initialize a variable q to the index of the qubit in the state
3. Create an empty matrix, m
4. If $i = q$ and if $i = 0$, then initialize m to a single-qubit version of the gate
5. If $i = q$ and $i \neq 0$ then tensor m with I where I is the identity matrix
6. If $i = 0$ and $i \neq q$ then initialize m to a single-qubit version of I
7. If $i \neq 0$ and $i \neq q$ then tensor m with I

An obvious potential optimization here is to check for all operations on other qubits that would be able to occur in parallel. In this case, instead of performing a tensor product with I one can instead tensor together all non-sequential operations, thus producing a matrix which can apply those operations in one swift stroke. To illustrate this, let us assume the existence of a state, $|101\rangle$. It is desirable to apply X to the first qubit, Z to the second, and Y to the third. Instead of:

$$(X \otimes I \otimes I)|101\rangle$$
$$(I \otimes Z \otimes I)|101\rangle$$
$$(I \otimes I \otimes Y)|101\rangle$$

One can simply combine the aforementioned operations into:

$$(X \otimes Z \otimes Y)|101\rangle$$

However, this optimization has not yet been implemented but is a future development goal.

The simulator implements measurement by first fetching the probabilities associated with each basis state. This is done by simply getting the absolute square of the complex components. Then, a random floating-point number, r, is generated from zero to one. To test to check if the state $|x_n\rangle$ was measured is:

$$\sum_{i=0}^{n-1} P(|x_i\rangle) < r < \sum_{i=0}^{n} P(|x_i\rangle)$$

where $P(|x_i\rangle)$ is the probability of measuring $|x_i\rangle$. If the state $|x_n\rangle$ was measured, it is stored in decimal form in either the shared memory buffer, or in a valid location if an address is specified, and an IRQ is raised. The quantum state is reset after each measurement operation.

6.1 Classical Controller Simulator

The classical controller simulator is integrated with the quantum simulator. In fact, it is a single piece of software, since it is the controller that calls on the quantum gates to perform linear operations such as those described above. Therefore, we can visualize the entire simulator as a classical simulator that calls on a quantum simulator through an "apply_gate" function.

Once the classical controller is sent the execution command, it treats the first 1KB of the shared buffer as a QACI file, and the remaining 3KB as a standard quantum instruction file. Its first role is to check the QACI header and iterate over each function table entry, mapping the memory that each function takes up, and copying each function to its respective location.

The next phase consists of iterating over the quantum instruction file. The structure is such that the opcode is first, followed by its parameters, each of which are 32-bit in size. The appropriate gate from the quantum-gate table is fetched and called. However, the opcode 0xA corresponds to a function "call", followed by the function name in ASCII and parameters. The function start address is looked up in its table and is called.

7 Testing

The primary question the author grappled with was what variable to test. Execution time was suggested, but it was reasoned that it would create new variables that would have to be taken into account or would otherwise bias the data, since comparing the execution times between a local machine simulator and an IBM supercomputer controlling state-of-the art quantum hardware would have been analogous to comparing apples and oranges. Furthermore, since IBM does not provide functionality for partially running custom code on the controller without explicitly writing an Open Quantum Assembly File, the comparison between a high-level and low-level language would have been even stranger. The simpler and less prone-to-bias approach, the author reasoned, was to execute programs on the local device, and validate their results via mathematics or logic. The following are four tests which each measure some unique aspect of the system. The programs are written in Q++, compiled and assembled on a 32-bit Linux environment, and manually written to a virtual hard-drive. The emulator QEMU is then started and boots from the drive, reading the program from the file-system and loading the ELF-file and executable program.

7.1 Test 1

The first test is purely classical. It is to check whether or not Q++ can compile a classical program and whether or not the kernel can load a classical process and execute it correctly. The program tests conditionals, loops, and functions. The program accepts user input, puts it into a character buffer of size 256, checks if it is equal to the string "hello", and if it is, greets the user ten times. If it is not, the string "Bye!" is printed and the program returns.

Figure 13: Test 1 Q++ Program

```
link QGATE;
link print;
link printh;
link input;
link strcmp;

function say_hi_back() {
   print("hello to you to!\n");
   return;

}
function main() {
   # declare buffer
   char buff[256];
    # put user input into buffer
   input(buff,256);
    # check if user input is "hello"
   if(strcmp(buff, "hello") == 1) {
      # say hi back 10 times if so
      for(int i = 0, i < 10, i+= 1) {
         say_hi_back();
      }
   }
    # else say bye
   else {
      print("Bye!\n");
   }
   return;
}
```

7.2 Test 2

The second test runs the quantum search algorithm on two qubits. This algorithm offers a polynomial-time advantage over the best possible unordered classical search[3]. The first part of the algorithm involves generating a uniform superposition, $|\psi\rangle$, and "marking" the term to search for. In this case, the state to be marked is $|11\rangle$. The Grover Diffusion Operator $(2|\psi\rangle\langle\psi| - I)$ is applied once, followed by a measurement of both qubits.

Figure 14: Test 2 Q++ Program

```
link QGATE;
link qrun;
function apply_oracle() {
   # mark the |11> state
   H(1);
   CX(0,1);
   H(1);
   return;
}
function diffuse() {
   # Grover diffusion operator increases
   # probability of measuring |11> state
   H(0);
   H(1);
   X(0);
   X(1);
   H(1);
   CX(0,1);
   H(1);
   X(1);
   X(0);
   H(1);
   H(0);
   return;
}

function main() {
   # prepare state as even superposition of 2 qubits
   H(0);
   H(1);
   apply_oracle();
   diffuse();
    MEASURE_ALL();
    # execute and print result
   qrun;
   return;

}
```

7.3 Test 3

This third test is for testing interaction with the classical controller and runs a program on it. A function running on the classical controller generates a uniform superposition of four values, measures and stores the result in a variable a, and if $a \geq 3$ then the value 10 is returned, otherwise the value 20 is returned. It is

expected that since the superposition is of the form: $\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$, the value 10 will be returned $\approx 25\%$ of the time.

Figure 15: Test 3 Q++ Program

```
link QGATE;
link qrun;
link sendq;
link strcpy;
int function process() {
   int a = 0;
   # generate uniform superposition
   H(0);
   H(1);
   MEASURE_ALL(addr(a));
   # check if result is greater than or equal to 3 and return 10 if so
   if(a >= 3) {
      return 10;
   }
   # else return 20
   else {
      return 20;
   }
}
function main() {
   # set this function to run on the controller
   nloc process();
   # execute and print result
   qrun;
   return;

}
```

7.4 Test 4

This is designed to test a real-world quantum application known as the Variational Quantum Eigensolver (VQE), written in Q++ and executed on the OS and controller. VQE was proposed as a Noisy-Intermediate-Scale-Quantum (NISQ) algorithm − a program that is useful on current quantum computers − by Peruzzo et al in 2013 [11]. The algorithm finds the lowest eigenvalue of a given Hamiltonian operator (H), and is used in computational chemistry to find the lowest-energy state of molecules. The algorithm makes use of the Variational Principle[4] that states that the expectation value, $\langle H \rangle$, is bounded by the minimum eigenvalue E_0. In this program, the minimum eigenvalue of the ZZ operator is calculated, such that $H = ZZ$. The algorithm begins by initializing a two-qubit state $|\psi(\theta)\rangle$, known as an ansatz, based on some variable parameter (θ). This initialization includes applying a parameterized $RX(\theta)$

gate. The algorithm proceeds by measuring ZZ. If the measured value is 0, then the expectation value is incremented by 1, otherwise it is decremented by 1. However, this itself is not the expectation value − we should ideally divide it by the total number of times we measure ZZ, but it makes no difference during minimization, only during the final processing. The expectation value is then calculated a certain number of times, each time checking if $\langle\psi(\theta)|ZZ|\psi(\theta)\rangle$ is less than the previous value. If it is, θ is incremented by 1, otherwise it is decremented. Note that θ is scaled by a factor of ten in this example due to floating point errors that occur during compilation. This discrepancy is easily addressed by modifying the controller to account for this factor. Therefore, one is actually modifying θ by 0.1. The optimization method described above is a primitive one, since there are better optimizers such as COBYLA and SPSA, but for our purposes a simple scalar optimizer works just as well. Eventually, θ will reach a point where $\langle\psi(\theta)|ZZ|\psi(\theta)\rangle \approx E_0 =$ lowest energy state.

Reasoning from:

$$ZZ|00\rangle = 1|00\rangle$$
$$ZZ|01\rangle = -1|01\rangle$$
$$ZZ|10\rangle = -1|10\rangle$$
$$ZZ|11\rangle = 1|11\rangle$$

we can see that E_0 is -1.

The following program calculates the expectation value (scaled by 1024) by varying a state based on an arbitrary angle $\frac{\theta}{10}$. The result measured should be -1 after adjusting for the scaling factor. The key advantage of executing this VQE function on the controller is that communication with the client does not need to occur upon each iteration of the optimization process, and thus the main execution bottleneck is greatly reduced.

Figure 16: Test 4 Q++ Program

```
link QGATE;
link qrun;
link sendq;
link strcpy;
link strcmp;

int function vqe() {
   # calculate expectation value
    int theta = 2;
    int prev = 8192;
    for(int iter = 0, iter < 100, iter+=1) {
      # set expectation value to 0
      int exp = 0;
      for(int i = 0, i < 1024, i+=1) {
        int meas = 0;
        # prepare ansatz
        H(0);
        CX(0,1);
        RX(0, theta);
        CX(0,1);
        # store measured output in meas
        MEASURE_ALL(addr(meas));
        # check if last qubit is 0,
        # if so increment expectation value by 1
        if((meas % 2) == 0) {
         exp += 1;
        }
        # else decrement by 1
        else {
         exp -= 1;
        }
      }
      # compare expectation value to previous one
      if(exp < prev) {
         # if it is less then assign and increment by 1
        copy(prev, exp);
        theta += 1;
      }
      # else decrement by 1
      if(exp > prev) {
        theta -= 1;
      }

   }
   return prev;
}
function main() {
   char buff[256];
    input(buff,256);
    if(strcmp(buff,"VQE") == 1) {
      # set vqe function to execute non-locally
         nloc vqe();
        # execute and print result
         qrun;
   }
   return;
}
```

8 Results

8.1 Test 1 Results

Upon running the program, a blank black screen is displayed with a flashing green cursor. Upon keying in the characters corresponding to the string "hello", the text "hello to you too!" is displayed ten times, each on a new line. If any other string is keyed in as input, the text "Bye!" is printed on a new line.

8.2 Test 2 Results

The program returns the value $|11\rangle$ with 100% probability. This is not surprising since a two-qubit quantum search algorithm generally works perfectly when tested on a simulator. This shows that the quantum simulator can perform matrix-multiplication, measurement, and also return the result in a local stack variable.

8.3 Test 3 Results

After running this program 15 times, the value 10 was returned a third of the time, and the remaining values were all 20. This is not surprising, since the algorithm had a 25% chance of returning 10. This test shows how measurement results may be post-processed by the controller prior to returning them back to the client.

8.4 Test 4 Results

After executing the VQE algorithm a number of times, the expectation value always returns -1024. Recalling that we did not divide the value by number of iterations during optimization, doing so at the end yields $\frac{-1024}{1024} = -1 =$ minimum eigenvalue of the ZZ operator. This indicates that the quantum circuit has run properly and a real-world application has been developed using Q++, compiled correctly, and tested on the author's system.

9 Summary

The author designed and built a novel operating system kernel and programming language for facilitating communication between a client and a quantum computer through a shared buffer. The author has tested the system by extending the open-source classical computing emulator QEMU by adding a quantum simulator and running the kernel on it. While many programs have been tested and do seem to yield correct results, there is still scope for optimization and the system could be expanded to target different CPU architectures. It is also a further development goal of the author to merge this system with the Linux Kernel in order to support a multitude of applications and adhere to the POSIX

standard.

In May 2021, mid-way through the system's development, IBM announced the beta version of a service known as "Qiskit Runtime", where one can upload and run hybrid programs on IBM's devices on the cloud[12]. This largely reduces the need to share code/data, since all will run on the controller. Unfortunately, though, this means that no aspect of the program can run locally, and also results in lock-in to a specific quantum computing architecture (superconducting qubits). Furthermore, Qiskit run-time is currently offered in a limited way and can only run a predefined set of programs, whereas the author's system can run any program written using Q++. However, the fact that IBM, an industry leader, has started to explore something similar to what the author has already built, heavily indicates that there is a need for the author's system.

10 Acknowledgements

The author would like to thank Prof. David Abrahamson (TCD Compiler Design and Optimization) and Prof. Brendan Tangney (TCD Computer Science and Statistics) for their help in crafting the narrative of this paper and providing helpful insights. Thanks are also due to Dr. Keith Quille (Computer Science lecturer at TUDublin), Dr. Lee O'Riordan (ICHEC, Xanadu AI), Dr. Peter Rohde (Quantum Computing at UT Sydney) and Robert Smith (Chief of Quantum Software at HRL Labs) for discussing this project and offering valuable feedback.

11 References

[1] Richard P Feynman. Simulating physics with computers, 1981. International Journal of Theoretical Physics, 21(6/7)

[2] Peter Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer, 1994. arXiv:quant-ph/9508027

[3] Lov K. Grover. A fast quantum mechanical algorithm for database search, 1996. arXiv:quant-ph/9605043

[4] Nielsen and Chuang. Quantum Computation and Quantum Information, 2000.

[5] Ayush Tambde. A Programming Language For Quantum Oracle Construction, 2021. https://arxiv.org/abs/2110.12487

[6] https://www.qemu.org/

[7] Andrew S. Tanenbaum, Herbert Bos. Modern Operating Systems. Fourth Edition. Pearson

[8] https://wiki.osdev.org/

[9] Donald Knuth. The Art Of Computer Programming. Addison-Wesley

[10] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau (University of Wisconsin-Madison). https://pages.cs.wisc.edu/ remzi/OSTEP/

[11] Peruzzo et al. A variational eigenvalue solver on a quantum processor. https://arxiv.org/abs/1304.3061

[12] https://qiskit.org/documentation/partners/qiskit_runtime/

[13] https://www.intel.com/content/dam/develop/public/us/en/documents/325462-sdm-vol-1-2abcd-3abcd.pdf

12 Appendices

12.1 Q++ Lexical Specification in Regex

digit = [“0”..“9”]
number = digit+
letter = [“a”..“z”] | [“A”..“Z”]
identifier = letter (letter | digit | “_”)*
keyword = “else” | “for” | “function” | “if” | “int” |
“return” | “float” | “char” | “string” | “while” | “nloc” |
“async” | “link” | “qrun”
macro = “addr” | “copy”
quantum_instruction = “H” | “X” | “Y” | “Z” | “P” | “S” | “T” | “RX” | “RY”
“RZ” | “MEASURE” | “MEASURE_ALL”
operator = “+” |“-” | “*” | “+=” | “-=” | “*=” |
“<” | “>” | “<=” | “>=” | “==” | “!=”

Note: Keywords are reserved names and cannot be used as identifiers

12.2 Q++ EBNF Specification of Syntax

program = subroutine {[subroutine]}
subroutine = (function)
function = [type] **function** identifier parameters **{** body [**return** identifier] **}**
type = (**char** | **int** | **float** | **string**)
parameters = **(** [type identifier {**,** [type identifier]}] **)**

body = {assignment | fcall | operation | loop | cond}
assignment = type identifier **=** (integer | operation | fcall)
fcall = identifier **(** {[operation | identifier | fcall]} **) ;**
operation = (identifier | integer | fcall | operation) operator
(identifier | integer | fcall | operation)
loop = (for | while)
for = **for (** assignment semicolon operation semicolon operation **) {** body **}**
while = **while (** operation **) {** body **}**
cond = (if | elsif | else)
if = **if (** operation **) {** body **}**
elsif = **elsif (** operation **) {** body **}**
else = **else {** body **}**