Node OS

Lennart Bittel, QuTech

December 20, 2017

1 Overall outline

Node OS is separated into a classical and quantum part. The classical part can be any multi purpose operating system such as linux. A new scheduler to handle these tasks connected to the quantum part will be useful, but is not yet implemented.
A program will be used to generate instructions to the quantum CPU. OpenQL, which was developed for superconducting computers, could potentially be modified to work for NV- centers as well as include multi-device entanglement and the necessary classical communication in such protocols

2 Quantum CPU

Currently QCPU is implemented as a Linux Kernel model, but could also, if latency requirements are too strict also run on a micro processor directly. Currently it has the following properties:

- Perform arbitrary single device quantum computation.
- Keep track of all measurement outcomes of a certain protocol.
- Allow for classical communication between other QCPUs in a header based system. This is mandatory as even simple protocols like teleportation require fast exchange of measurement results. Additionally QCPU in the future should be aware of network headers which are relevant to its current state.

2.1 Programs

A quantum program (Progs) is a physical space in memory which contains the executable code as space for the code variables.

2.2 Scheduling

There are currently three queues:

- Not_started queue: I contains programs which are uploaded to the QCPU, but not yet running. This is useful, as the computational power is only limited and only one or maybe two programs will run on one machine in a given moment.
- Running: These contain all programs which want to be executed. The scheduler chooses which program to execute first. As the list of running programs on a quantum computer is short, deciding the best can still be computationally expensive without taking long amounts of time ($\propto 10^{-8}s$).
- Sleeping: Here items which require further input to proceed are stored here. This can mean a measurement outcome,classical communication or waiting for entanglement to be generated. Depending on the type of CQC, there might also be a response after every performed gate. In this case the program also sleeps until this message is received.
  Currently programs which are done with their task will also sleep.

2.3 QCPU communication with NV or SimulaQron

QCPU works with quantum devices through the CQC interface. All necessary functionality to run a protocol are implemented. Additional feedback, like precise timing or general information about the quantum computer are not yet implemented. This is a necessary next step, but needs further input from experimentalists and might therefore still drastically change.

2.4 Communication between two QCPUs

Currently this communication is performed through UDP and IP in a local network. This is slow and leads to long latencies on the raspberry pi, but in principle different hardware and modes of communications (raw ethernet,...) can make this arbitrarily small.

2.5 User $\leftrightarrow$ QCPU interface

Currently QCPU is built as a kernel module for Linux. The savest way for this communication is by making it a char device. A char device takes in a string that is written to it and returns a string if it is read. The current functionality of this interaction includes:

- Load a program into QCPU, this includes ID of the node it should run on. If they do not match, the job is transfered to the correct node. This allows to perform protocols from a single entry point. (can of cause also be turned off)

- Get an update about the process of a program (e.g. the current line of code which is waiting to be run).
- Receive the resulting variables at the end of the computation.

Additionally the interface should also allow further instructions like pausing or killing a job (killing also demands Node OS to perform this securely, meaning to unregister all used qubits and let nodes involved know that a kill command was triggered).
After receiving the results of computation, might decide to rerun a protocol or delete it. To have QCPU also handle these requests might be useful.
Additionally reseting the tasks priority queue could be beneficial.

3 User space environment

The user side needs to define the source code which QCPU can understand.

3.1 Quatum Key Distribution

An example code which works on the current implementation would be for QKD:

```
1  //macros to define on which machines the quantum protocol should be
       perfomed
2  #define alice 0
3  #define bob 1
4
5  unsigned int prot_id=100; //needs to be unique to properly function
       on both machines
6
7  qsys sysA;init_sys(&sysA,alice,prot_id); //define environment for
       alice
8  qsys sysB;init_sys(&sysB,bob,prot_id); //define environment for bob
9
10 int resa[10], resb[10],feedbackA,feedbackB;
11 for(int i=0; i<10;++i) // 10 cycles of QKD
12 {
13   //part for alice
14   int q1=create_q(&sysA);
15   int q2=create_q(&sysA);
16   apply_gate(&sysA,cmd_hgate,q1);
17   apply_2gate(&sysA,cmd_cnot,q2,q1); //An EPR pair is generated
18   send_q(&sysA,q1,bob); //send one half to bob
19   meas_q(&sysA,q2,&resa[i],remove); //measure the other half and
       store it in resa[i], also indicate that it is not needed
       anymore)
20
21   //part for bob
22   int qb=recv_q(&sysB,alice); //get the qubit from alice
23   meas_q(&sysB,qb,&resb[i],remove); //and measure it into resb[i]
24 }
25
26 //The protocol is now defined and can be executed
27 //This could also be in a different file for convenience
```

```
28
29 start_exec:
30 feedbackA=perform_sys(&sysA,10);//send alice's side to be executed
31 feedbackB=perform_sys(&sysB,10);//send bob's side to be executed
32
33 if(!feedbackA || !feedbackB) //statement to handle failure
34   goto start_exec;
35
36 //further classical post processing, eg. generate a key from the
       results
37
38 get_key(resa);
39
40 ...
```

- prot_id is needed here to allow QCPU to know that these two protocols work together. Coming from the same source this could be automatized, but they could also be committed from each node individually this quantity allows it.
- perform_sys(qsys, prio) is the function where the actual quantum part is performed. Here the code is sent to the QCPU and the return values are returned.
- A compiler should be implemented within or before perform_sys() (eg. OpenQL), which optimizes the code for NV centers or other quantum objects.

4 NetSQUID

The core components of the compiler should also optimize classical communication and the interconnected part of the network protocols. For this reason it seems incredible useful to give perform_sys() a simulate option on which the protocols are evaluated in NetSQUID and analyzed. In principle, assuming for instance that the 2020 demo is properly defined in NetSquid, it should also be able to handle arbitrary quantum code and analyze the effectiveness of the compiler.
Also for the real demo it would be useful to compute the expected theoretical results and then compare them from the actual. This feedback loop can therefore lead to optimizations in the compiler, the protocols and also the model of the NV center.