A Program Transformation and Architecture Support for Quantum Uncomputation

Ethan Schuchman and T. N. Vijaykumar

School of Electrical and Computer Engineering, Purdue University

{erys, vijay}@purdue.edu

Abstract

Quantum computing's power comes from new algorithms that exploit quantum mechanical phenomena for computation. Quantum algorithms are different from their classical counterparts in that quantum algorithms rely on algorithmic structures that are simply not present in classical computing. Just as classical program transformations and architectures have been designed for common classical algorithm structures, quantum program transformations and quantum architectures should be designed with quantum algorithms in mind. Because quantum algorithms come with these new algorithmic structures, resultant quantum program transformations and architectures may look very different from their classical counterparts.

This paper focuses on uncomputation, a critical and prevalent structure in quantum algorithms, and considers how program transformations, and architecture support should be designed to accommodate uncomputation. In this paper, we show a simple quantum program transformation that exposes independence between uncomputation and later computation. We then propose a multicore architecture tailored to this exposed parallelism and propose a scheduling policy that efficiently maps such parallelism to the multicore architecture. Our policy achieves parallelism between uncomputation and later computation while reducing cumulative communication distance. Our scheduling and architecture allows significant speedup of quantum programs (between 1.8x and 2.8x speedup in Shor's factoring algorithm), while reducing cumulative communication distance 26%.

Categories and Subject Descriptors: C.m [**Computer Systems Organization**]: Miscellaneous, D.1.3 [**Programming Techniques**]: Concurrent Programming - Parallel Programming.

General Terms: Algorithms, Performance.

Keywords: Quantum Computing, Uncomputation, QLA

1 Introduction

Quantum computing's power comes from new algorithms that exploit quantum mechanical phenomena for computation. These new quantum algorithms are different from their classical counterparts, in that quantum algorithms rely on algorithmic structures that are simply not present in classical computing. Just as classical program transformation and architectures have been designed for common classical algorithm structures, quantum program transformations and quantum architectures should be designed with quantum algorithms in mind. Because quantum algorithms come with these new algorithmic structures, resultant quantum program transformations and architectures may look different from their classical counterparts.

Previous work in architectures for quantum computation has not addressed the algorithmic differences between quantum and classical computations because they have taken a bottom-up approach [2, 5]. Previous work has been concerned with the bits and gates, staying close to the circuit level. These papers have evaluated how error-correcting operations should be mapped to basic gates, and where different types of quantum wires should be used [8]. Because these papers do not consider the algorithmic structures being executed, the papers advocate gate-array style architecture with some heuristic-based scheme for efficiently mapping a given algorithm to the array of quantum gates and wires.

This paper, on the other hand, takes a top-down approach, examining quantum algorithms and proposing a program transformation and architectural features best suited for these quantum algorithms.

Specifically, this paper addresses quantum uncomputation. The power of quantum computing comes from entanglement and quantum interference, and uncomputation is key to maintaining entanglement and engineering quantum interference. In quantum computing terms, entanglement implies that if two qubits (quantum bits) are operated on by a logic gate and one qubit is measured, the other qubit automatically collapses to the states that agree with the logic gate operation and the measured qubit value. Interference comes from the fact that the qubits are actually waveforms and can have negative amplitudes. Quantum logic gates can result in negative amplitude terms in a qubit's state, and these negative amplitude terms can cancel positive amplitude terms to remove a state from the list of possible measured states.

As quantum computation proceeds, it generates garbage qubits, just as in classical computing. To bound memory usage, these garbage qubits have to be reclaimed and reused. Unlike classical computing, the reuse cannot be a simple overwrite of the qubit with a new value.

In quantum computing, overwriting a qubit is not allowed because overwriting destroys information, releases energy into the environment, and collapses entangled qubits to agreeing values (just as a measurement would). To free an unneeded qubit while maintaining entanglement, quantum algorithms use *uncomputation* which returns a qubit to its initial value by undoing (i.e., performing in reverse) all the operations on that qubit. Because quantum computation proceeds without destroying information in order to avoid unintentional collapse of program state, all operations are reversible.

While often used to bound memory usage in quantum algorithms, uncomputation is also a key tool in engineering quantum interference and required for the most important quantum algorithms [9,6,10]. These algorithms work by using interference to reinforce desired solutions, and to cancel out incorrect solutions. Extra garbage qubits entangled with the qubits of interest prevent otherwise equivalent states from matching and interfering. Therefore, even if

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASPLOS06 October 21-25, 2006, San Jose, California, USA.
Copyright © 2006 ACM 1-59593-451-0/06/0010 $5.00.

Superposition	Qubits can be in states that are a linear combination of the base binary states 0 and 1.
Entanglement	When multiple qubits are logically related, measuring one qubit restricts possible measured values of the others so that all measured values reflect the logical relation.
Interference	The act of superposition of qubits states reinforcing some states and canceling out others.
Uncomputation	The process of performing additional operations to qubits to transform them from their current state back to an initial state in one of the base binary states (i.e., not a superposition).
Cloning	The process of making an exact copy of a quantum state and disallowed by physics.
Teleportation	The process of transporting quantum state from one qubit to another by exchanging only classical information.

Table 1: Definitions of key terms

free qubits are available, quantum algorithms still uncompute all temporary qubits. As such, at least half, but likely more (due to nesting of uncomputation leading to recomputation of uncomputation, as we explain later) of their operations performing uncomputation.

The key contributions of the paper are:

- We identify uncomputation as a major component of runtime in quantum computation, and argue that it should be treated as a first-priority concern in all quantum systems design.
- We show a simple program transformation that allows uncomputation to be executed in parallel with later computation to reduce the runtime.
- We propose a multicore architecture tailored to exploit this parallelism between computation and uncomputation of quantum algorithms. While previous quantum architectures exploit fine-grain parallelism in an FPGA-like architecture, we exploit the coarse-grain compute/uncompute parallelism in a multicore.
- We show how exposing this multicore architecture to the scheduler simplifies scheduling, and produces more efficient schedules. Previous scheduling schemes [5, 2] are oblivious of the structure of computation and uncomputation, and exploit unstructured, fine-grain, operation-level parallelism requiring greater communication distances than our structured, coarse-grain parallelism.
- While most of uncomputation and later computation are data-independent, the values to be uncomputed need to be communicated over a long distance between cores. We propose dedicated inter-core communication paths so that the long-distance communication across cores does not obstruct the short-distance communication within cores. Previous work [5] advocates a generic interconnect incurring the overhead of this obstruction.
- We evaluate the performance of Shor's factoring algorithm and show between 1.8x and 2.8x speedup in our parallelized version over the original algorithm, and 26% qubit-communication cumulative distance reduction when comparing our architecture to the previously-proposed gate-array architecture [5] running our parallelized version mapped to the gate array using automated tools from [2].

While this paper makes important contributions in describing how quantum architectures and quantum program transformations should be designed, some may say that our constant-factor improvements are insignificant considering that our improvements are on top of exponential improvements provided by the most basic unoptimized quantum computer. However, the exponential speedup from quantum computing is due to algorithmic innovation; algorithmic research, whether classical or quantum, aims at improving algorithmic complexity. Systems research, on the other hand, does not change algorithmic complexity but instead aims at executing a given algorithm as efficiently as possible. In algorithmic terms, systems techniques improve the constants. Systems research will remain a constant-factor pursuit in quantum computing as it has in classical computing, but is still worthwhile especially because device scaling (which will come when quantum computers become a reality) will afford constant-factor improvements every technology generation. These factors will accumulate over technology generations to result in large improvements, just as they have done for classical computers. In particular, [5] calculates that Shor's algorithm would take 21 hours to factor a 128-bit number. Thus, our technique which reduces runtime from 21 hours to 12 hours is a valuable contribution.

Because quantum operations and qubit movement are prone to error and qubits tend to decohere over time, error rates are an important consideration in quantum architectures. A previous work [2] argues that quantum architecture can have orders of magnitude impact on error rates. This large impact may make our improvements look small, although that work targets error rates and we target runtime. The previous work, however, does not show how architecture can improve error rate, but instead shows that movement of qubits hurts error rates and that a poor architecture can *worsen* error rates by five orders of magnitude than those previously calculated without architectural constraints. It is important to note that orders-of-magnitude negative impact does not imply potential for similar impact on the positive side. In particular, movement in [2] is so detrimental to error rates because large amounts of qubit movement occur after an operation *before* errors accrued during the operation are corrected. Qubit movement adds errors beyond those accrued during operations, allowing error rates within a single operation to get so high that correction is not possible. In addition, [2] does not use quantum teleportation, the long-distance version of quantum wires, which does not incur the high error rates of qubit movement. A later work, [5], shows an architecture that ensures that all movement before error correction are kept local, and uses quantum teleportation for distant communication after values are corrected. By minimizing movement, [5] is able to achieve error rates close to the previous calculations with a straight-forward architecture. Thus, [5] dispels [2]'s claim of orders-of-magnitude impact from the architecture alone.

The remainder of the paper proceeds as follows. Section 2 provides background on quantum computing and quantum uncomputation. Section 3 describes our quantum algorithm transformation that exposes parallelism in between quantum computation and quantum uncomputation. Section 4 describes an architecture and scheduling designed to efficiently support the exposed parallelism. Section 5 discusses our methodology. Section 6 presents results for our techniques. Section 7 discusses related work and Section 8 concludes.

2 Background

The following subsections provide necessary background and define terms that will be used in later sections. Additionally, we provide the definitions in Table 1.

2.1 Superposition and notation

Quantum bits (qubits) are different from classical bits because

they can exist in superposition states. This superposition state can be defined as a linear combination of the base binary states. Thus qubits can have states that are a combination of both 0 and 1. Qubits remain in this superposition state throughout a computation. Only when observed must the qubits take on a classical logical value of 0 or 1. The probability of which value a measured qubit assumes depends on how close the pre-measurement superposition was to a 0 or a 1.

Quantum computation on qubits can best be described by matrices representing basic logic gates, and vectors representing qubit state. The vector qubit state is often called a wave function and represented by $|\Psi\rangle$. A single qubit in an equal superposition of 0 and 1 would be represented as $|\Psi_1\rangle = (1/(\sqrt{2}))|0\rangle + (1/(\sqrt{2}))|1\rangle$. The coefficients in front of each logical value represent the square roots of the probabilities that each of the two state are observed when measured. In this case, a $|0\rangle$ and $|1\rangle$ each have a probability of 0.5 of being observed (the sum of all squared coefficients must sum to 1). Wave functions reflecting multiple qubit states are similar. If $|\Psi_1\rangle = (1/(\sqrt{2}))|0\rangle + (1/(\sqrt{2}))|1\rangle$ and $|\Psi_2\rangle = (1/(\sqrt{2}))|0\rangle + (1/(\sqrt{2}))|1\rangle$ then the product state is $|\Psi_{12}\rangle = (1/(\sqrt{4}))|00\rangle + (1/(\sqrt{4}))|01\rangle + (1/(\sqrt{4}))|10\rangle + (1/(\sqrt{4}))|11\rangle$. This wave function reflects that fact that if we measure both qubits we can obtain 1 of 4 different results, each with equal probability.

2.2 Entanglement

Multi-qubit wave functions reflect probabilities of observed values as above, but also represent quantum entanglement. Entanglement (briefly described in Section 1) can be more formally defined as the property that when multiple qubits are logically related, measuring one qubit restricts possible measured values of the others so that all measured values obey the logical relation. This fact is not quite expected because when the first qubit is measured it has freedom to choose its measured value as either a 0 or a 1 (with probabilities determined from its coefficients). For the second related qubit to guarantee agreement of its measured value, it must somehow be aware of which value the first qubit chose. As as example of a 2 qubit wave function representing entanglement consider $|\Psi_3\rangle = (1/(\sqrt{2}))|01\rangle + (1/(\sqrt{2}))|10\rangle$. Here $|\Psi\rangle$ represents two qubits where each qubit is in an equal superposition of $|0\rangle$ and $|1\rangle$, and the two qubits are each the invert of the other (the two qubits are entangled). When the first qubit is measured, it has equal probability of being a 0 or a 1. But then when the second qubit is measured, it must result in the invert of the value measured from the first qubit. Entangled states like $|\Psi_3\rangle$ arise due to logic gates. $|\Psi_3\rangle$ could be produced from initial qubit states $|\Psi_1\rangle = (1/(\sqrt{2}))|0\rangle + (1/(\sqrt{2}))|1\rangle$ and $|\Psi_2\rangle = 0|0\rangle + 1|1\rangle$ passing through a controlled not (CNot) gate. A CNot gate uses one qubit to enable the gate, and if enabled, the gate inverts the other qubit. If $|\Psi_1\rangle$ was used as the control qubit, the two qubits would be entangled by the operation and result in $|\Psi_3\rangle$ above.

2.3 Interference

In addition to entanglement, interference is another unique feature of quantum computing. Interference is the act of superposition of qubits' states reinforcing some states and canceling out others.

Interference comes from the fact that qubits have a phase and operations can result in wave function components with negative coefficients. As an example, consider the Hadarmard operation H which takes $|0\rangle$ to $(1/(\sqrt{2}))|0\rangle + (1/(\sqrt{2}))|1\rangle$ and $|1\rangle$ to $(1/(\sqrt{2}))|0\rangle - (1/(\sqrt{2}))|1\rangle$. Notice that $H(H(|0\rangle)) = (1/2)|0\rangle + (1/2)|1\rangle + (1/2)|0\rangle - (1/2)|1\rangle = 1|0\rangle$; interference causes the $|1\rangle$ basis to cancel out returning to a single base binary state with no superposition.

Interference is key for quantum computation, because quantum algorithms produce interference and cause unwanted solutions to be canceled out. When only desired solutions remain, the qubits can be measured and one of the desired solutions will be returned. In order for interference to work there must be identical but opposite sign wave function components. Unfortunately, unwanted temporary qubits can make otherwise equivalent wave function components different so they can not cancel. Accordingly, quantum algorithms that use interference need to clean up all their temporaries. As an example consider the entangled state $|\Psi_3\rangle = (1/(\sqrt{2}))|01\rangle + (1/(\sqrt{2}))|10\rangle$. If we write a wave function for only the leftmost qubit it looks like $|\Psi_4\rangle = (1/(\sqrt{2}))|0\rangle + (1/(\sqrt{2}))|1\rangle$. From above, we know that H applied to $(1/(\sqrt{2}))|0\rangle + (1/(\sqrt{2}))|1\rangle$ results in $|0\rangle$, so we might expect that if we applied H to the left most qubit in $|\Psi_3\rangle$ we would return the left most qubit to $|0\rangle$ by canceling out the $|1\rangle$ basis and obtaining $|\Psi_5\rangle = (1/(\sqrt{2}))|01\rangle + (1/(\sqrt{2}))|00\rangle$. In fact, what we get is $|\Psi_6\rangle = (1/(\sqrt{4}))|01\rangle - (1/(\sqrt{4}))|11\rangle + (1/(\sqrt{4}))|00\rangle + (1/(\sqrt{4}))|10\rangle$. $|\Psi_5\rangle$ and $|\Psi_6\rangle$ cause the leftmost qubit to return different values when measured. With $|\Psi_5\rangle$ measuring the leftmost qubit always results in 0, while with $|\Psi_6\rangle$ measuring the leftmost qubit results in both 0 or 1 with equal probability. Quantum algorithms need to clean up their temporary qubits if they intend to exploit interference.

2.4 Uncomputation

Cleaning up temporary values cannot be done by overwriting them. In quantum computing it is not possible to overwrite value, because overwriting destroys information, releases energy to the environment, and is consequently equivalent to measurement. Instead quantum algorithms use additional processing to clean up garbage qubits and return them back to their initial values.

This additional processing is called uncomputation. Uncomputation is defined as the process of performing additional operations to qubits to transform them from their current state back to an initial state in one of the base binary states. Because the current state can not be measured, the operations that uncompute a qubit are related to the operations that were previously applied to the qubit. Typically, uncomputation is done by taking all operations previously applied to a qubit and reapplying them in reverse (taking outputs back to inputs). Those qubits that are returned to an initial state that is not a superposition and has no dependence on the state of other qubits are thus disentangled.

If the only purpose of uncomputation was for preparing qubits for interference and measurement, uncomputation would be done once at the end of the algorithm. In that case, because no overwriting is allowed, algorithms cannot reuse memory and the memory footprint would grow out-of-control quickly. To address this problem. uncomputation is used to play another role in addition to engineering interference, and that is to bound memory usage. Unneeded qubits are uncomputed and freed as the program runs. Figure 1 shows how uncomputation is used to uncompute a temporary value. In the example, function f_1 reads q_0 as an input and converts a blank qubit to $f_1(q_0)$ (f_1 passes q_0 through to avoid destroying information). We use $f_1(q_0)$ as input to f_2, but when f_2 completes, $f_1(q_0)$ is no longer needed. At this time we can uncompute $f_1(q_0)$ by running f_1 a second time, but this time backwards (reverse of f_1 is denoted by $!f_1$ in Figure 1). At the end of uncomputation of f_1, only q_0 and $f_2(f_1(q_0))$ exist. If the algorithm is complete and ready for measurement, all is well; but if the algorithm must continue and later decides to free $f_2(f_1(q_0))$, there is a complication. To uncompute $f_2(f_1(q_0))$ to 0 we must run f_2 backwards, but running f_2 backwards requires $f_1(q_0)$ which we have already uncomputed. Consequently, to uncompute $f_2(f_1(q_0))$ we have to reverse all three

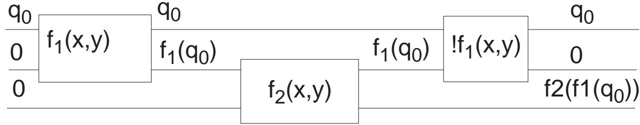

FIGURE 1: Basic uncomputation

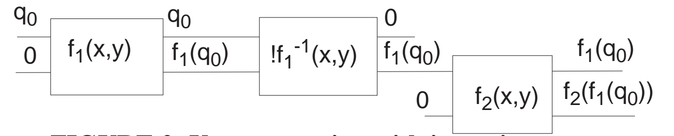

FIGURE 2: Uncomputation with inversion

blocks in Figure 1. We must run $!!f_1$ (*!!* means reverse-reverse, and $!!f_1$ equals f_1), $!f_2$, and $!f_1$, which amounts to recomputing f_1, reverse-computing f_2 and then reverse-computing the recomputed f_1. As the computation grows and we need to uncompute more times, each uncomputation grows. In general, uncomputing the ith computation step would recompute and uncompute the previous i-1 steps, quadratically increasing the work.

To combat this growth, [9] uses a different form of uncomputation. [9] relies on the use of an inverse function. Note that the inverse F^{-1} of a function is different than the reverse of a function *!F*. If $F([0,x]) = [F(x), x]$ and $F^{-1}([0,F(x)]) = [x,F(x)]$ then $!F^{-1}([x,F(x)]) = [0,F(x)]$. Using $!F^{-1}$ allows us to uncompute F's input (i.e., turn x into 0) while keeping F's output ($F(x)$) to be used for later computations. Figure 2 shows the same example as Figure 1 using inversion where f_1 uses q_0 as input to produce $f_1(q_0)$ and then $!f_1^{-1}$ is used to uncompute q_0 while keeping $f_1(q_0)$. Notice that if $f_1(q_0)$ were no longer needed after $f_2(f_1(q_0))$ is produced, $f_1(q_0)$ could be uncomputed by executing only $!f_2^{-1}$ without recomputing f_1. This important observation means that we can uncompute many times without needing to recompute previous uncomputation so that uncomputation does not grow with the computation length but remains comparable to computation. Thus, inverse-based uncomputation amounts to roughly doubling the work even with frequent uncomputation. As such, inversion based computation is a key tool in such quantum algorithms as order finding, factoring, and discrete logarithms [6, 9].

2.5 Cloning

Cloning is defined as the process of making an exact copy of a quantum wave function and is disallowed by physics [6]. The fact that qubits can not be identically copied is a consequence of the uncertainty principle which says the more we know about one physical property, the less we know about another. If we could make copies of a qubit, we could make many copies and measure precisely a single physical property from each copy allowing us to deduce precisely all the properties of the original qubit.

2.6 Teleportation

Teleportation is defined as the process of transporting quantum state from one qubit to another by exchanging only classical information. Teleportation is an important tool in quantum computing because physical movement is a major source of qubit error and physically moving a qubit is much slower than sending a classical bit via electrical signals. A teleportation channel is created by entangling two *carrier* qubits and sending them to distant points — one at the source, and the other the destination. A *data* qubit can be transmitted by interacting it with the source carrier qubit, measuring the interacted qubits, and sending the classical result of the measurement to the destination where it is used as the final step to transform the destination carrier qubit into the data qubit. Because the carrier qubits are entangled, measurement of the source carrier qubit transforms the destination qubit's state into a function of the data to be transmitted. For complete recreation of the data, we need to know what that function is so it can be inverted (obtaining the actual data) at the destination. Fortunately the classical values obtained from measuring the qubits at the source reveal the function, allowing exact recreation of the quantum state by sending only these classical values to the destination. Because the classical information is sent from the source to the destination, teleportation does not exceed the speed of light limit, as required by physics.

A key point here is that the carrier qubits can be entangled and sent to source and destination without using the value to be transmitted, and hence can be done well before the value is ready. Thus, the carrier set-up can be hidden from the communication critical path. When the value is ready, the interaction with the source carrier qubit, the sending of the classical information, and the transformation of the destination carrier qubit are the only actions in the critical path. These actions are much faster and much less error-prone than physical movement (especially over long distances). That is why teleportation is the choice tool for communication over long distances.

3 Program Transformation

In this section we describe our program transformation that allows us to execute reverse computation of one function in parallel with forward computation of another.

3.1 Transformation approach

Using Figure 2 as reference, we want to run $!f_1^{-1}$ in parallel with f_2. We can see both $!f_1^{-1}$ and f_2 need $f_1(q_0)$. In classical we could simply pass $f_1(q_0)$ to both functions, but this requires copying and in quantum copying is disallowed (i.e., physically infeasible) by the no-cloning theorem (Section 2.5). It would seem that one of the most important techniques for achieving parallelism in classical computing is not allowed in quantum computing.

We make the key observation that to achieve parallelism between $!f_1^{-1}$ and f_2 it is not actually a copy that is needed but rather another, physically feasible, well-known quantum operation called *Fanout*.

If it were physically possible, a quantum copy operation would take $\Psi_4 0\rangle = (\alpha|0\rangle + \beta|1\rangle)|0\rangle$ to $|\Psi_4\Psi_4\rangle = (\alpha|0\rangle + \beta|1\rangle)(\alpha|0\rangle + \beta|1\rangle)$. Notice that when multiplied out $|\Psi_4\Psi_4\rangle = \alpha^2|00\rangle + \alpha\beta|01\rangle + \alpha\beta|10\rangle + \beta^2|11\rangle$. Notice that the copy operation would cause no entanglement between the two qubits and if one of the qubits were measured, the state of the other qubit would still be undetermined. A Fanout operation, instead, takes $|\Psi_4 0\rangle = \alpha|00\rangle + \beta|10\rangle$ to $|\Psi_5\rangle = \alpha|00\rangle + \beta|11\rangle$. Notice that $|\Psi_5\rangle$ is not equal to $|\Psi_4\Psi_4\rangle$ and therefore Fanout is not a copy operation. But also notice that the probability of each qubit being measured as 0 or 1 are equal in $|\Psi_4\Psi_4\rangle$ and $|\Psi_5\rangle$. For example, the probability that the left qubit in $|\Psi_4\Psi_4\rangle$ is 0 is $\alpha^4 + (\alpha\beta)^2 = \alpha^4 + \alpha^2(1-\alpha^2) = \alpha^2$ which is equal to the probability that the left qubit in $|\Psi_5\rangle$ is 0. Fanout does copy qubit probabilities correctly, but introduces entanglement artifacts that keep it from making *independent* copies. Fanout can be implemented using a simple bitwise CNot operation with the qubit to be fanned-out as control, and 0 as the other input.

In quantum computing, entanglement is an important component of the computation and should be thought of as one of the key components of any input or output of a function. Function results depend not only on the input qubits wave function coefficients, but also on any entanglement between the input qubits and other qubits used in the function. Therefore, if we want to make two functions parallel,

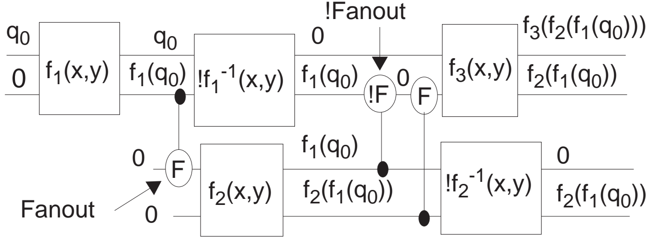

FIGURE 3: Using fanout for parallelism

that were originally sequential we must ensure input wave function component probabilities are maintained, but also that any entanglement is maintained. It turns out that the entanglement artifacts from Fanout make this assurance. Because the fanned-out qubit is entangled to the original, the fanned-out qubit is transitively entangled to all the qubits to which the original is entangled. As an example, consider three qubits a_0, a_1 and a_2, where a_0 is entangled to a_1. If we fanout the three qubits to b_0, b_1 and b_2 then b_0 is entangled to a_0, b_1 to a_1 and b_2 to a_2. Because b_0 is entangled to a_0, a_0 is entangled to a_1, and a_1 is entangled to b_1, b_0 is transitively entangled to b_1. Furthermore, although b_2 is entangled to a_2, a_2 is not entangled to either a_0 or a_1, and therefore b_2 is not entangled to b_0 or b_1. Consequently, whether a function operates on qubits a_0 through a_2 or b_0 through b_2, the function will produce the same results. Therefore, in Figure 2 we can fanout $f_1(q_0)$ and one of the fanouts can go to $!f_1^{-1}$ and the other to f_2.

We point out the subtle but key reason why our fanout strategy works. The combination of uncomputation followed by !Fanout returns one of the fanout qubits to its initial state (*0* in our example). Thus, the uncomputed fanout qubit is disentangled from the other fanout qubit ($f_1(q_0)$), which can be used by other computation (f_2) without any problems of unwanted entanglement. However, if we were to fanout a qubit into two and then expect to perform independent computation on the pair, to achieve general parallelism for instance, then that would not work because the pair is entangled and cannot be operated on independently. It works here only because the specific operation we perform — uncomputation followed by !Fanout — disentangles the fanout pair.

The remaining constraints on parallelism are similar to those in classical computing including the requirement that any functions executed in parallel must modify different qubits. For our purpose of overlapping computation with uncomputation this requirement is automatically satisfied because one function is producing a value to be used in later computation, and the other function is removing a value no longer needed.

Figure 3 depicts how Fanout allows uncomputation to be overlapped with later computation. First, $f_1(q_0)$ is produced from q_0. $f_1(q_0)$ is then fanned-out using a free qubit. One of the fanned-out pair is used by $!f_1^{-1}$ to uncompute q_0. The other is used to produce $f_2(f_1(q_0))$. Notice that $!f_1^{-1}$ and f_2 can execute in parallel. When $!f_1^{-1}$ and f_2 are complete, we still have a pair of $f_1(q_0)$. It is important that we reduce the pair back to a single qubit before running $!f_2^{-1}$. We can reduce the pair to a single qubit by preforming the *!Fanout* operation (which is Fanout in reverse) on the pair. The result is $f_1(q_0)$ and a 0 qubit. The remaining $f_1(q_0)$ is uncomputed by $!f_2^{-1}$. It is important that the !Fanout is performed *before* $!f_2^{-1}$ because otherwise we would be left with a single version of $f_1(q_0)$ and no easy way to get rid of it. We could not do an !Fanout then because there would be only a single version, and the only alternative would be to run the more-heavy-weight $!f_2^{-1}$ again.

The above-described procedure can be made easily into a general program transformation that can be applied to a string of function calls. After each function call (for forward computation), code should be inserted to Fanout each qubit of the function's output. The fanned-out qubits are used for further computation as well as for uncomputing the function's inputs. The corresponding !Fanout operation should be placed immediately before the point where the program uncomputes the function's outputs.

One final issue is load imbalance. The uncomputation step may not be equal exactly to the next computation step — i.e., $!f_1^{-1}$ and f_2 may not be perfectly load-balanced. This load imbalance depends upon how different the inverse function is from the forward function. This load imbalance will introduce some inefficiency and reduce speedups. However, in practice, f_1 and f_2 are often successive iterations of a loop performing the same computation on different data, and therefore are likely to be naturally load-balanced, and if f_1 and $!f_1^{-1}$ are similar in computational weight, then this transitively makes $!f_1^{-1}$ and f_2 to be naturally load-balanced.

3.2 Implementation

We do describe how the transformation could be automated using preexisting language features in QCL, a high-level quantum programing language [7].

To automate our transformation the **c**ompiler must consider a quantum program like the one in Figure 4a and decide whether a function is performing an uncomputation and nothing else (i.e., distinguish between functions performing uncomputations and others performing regular computations). Once such a function is identified, our transformation can parallelize the uncomputation with later computation. To isolate uncomputations, the compiler has to identify (1) where and which qubits are being returned to the zero basis state (have been uncomputed), and (2) ensure that the function does not change other qubit values but only uses them for the uncomputation.

Fortunately, QCL already provides the language features needed in the form of types for function arguments, which allow the compiler to achieve both the above requirements. Currently the compiler uses these types for type checking but we extend the use for our transformation.

QCL uses a *quvoid* argument type to mark that an argument must take qubits that are initialized to the zero basis state as input (i.e., the function needs an empty register to "write to"). At output, the quvoid argument is typically not in the zero state. Figure 4b shows example function definitions using the quvoid type arguments. Notice that the quvoid arguments are used as the destination for "assignments". In reality, we do not assign to the quvoid register (recall that no overwriting is allowed in quantum), but manipulate the known zero input state into the intended output. Currently, the compiler uses the quvoid type to detect bugs of the form where a function is called with a output register that is not known to be in the zero basis state. We note that the quvoid argument is sufficient to allow the compiler to identify uncomputation. Consider what happens when a function that has a quvoid argument is called in reverse. Because the function is being called in reverse, the quvoid argument need not be zero basis, but the output of the function must be zero basis (because the input of the forward function is zero, so the output of the reverse must be zero). In other words, the function performs an uncomputation to return the quvoid argument back to the zero basis. Therefore, the compiler can infer that a reverse call to a quvoid function would result in the quvoid arguments being uncomputed to the zero basis, satisfying the first requirement above**.**

To address the second requirement**,** QCL also provides the *quconst* type to function arguments that allows the compiler to infer

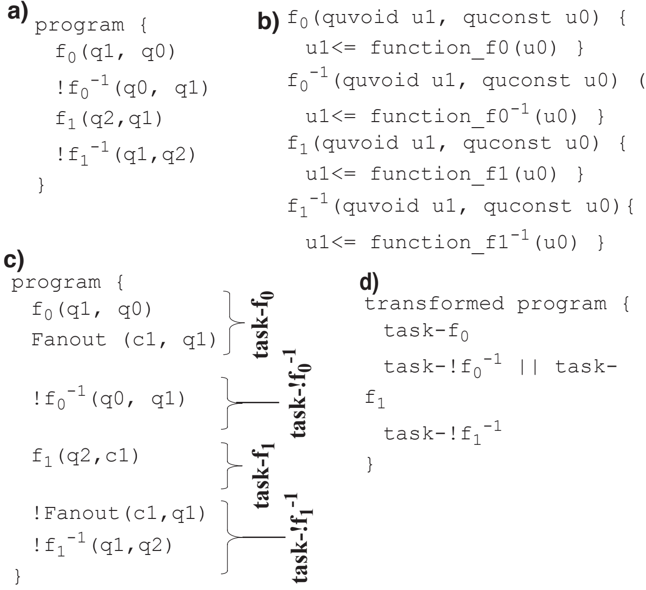

FIGURE 4: Example depicting transformation: a) top-level program, b) function definitions, c) automatically-generated tasks to pass to scheduler, (d) transformed program at task level

when a function is using the argument only as input. Figure 4b shows four functions that use a quconst to determine the output that they compute in the quvoid argument. The compiler can be sure that any arguments marked quconst are not modified by the function, thus satisfying our second requirement.

Once the two requirements are verified as described above, we consider how the compiler takes a program like the one in Figure 4a and transforms it into the one in Figure 4d that can later be easily scheduled for parallel execution on our multicore architecture.

When the compiler examines the string of function calls in Figure 4a and their arguments' type declarations in Figure 4b, the compiler can infer that *q0* is being uncomputed by $!f_0^{-1}$. The compiler also infers a true data dependence between f_0 and $!f_0^{-1}$ through only q1 because the quvoid *q1* can be modified by f_0 but not the quconst q0. Similarly, there is a true dependence between f_0 and f_1 through only q1. Thus, the compiler infers that f_0's outputs (q_1) is being used by the uncomputation $!f_0^{-1}$ and later computation f_1. Accordingly, as described in Section 3.1, the compiler inserts a Fanout for each of f_0's outputs, in this case q1, so that one copy is used by $!f_0^{-1}$ and the other by f_1. After the Fanout, f_1 and $!f_0^{-1}$ can run in parallel. Finally, the compiler inserts a !Fanout to merge the fanout pair just before the function that uncomputes q_1 itself (i.e., $!f_1^{-1}$), as described in Section 3.1 ($!f_1^{-1}$ is identified as an uncomputation just as $!f_0^{-1}$ is).

In addition to using the data dependence information for these insertions, the compiler also uses the information to demarcate tasks that can be scheduled for parallel execution on our quantum multicore architecture. Figure 4c shows how operation are divided into tasks. task0 performs f_0 and fans out q_1. task1 uncomputes q_0. task2 computes f_1 with the fanned out copy, and task3 performs the cleanup of the extra fanned out copy using q_1 and then does the uncompute of q_1. After the transformation, task1 and task2 are dependent on task 0 but are independent of each other, and task3 is dependent on task1 and task2. The compiler records this parallelism information as depicted in Figure 4d to be used by our scheduler in Section 4.3.

Figure 5 shows a pseudocode algorithm for the transformation example described above. The pseudocode assumes a program structure of alternating compute/uncompute functions as shown in Figure 4a, but extension to other program structures is possible. *transform* begins with startup code that performs the first task and generates a fanned out pair. In steady-state, *transform* appends a Fanout operation at the end of the second task generated by each iteration of the *for* loop. The next iteration uses the fanned out pair (one for each of the two following tasks), and undoes the fanout that was performed two iterations ago by prepending a !Fanout to the start of the first task generated by each loop iteration. *transform* also marks that the two tasks in each *for* loop iteration are parallel. After the top-level function has been transformed, *transform* is executed on each of the functions called within the top-level function. Thus transform marks tasks and parallelism in the top-level function but also in nested function calls so that it is possible for computation and uncomputation to be overlapped at multiple call depths, allowing potentially two raised to the call depth speedup.

Because there are overheads associated with our transformed code, it is important that transformation is only performed when it is feasible and beneficial considering hardware constraints. Because quantum computation can use teleportation to prepare ahead of time for communication, there is little latency penalty for synchronizing parallel tasks (unlike classical communication). In addition the latency introduced by instrumented code is very small. A Fanout of an n-bit register is n parallel simple CNot gates, and therefor the additional Fanout and !Fanout for each function is insignificant. While the latency penalties are small hardware requirements are likely to limit how much parallelism should be exposed. More parallelism clearly requires more computational resources, and thus the balance between achievable speedup and additional hardware must be considered in performing transformation

3.3 Implications

We note, that the cost of this runtime improvement afforded by this transformation is a similar constant factor increase in number of qubits. This increased qubit footprint is an obvious outcome of

```
transform(func) {
  emit(start task0);
  emit(func.funcCall[0]);
  emit(fanout func.funcCall[0].out, copy);
  emit(end task0);
  for (n=1 to number of function calls, n+=2){
    emit (start parallel task);
    emit (start task(n)); //uncomputation
    if (func.funcCall[n-2].copy exist)
      emit(!fanout func.funcCall[n-2].out,
              func.funcCall[n-2].copy);
    emit(func.funcCall[n] using copy);
    emit(end task(n));
    if (func.funcCall[n+1] exists) {
      emit(start task(n+1)); //next forward computati
      emit(func.funcCall[n+1], func.funcCall[n-1].ou
      emit(fanout func.funcCall[n+1].out,
              func.funcCall[n+1].copy);
      emit(end task(n+1));
    }
    emit(end parallel tasks);
  }
  for (n=0 to number of function calls)
    transform(func.funcCall[n]);
}
```

FIGURE 5: Pseudocode for transformation

exploiting parallelism. While today, device physicists are worried about constructing a handful of qubits, we argue that for practical quantum computing, scalable solid-state implementations have to become a reality. Once quantum hardware can scale to practical sizes, scaling an addition factor of 2 or 4 is not an impediment.

We point out that this increased qubit footprint will not lead to worsened error rates. Errors are a problem only when they accumulate to the point that they can not be corrected (i.e., they exceed the error correction codes strength) [6], Errors that occur in parallel computation are corrected before the results are combined (every logical gate has error correction built in, as explained in Section 4.1), and therefor do not accumulate to become irrecoverable. In fact, our reduction in runtime likely leads to better error rates because shortening runtime also shortens live-ranges of quantum values thus reducing decoherence time that occurs between the end of one correction and the start of another operation (which will end in another error correction). Consequently, the qubits have higher fidelity at the start of the second error correction.

4 Architecture

In this section, we describe our technique for mapping parallel computation and uncomputation to quantum hardware. We then consider hardware optimizations tailored for the resulting communication patterns. We first include some background on the previous work in quantum architecture upon which our technique builds.

4.1 Background

We build on the architecture proposed in [5]. In quantum computation, errors are extremely common and every operation must be performed fault-tolerantly. Error Correction Codes (ECCs) are used to encode logical qubits into many physical qubits (e.g., one logical qubit may need seven or more physical qubits). Fault tolerant gates operate on logical qubits; each fault tolerant gate mapping to many non-fault-tolerant gates operating on the physical qubits. At the end of a logical gate, incorrect physical qubits are corrected resulting in error-free logical qubits.

Because physical movement of qubits introduces errors, the architecture proposed in [5] minimizes physical movement. In particular, [5] uses an FPGA-like substrate, called Quantum Logic Array (QLA), which consists of many fault tolerant gates arranged in an array and interconnected with teleportation channels in a grid fashion. [5] *locally* maps all the physical gates within one logical fault-tolerant gate so that physical movement within the fault-tolerant gate is kept within a small area. For communication from one gate to another, [5] relies on teleportation.

Recall from Section 2.6 that teleportation works by entangling two carrier qubits, keeping one at the source and sending the other to the destination. This sending is done well before the data to be transmitted is even produced so that the carrier set up is hidden under data production and is not on the teleportation critical path. When the data is ready, it is interacted with the source carrier qubit which is then measured, forcing the destination carrier qubit into a state which is a function of the data. The actual data is recreated by applying the classical value obtained from the measurement to the destination carrier qubit. The interaction, measurement, sending and applying of classical information are much less error-prone than physical movement.

There is one complication left: From our description, it may seem that the carrier qubits themselves physically move through the large distance between source and destination. Such movement would corrupt the carrier qubits making teleportation unreliable and slow (the

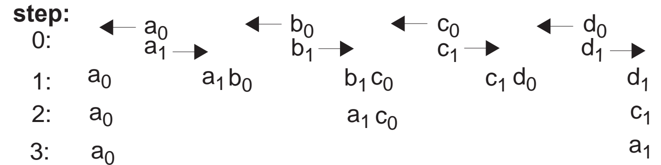

FIGURE 6: Preparation for teleportation

setup would be long and would require much overlap with other work). [5] shows a strategy to address this problem. [5] breaks the distance between source and destination into *n* hops separated by islands, where physical movement occurs *only* within one hop, as shown in Figure 6. In step 0, each hop uses a pair of carrier qubits, entangles the pair, moves one of the pair to the source end of the hop and the other to the destination end of the hop. At end of the step, there is a string of *n* pair-wise entangled carrier qubits. Next, to communicate from one hop to the adjacent hop, a set of *n/2* teleportations are done. These teleportations interact the carrier qubits from every other pair of adjacent hops (e.g., a_1 is interacted with b_0 and c_1 with d_0) and apply the classical information of the interaction to transform the entangled counterpart (e.g., a_1b_0 interaction is applied to b_1 to transform b_1 into a_1). At the end of this step, there are *n/2* carrier qubit pairs each of which span two hops (as shown in step 1). This process continues where every step halves the number of carrier qubit pairs and doubles the spanned distance, until there is one pair that spans the entire distance from source to destination.

To summarize, this strategy permits long-distance communication across many hops even though each qubit moves only one hop in step 0. In addition, all these preparatory steps (steps 0-3) are done before the data to be sent is actually ready. After teleportation, the used carrier qubit pairs return to their source island to be repaired so that they can be reused for another teleportation. [5] handles errors due to the physical movement over one hop with a technique called distillation [6] which discards corrupted qubit pairs not suitable for teleportation. Qubit pairs discarded due to distillation have the effect of increasing the number of qubits that must be sent between hops. Pairs lost due to distillation must either be compensated by sending more pairs at a time (higher inter-hop bandwidth) or by increasing the teleportation setup time during which enough good pairs are accumulated from multiple rounds of distillation.

4.2 Our quantum multicore architecture

While previous proposals, such as QLA, have advocated using a homogenous sea-of-gates FPGA-like architecture, we propose that the FPGA-like substrate be split into multiple, well-defined cores each of which contains a subset of the original gates and internal communication (teleportation) channels. The cores are also connected with inter-core teleportation channels. It may seem that by creating partitions in the hardware our multicore architecture introduces granularity boundaries and reduces the flexibility of the homogenous, non-granular FPGA-like QLA. However, with our transformation, coarse-grain parallelism becomes a fundamental part of quantum algorithm structure and our multicore architecture matches this coarse-grain parallelism better than QLA. As such, it is hard for a fine-grain gate-level scheduler to map this coarse-grain parallelism on to the homogenous QLA hardware (even if we assume that the fine-grain scheduler uses our transformed code). Our multicore granularity makes it easier for the scheduler to achieve speedups and at the same time reduce communication bandwidth.

The coarse-grain parallelism between uncomputation and later computation maps well to multiple cores, and implies that uncompu-

tation runs on one core and the later computation on another. Most of the communication is *intra-core,* corresponding to dependencies within functions. Accordingly, each core is optimized for many short intra-core teleportations, borrowing from [5] which employs physical movement only within a logical gate and uses teleportation among gates. However, there is a small amount of communication needed between the cores. This need for *inter-core* communication arises from the fact that both cores need the fanouts of the output of the previous computation to proceed (e.g, q_1 which is f_0's output, is needed by both f_1 and $!f_0^{-1}$ in Figure 4), and only one of the cores could be running f_0 and could avoid communication of q1 while the other core must obtain the other fanout of q1 via inter-core communication. Similarly, after the cores execute in parallel, the fanouts need to be merged via a !Fanout, requiring another inter-core communication.

These communications being inter-core are necessarily over long distances.While the teleportation scheme described above provides a method for long-distance communication with low error rate and almost no latency, there still are disadvantages to long communication. Each teleportation must allocate a number of carrier qubits proportional to the communication distance (i.e., number of hops), as shown in step 0 of Figure 6. To form a teleportation channel, each hop must have as many good pairs after distillation as the size of the message to be transmitted. If each hop has probability p of having enough correct pairs for transmission then the probability of all n hops having enough correct pairs is p^n. These qubits are not available to be allocated for any other communication until the current teleportation is complete and the qubits return to their source islands. If enough carrier qubits are not available, the teleportation channel will not complete preparation before data is ready for transmission and the data qubit will have to wait. Length of communications matters because either we must provide higher bandwidth between each hop to increase p (i.e., more carrier qubit pairs needed at each island), or we must wait longer to accumulate enough good pairs across each hop for the later teleportation steps, as discussed in Section 4.1. The second option is easier than the first.

Because the inter-core communication arises from the uncomputation-computation parallelism which is fundamental to quantum computing, we propose dedicated inter-core communication channels to optimize this communication. Because these long communications are infrequent, and occur at the *ends* of functions, the communication preparation time (i.e., teleportation setup as described in Section 4.1) can be hidden under the execution of the functions. Accordingly, the dedicated channels can have low bandwidth connections, and can spend the entire function execution time to prepare the channel and still allow almost latency-free communication at the end of the function (this is the easier, second option mentioned above for long-distance communication). By providing dedicated long-distance channels, we prevent the carrier pairs being held idle in preparation of the long distance channel from obstructing other close intra-core communication within the function. While our architecture distinguishes between intra-core and inter-core communication, the original homogenous, non-granular QLA does not make this distinction. The lack of this distinction implies that short and long communication are handled alike and may obstruct each other.

We point out that our architecture scales as quantum devices become denser. Until all levels of loop nesting of computation-uncomputation parallelism are exploited, doubling the hardware would double performance.

4.3 . Scheduling on our multicore architecture

One possibility for mapping our transformed computation to quantum gates in the QLA would be to use a fine-grain scheduler such as [2]. [2] has created a toolset for mapping quantum operations to such quantum gate arrays. The tool takes a backwards pass through the program graph scheduling ready operations as close as possible to their consumers to minimize the amount of time a value waits to be consumed (and thus reducing accumulation of error by decoherence). The scheduler is oblivious of the algorithmic structures of computation and uncomputation and optimizes the low-level communication among gates. While such a scheduler is capable of scheduling our exposed parallelism with reasonable speedup (we will see this in our results), in this section we describe how scheduling at the high-level granularity of computation and uncomputation can be more efficient in terms of communication distance, and hence bandwidth (Section 4.2).

If we were to use the scheduler in [2] to schedule our transformed computation on the QLA, then parallelism created by our transformation creates a difficulty for the scheduler. It is difficult for the scheduler to determine if two independent operations should be scheduled close together to minimize communication when their dependence chains join (the operations though independent of each other do belong to their respective dependence chains), or if the dependence chains will be independent for long enough to amortize the cost of executing them far apart at the benefit of leaving free resources for other dependent operations. This difficulty arises because the scheduler is oblivious of the high-level structure in computation and uncomputation and the coarse-grain parallelism between them. Therefore, while we allow the scheduler to schedule fine-grain operations *within* computation and uncomputation function calls on each core, we explicitly schedule the coarse-grain computation and uncomputation function calls themselves to separate cores knowing the high-level parallelism between the previous uncomputation and the next computation (Section 3). That is, we solve the scheduler's difficulty by knowing that the dependence chains within computation and uncomputation are long enough, and the communication between a previous uncomputation and the next computation is small enough that they can be scheduled far apart. In contrast, the original scheduler in [2] does not schedule independent operations as far apart as our approach, forcing other operations in their dependence chains to be farther away and incurring higher bandwidth.

In addition, the knowledge of the coarse-grain parallelism between computation and uncomputation allows our scheduling scheme to take advantage of the dedicated inter-core communication channels in our multicore architecture which the previous scheduler does not have in its homogenous QLA architecture. Because these dedicated channels exist for inter-core communication and are not needed for intra-core communication which occurs through intra-core teleportation channels, our scheduler can hold up these channels for long preparation time without causing a shortage of communication channels, allowing low-bandwidth channels to be sufficient (Section 4.2). In contrast, the original scheduler in [2,5] does not allow long preparation time because doing so would block other communication through its homogenous communication channels. Instead, the scheduler must either rely on high-bandwidth channels or delay communication. Thus, our architecture makes bandwidth-efficient scheduling easier, as mentioned before.

At the coarse-grain level we exploit only the known parallelism, between previous uncomputation and later computation, and nothing else. Any lost opportunity for additional coarse-grain parallelism seems negligible.

Figure 7 shows a pseudocode description of the scheduling algorithm we use to schedule on our multicore architecture. The algo-

```
schedule(height, task, freeCores) {
  coreSet1=allocateNcores(2^(height-1), freeCores);
  coreSet2=allocateNcores(2^(height-1), freeCores);
  toggle=1;
  if (task==null) {
    return;
  } else if (height==0) {
    execute(task, freeCores);
  } else {
    foreach step in task {
      if (toggle) {
        schedule(height-1, step.compute, coreSet1);
        schedule(height-1, step.uncompute, coreSet2);
      } else {
        schedule(height-1, step.compute, coreSet2);
        schedule(height-1, step.uncompute, coreSet1);
      }
      toggle=!toggle;
    }
  }
}
```

FIGURE 7: Pseudo-code for scheduling

rithm takes as inputs the tasks with their dependence information as demarcated by our transformation in Section 3.2. The algorithm starts with a top level task and allocates two sets of cores onto which it allocates its parallel tasks. There are two sets because we support only two-way parallelism at each level (overlapping of uncomputation and later computation at that level) keeping in mind that the overall parallelism is two raised to the number of levels. As the scheduler schedules its tasks to the two sets of cores it recurses into each task's subtasks similarly partitioning cores for two-way parallelism. When a task schedules its sequence of steps it toggles the set of cores it uses for forward and reverse computation at each step. This toggling causes corresponding forward and reverse functions to execute on the same cores and, as we discuss later in this section, reduces communication.

To discuss scheduling in detail we walk through an example scheduling depicted in Figure 8. Figure 8a depicts the tasks output by our transformation (described in Section 3.1). Notice that this program exhibits two levels of nesting of uncomputation (one at the program level and the other within each function call), thus the transformation was preformed recursively to depth two. In addition, the program uses inverse-based uncomputation as we discussed in Section 2.4. This generalized program structure is apparent in multiple quantum algorithms [10,9, 6], including Shor's algorithm. In Shor's algorithm [9], the dominant computation phase is an exponentiation on a quantum register by iteratively performing multiplications (corresponding to each task-f_x in Figure 8), with each multiplication iteratively performing additions (corresponding to each *task-g_x*, *task-h_x*, *task-i_x*, and *task-j_x* in Figure 8). Thus the generalized example in this section translates directly to Shor's factoring algorithm which will be evaluated in Section 6.

Notice that the input to the scheduler already specifies dependencies and thus the scheduler needs only to schedule the specified parallelism in a favorable way.

The scheduler's first step, is to map *task-f_0* (the only ready task) to computational resources. Because *task-f_0* has internal parallelism of up to 2 internal tasks, the scheduler allocates two cores (cores 1 and 2) to *task-f_0*. The scheduler then examines *task-f_0*, and maps the first subtask (*task-g_0*) to core 1 (either core 1 or core 2 could have been chosen) at time step 0. *task-g_0* finishes at the end of time step 0, having produced output that is required by both *task-$!f_0^{-1}$* and *task-f_1*. The scheduler sees that when task-g0 finishes *task-$!f_0^{-1}$* and *task-f_1* are ready to execute in parallel. At this point we have to make a decision as to where to schedule task-$!g_0^{-1}$ and where to schedule g_1. One option is to schedule all the forward computation in *task-f_0* in core 1, and all uncomputation of *task-f_0* in core 2. The other option is to alternate between computation and uncomputation on each core. It turns out to be more bandwidth-efficient to alternate between computation and uncomputation on each core. If we were to schedule *task-g_0* and *task-$!g_0^{-1}$* to different cores we would have to send not only the fanned out copy of the output of g_0 across the core, but also the qubits that are to be uncomputed. Recall from Section 2.4 and Section 3 that *task-$!g_0^{-1}$* uncomputes *task-g_0*'s inputs and needs both the inputs and outputs of *task-g_0*. Instead if we schedule *task-g_0* and *task-$!g_0^{-1}$* on the same core we don't have to transmit the qubits to be uncomputed, and we transmit (via teleportation) only one of the fanouts of *task-g_0*'s output to be used by task-g_1 on the other quadrant. Therefore, the scheduler schedules *task-g_0* and *task-$!g_0^{-1}$* to core 1 and *task-g_1* to core 2. Upon completion of *task-$!g_0^{-1}$* the fanned-out input is sent to core 1 where *task-$!g_1^{-1}$* is scheduled. The fanned-out input is uncomputed by an inverse fanout with its fanned-out pair in *task-$!g_1^{-1}$* before *task-$!g_1^{-1}$* actually performs $!g_1^{-1}$ to uncompute the

a)

```
transformed program {
task-f0
task-!f0^-1 || task-f1
task-!f1^-1
}

task-f0 {
task-g0
task-!g0^-1 || task-g1
task-!g1^-1 || task-g2
task-!g2^-1
}

task-f1 {
task-h0
task-!h0^-1 || task-h1
task-!h1^-1 || task-h2
task-!h2^-1
}

task-f0^-1 {
task-i0
task-!i0^-1 || task-i1
task-!i1^-1 || task-i2
task-!i2^-1
}

task-f1^-1 {
task-j0
task-!j0^-1 || task-j1
task-!j1^-1 || task-j2
task-!j2^-
}
```

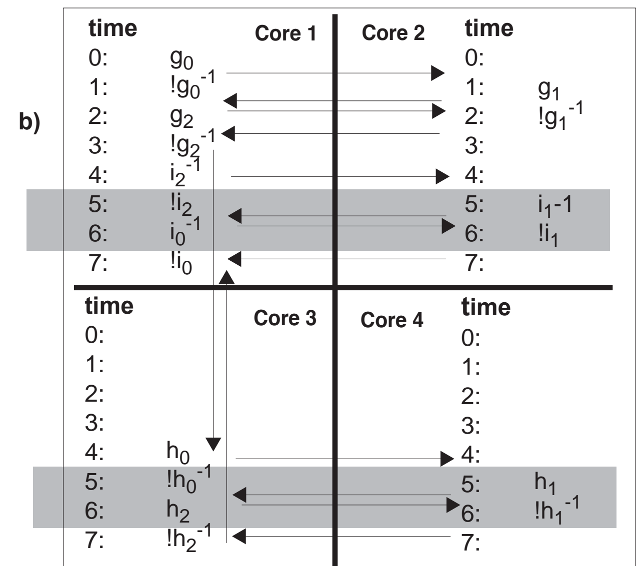

FIGURE 8: Scheduling: a) code b) mapping to cores

other member of the pair. Therefore, each time step contains one transmission of the fanout qubit in each direction, one for the next step of computation, and the other to be reunited with its pair and uncomputed. The scheduler continues this process of alternating computation and uncomputation on core 1 and core 2 until we have completed forward execution of *task-*f_0.

When the forward execution of f_0 completes at the end of time step 3 (with the completion of *task-*$!g_2^{-1}$), dependence information from the transformation specifies that task-f_1 and *task-*$!f_0^{-1}$ are ready for execution. Similar to scheduling internal to *task-*f_0, the fanout of output of *task-*f_0 is sent to *task-*f_1 in core 3 and keep the other in core 1 for *task-*$!f_0^{-1}$. At time step 4, the scheduler allocates core 1 and 2 for *task-*$!f_0^{-1}$, and core 3 and 4 for *task-*f_1 (note that the code shown for task-f_0^{-1} in Figure 8(a) is regular, forward code but the execution shown for *task-*$!f_0^{-1}$ in Figure 8(b) is in reverse). Similar to the scheduling internal to *task-*f_0, we keep *task-*f_0 and *task-*$!f_0^{-1}$ local to the same set of cores so that qubits being uncomputed do not have to be moved. Scheduling of *task-*$!f_0^{-1}$ to the top 2 cores and of *task-*f_1 to the bottom 2 cores is performed similar to that of *task-*f_0, as per the dependence information provided by the transformation pass.

All the long arrows in Figure 8(b) correspond to inter-core communications. Recall that these communications occur through dedicated channels that are low bandwidth because the communication occurs at the end of the function so that the communication preparation can be slow and occur throughout the function execution.

Also notice that at time steps 5 and 6 (shaded regions), four tasks are executing concurrently. This overlapping of four tasks into one time step allows us to achieve up to a 4x speedup. However, there are two issues that may reduce this speedup: (1) As mentioned in Section 3, load imbalance among the concurrently executed tasks may reduce this speedup. (2) Figure 8 shows that we do not attempt to overlap starting and ending functions at each nesting level. Consequently, in this example we see only two time steps where four functions are executing concurrently. Real algorithms have many functions at each nesting level, making the lack of overlap of starting and ending functions insignificant and achieving speedups close to 4x with two levels of nesting and good load balancing.

5 Methodology

We evaluate our techniques on Shor's quantum factoring algorithm and circuit as presented in [9]. Shor's algorithm represents the most important class of quantum algorithms that provide polynomial-time solutions to classically exponential problems. Other quantum algorithms with similar speedup have similar structure [10,9, 6].

We use QCL, a quantum programming language and interpreter. We hand-modify a QCL implementation of Shor's factoring algorithm, making our transformations in the static code. Runtime of Shor's algorithm is dominated by a modular exponentiation that is composed of modular multiplications (corresponding to each task-f_x is Figure 8) each composed of modular additions (corresponding to each *task-*g_x, *task-*h_x, *task-*i_x, and *task-*j_x in Figure 8). Specifically, we apply our transformations to allow parallelism between computation and uncomputation of the multiplications, and computation and uncomputation of the additions within those multiplications.

Once we have our transformed algorithm, we use the QCL interpreter for two purposes: First, to actually simulate quantum programs on classical hardware (slowly) and verify that the transformed program's results match the original program's results, and second, to generate a complete trace of all quantum gate operations performed in the program.

After collecting the program trace, we use QUALE [2] to schedule operations onto a quantum gate array. We modify QUALE slightly to model the QLA architecture proposed in [5]. Specifically we allow latency-free teleportation, and collect statistics on cumulative communication distance. We assume there is ample bandwidth to prepare all teleportation channels. We evaluate speedup due to our program transformations by using QUALE's standard mapping procedure to map and simulate execution of both the baseline algorithm, and our transformed algorithm to quantum gate arrays. We simulate three different sizes of the modular exponentiation (8, 16, and 32 bit modulus) performed by Shor's factoring algorithm. Modular exponentiation is by far the dominant component of Shor's quantum operation runtime.

To compare our coarse-grain multi-core function-level scheduling to fine-grain single-core operation scheduling, we create two quantum gate arrays. One is a large, uniform array sized to hold enough qubits required for execution of the entire program,. and the other is composed of four smaller cores each sized to hold only enough qubits required for execution of the modular additions. We arrange the cores into a 2x2 array. We use QUALE to evaluate the runtime and cumulative distance for both cases. In quantum computing, communication distance is related to teleportation bandwidth requirement (as discussed in Section 4.2), and thus provides an important metric for comparison. For the uniform array, we use QUALE to schedule directly on the array. For our scheduling, we use QUALE to schedule within the cores, and use our own scheduler to schedule the QUALE-scheduled functions to our cores. We schedule functions as described in Section 4.3, scheduling each multiplication to either the top two or bottom two cores, and scheduling the additions to one of the two cores within either of the two halves. To calculate cumulative distance we assume square gates and measure transmission distances in gate lengths traversed. We make the simplifying assumption that communications between add functions cross the width of one of the cores in the x direction, and cross half of the height of the cores in the y direction. Similarly, for communications between multiplications, we assume a distance equal to the height of cores in the y direction and half the core width in the x direction.

6 Results

In the following two subsections we present our results, first for speedup due to our transformation and then communication distance due to coarse-grain scheduling to our multicore architecture.

6.1 Speedup

As described in Section 5 we evaluate our results on Shor's factoring algorithm. The runtime of Shor's algorithm is dominated by a modular exponentiation that performs a^b mod n where a and n are classical integer numbers and b is a quantum register holding a superposition of all possible integer values. The result of the computation is another quantum value. This exponentiation is composed of modular multiplications and corresponding uncomputation of these modular multiplications. Similarly, each multiplication is composed of modular additions and corresponding uncomputation of these modular additions. The nesting of additions inside computation provides the opportunity to shorten both multiplications by hiding addition uncomputation time, and exponentiation by hiding multiplication uncomputation. These two levels of uncomputation provide the potential for up to 4x speedup.

Figure 9 presents our results for speedup due to our transformation. Black bars show the speedup due to parallelizing the computa-

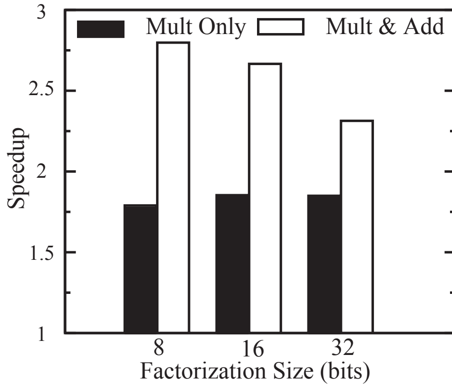

FIGURE 9: Parallelization speedup

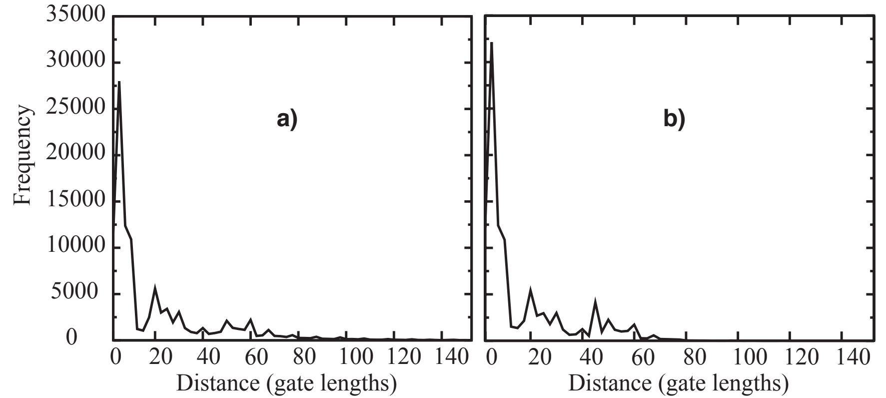

FIGURE 10: Frequency of teleportation distances for a) previous fine-grain scheme and b) our coarse-grain scheme

tion and uncomputation only at the level of multiplications, while white bars show speedup due to parallelizing both at the level of multiplications and additions. The three bars in each set represent results for factoring different size numbers ranging from 8 to 32 bit.

From the black bars, we see that parallelizing multiplication produces significant speedup. At 8 bits, parallelization provides 1.78x speedup, while at 16 and 32 bits, the speedup is 1.85x. The additional speedup moving from 8 to 16 bits is the result of the startup and finish overhead becoming insignificant with larger size input. Also note that while moving from 16 to 32 bits, speedup is maintained showing that parallelization of multiplication will be valuable for larger input sizes. The fact that speedup for 16 and 32 bits does not reach its maximum potential of 2x is a result of natural load imbalance between one iteration of multiplication computation and a previous iteration of multiplication uncomputation. We explain later why this load imbalance remains constant with input size.

From the white bars, we can see that also parallelizing the additions within each multiplication can result in significant additional speedup, but this additional speedup diminishes with the input size. For 8, 16, and 32 bit inputs, speedup is 2.8x, 2.6x, and 2.3x. This decreasing speedup with increasing input size occurs because additions have a variable number of quantum operations and the number is dependant on *classical* inputs.

Quantum programs can have two kinds of conditional statements, those conditioned on classical bits, and those conditioned on qubits. A classical *if* block can be skipped if the classical *if* condition evaluates to false. Quantum *if* conditions, on the other hand, can not be evaluated because doing so would require measurement of the conditional qubits and destruction of the computation. All statements internal to the quantum *if* block must be executed assuming they are conditioned on a superposition of both *true* and *false*, and consequently, all operations inside a quantum *if* block are always executed.

The load imbalance between addition computation and addition uncomputation occurs because the addition function uses classical *if* statements to add a classical value to a quantum register in ripple-carry style. If the *if* condition is true then a single-bit addition is performed otherwise the bit is skipped. Fewer 1's in the classical input imply fewer qubit manipulations and result in shorter runtime for the addition. If concurrent addition computation and addition uncomputation operated on similar classical inputs there would be good load balancing. While all nearby addition computations contain similar numbers of operations, and nearby addition uncomputation contain similar numbers of operations, in most cases uncomputation is long when computation is short and vice versa. This trend occurs because while a single modular addition may use x as input, its inverse modular addition (used for uncomputation) uses n-x as classical input. In other words, if x is large, forward computation is long and reverse computation is short. As n grow the difference between x and n-x grows larger and load imbalance increases.

While addition is load balanced poorly because of classical *if* statements, multiplication is load balanced well because it contains only quantum *if* statements. Multiplication uses each bit in the quantum exponent register b as a condition to a quantum *if* statement surrounding a single call to an addition. Each of the quantum *if* statements is always evaluated causing each multiplication computation and multiplication uncomputation to always have as many additions as the length of the b quantum register and in turn proportional to the size of number to be factored.

It may seem that because the additions are not balanced well, and the multiplication's runtime is determined by a sequence of calls to additions, the multiplications would also be unbalanced, but this is not the case. Although each addition computation is not of similar run-time to its co-scheduled addition uncomputation, the sum of runtimes of all additions within the multiplication is similar to the sum of runtimes of overlapping multiplication uncomputation leading to good balancing and significant speedup even as input size grows.

These results show that there will be levels of nested uncomputation where parallelization between computation and uncomputation produces large speedup and levels where the speedup is much lower and may not be worth the added qubit footprint (Section 3.2). Determining which levels depends on the algorithm structure, and the nature of the inversion function as we described.

6.2 Communication distance

In this subsection we discuss the effect of our scheduling scheme on communication distances. We show results for factoring an 8 bit number with the fine-grain, gate-level mapping (FM) as done in [2] and with our coarse-grain mapping (CM) to our multicore architecture. The total number of teleportations in FM is 107133 and in CM the total number is 107176. The number of teleportations correspond to the number of instances of dependencies. Because our mapping does not change dependencies the number of teleportations are similar in the two schemes. However, the total distance traversed by qubits in FM is 2121292 gate lengths, while qubits in CM traverse 1560966 gate lengths. These numbers present a 26% reduction in cumulative communication distance. This reduction means that teleportation channels can prepare for transmission faster and need to hoard fewer qubits during preparation (as discussed in Section 4.2). In addition, because fewer qubit pairs are used in a given transmis-

sion, more qubit pairs are available to be used for other transmissions leading to a reduction in contention.

To help explain the improved communication distances we compare FM and CM in detail in Figure 10a and Figure 10b respectively. The x-axes are the distance of communication, and the y-axes are the frequency of communication.

The most significant difference between the two occurs at the range from 3 to 5 gate lengths. In this range, FM has 27991 teleportations and CM has 32140 teleportations (the graphs have similar shape, but the peak heights are different). This 15% increase in the number of short range teleportations does not reflect a net increase in number of teleportations, but instead shows that CM is able to convert longer teleportations into shorter ones. In exchange for shortening many teleportations, CM has to increase some longer teleportations. Specifically, including communication between additions, CM performs 4135 teleportations that traverse 43 gate lengths and including communications between multiplications CM performs 2262 teleportations that traverse 49 gate lengths. These additional teleportations create significant peaks in Figure 10b, but these peaks are overshadowed by the distance savings discussed above.

CM performs better than FM because FM's greedy allocation does not map parallel functions well. FM tries to greedily place each operation as close to its consumer as possible. This greedy allocation is not favorable when considering parallel execution of computation and uncomputation on a fanned out value. The consequence of attempting to minimize each individual movement is that both parallel function get mapped to the same area near their common produced value. If there were frequent communications then this mapping would be favorable, but these functions communicate only at their start and end. The disadvantage is that the two parallel functions compete for the closest free gates resulting in both functions having to reach further to find free gates not taken by the other function. In addition, FM may find operations in functions that are ready due to sub-critical paths even though other inputs to the functions are not ready due to the actual critical path. In this case FM still schedules these operations as close to their consumers as possible, tieing up close resources so that they can't be used and providing almost no performance improvement. Because CM schedules at the function level, ready operations in later functions will not be scheduled until the rest of the function is ready.

7 Related Work

Uncomputation was originally proposed as a tool for proving that computation under bounded memory could be performed without consuming energy, called *adiabatic* computing [3]. Since then, other work [12,1] has considered performance, energy, and complexity trade-off techniques in adiabatic computing. Such techniques can optimize for complexity or performance by leaving a few bits not uncomputed with the only consequence being a small energy loss. Quantum algorithms require that *all* extra bits have to be uncomputed, making such trade-offs impossible. Recall that quantum algorithms do uncomputation not only for reclaiming memory, but also to engineer quantum interference for correctness. We are the first to show that even in quantum computing where complete uncomputation is required, techniques at the program and architectural level can improve performance by efficiently hiding uncomputation behind computation.

Previous work in quantum interpreters and compilers includes QCL [7] and Qubiter [11]. QCL provides a language and interpreter well suited for development and functional simulation of quantum algorithms but unlike our work, does not perform any optimization. Qubiter, although termed a quantum compiler, works closer to the logic synthesis level decomposing matrices representing complex quantum operations into multiple simple operations that can be mapped to a basic quantum gates.

Other orthogonal work has prosed techniques for optimizing algorithms within the forward computations inside quantum algorithms[4]. Such techniques still require uncomputation, and our techniques can be applied on top of theirs to hide their uncomputation behind their computation.

8 Conclusions

Quantum computing provides new opportunities for program transformations and architectures because quantum algorithms rely on algorithmic structures not present in the classical world. This paper has shown how program transformations and architecture can provide support for one of the key algorithmic structures, quantum uncomputation.

By recognizing uncomputation, we are able to expose parallelism between uncomputation and later computation in quantum algorithms. We have proposed a quantum multicore architecture tailored to this compute/uncompute parallelism, and shown how a scheduler can efficiently map the algorithm to our proposed architecture.

We have evaluated our techniques on Shor's factoring algorithm and shown speedup ranging from 1.8x to 2.8x with a 26% reduction in cumulative communication distance.

Acknowledgments

We thank Saman Amarasinghe and the anonymous reviewers for their valuable comments.

References

[1] W. C. Athas and L. J. Svensson. Reversible logic issues in adiabatic CMOS. *In Proceedings of Physics and Computation*, 1994.

[2] S. Balensiefer, L. Kreger-Stickles, and M. Oskin. An evaluation framework and instruction set architecture for ion-trap based quantum micro-architectures. In *Proceedings of the 32nd ISCA*, 2005.

[3] C. Bennett. Logical reversibility of computation. *IBM Journal of Research and Design*, 1973.

[4] R. V. Meter and K. M. Itoh. Fast quantum modular exponentiation. *Physical Review A*, 71:052320, 2005.

[5] T. S. Metodiev, D. Thaker, A. Cross, F. T. Chong, and I. L. Chuang. A quantum logic array microarchitecture: Scalable quantum data movement and computation. In *Proceedings of the 38th Micro*, 2005.

[6] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[7] B. Omer. *A Procedural Formalism for Quantum Computing*. Ph.D. thesis, Technical University of Vienna, 1998.

[8] M. Oskin, F. T. Chong, I. L. Chuang, and J. Kubiatowicz. Building quantum wires: the long and the short of it. In *Proceedings of the 30th ISCA*, June 2003.

[9] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, 1994.

[10] P. W. Shor. Why haven't more quantum algorithms been found? *J. ACM*, 50(1), 2003.

[11] R. R. Tucci. A rudimentary quantum compiler(2nd ed.), 1999.

[12] C. Vieri. Pendulum: A reversible computer architecture. Master's thesis, MIT, 1995.