A practical quantum programming framework and its highly-parallel virtual execution

Yves Vandriessche
Software Languages Lab
Vrije Universiteit Brussel
Email: yvdriess@vub.ac.be

Ellie D'Hondt
Software Languages Lab
Vrije Universiteit Brussel
Email: eldhondt@vub.ac.be

Theo D'Hondt
Software Languages Lab
Vrije Universiteit Brussel
Email: tjdhondt@vub.ac.be

***Abstract*—We present a practical quantum programming framework based on the formal framework of the Measurement Calculus and powered by our high-performance virtual execution engine: the Quantum Virtual Machine. Within this work, we employ various software engineering and high performance computing techniques in order to facilitate the design, composition, transformation, verification and execution of measurement-based quantum programs. Our first contribution lies in the design and construction of the framework itself. We present a complete prototype of the framework, which implements its entire layered architecture, viz. a complete quantum 'software stack'. Each layer deals with a separate concern, following the logical division lines in the semantics of the Measurement Calculus. Our second contribution is located in the execution layer of the framework: the formulation of the inherent parallelism in measurement-based quantum computing simulation and its high-performance implementation in the form of the Quantum Virtual Machine.**

I. Introduction

Formal frameworks can be powerful tools to assist the discovery and development of quantum computing applications. Generally speaking, these can be more useful in a virtual environment, where operations that are tedious when performed by hand are automated, in which the formal tools can be applied in analysis and verification of executable quantum programs and where applications can be developed interactively. Broadly speaking, there is a need for a Quantum Programming Paradigm: a collection of quantum programming tools, models and approaches. However, a programming paradigm is not constructed, but is grown over time by accumulating the contributions of a diverse expert community [1]. Pragmatically speaking, there is first and foremost a need for a practical quantum programming framework: an environment in which to interactively write, test and analyze quantum programs. In other words, there is a need for a programming framework that combines a low-level execution environment with current higher level formal tools and techniques.

Our approach to building a practical quantum programming environment is to first create a small working kernel with all the necessary elements from top to bottom, but which is designed from the start to be extended and grown in multiple directions. To this purpose, we build our first artifact as a series of abstraction layers. Such an abstraction layer approach is ubiquitous in todays computer software architectures.

In the Quantum Computing domain, there already exist executable languages, automated formal models and a great many low-level quantum computing simulators. Indeed, many quantum computing implementations have been developed over the years, both high and low-level. However, there has been a lack of an overarching approach or quantum computer model, one that is used for both higher-level formal work as well as low-level execution environments. The circuit model typically fills the role of quantum computing model for practical low-level implementations. Although, because of the semi-formal nature of quantum circuits, many implementations have to create their own specific formalization and architecture. This creates a moving target for practical high-level frameworks and for the low-level implementation environments themselves. A solution to this issue is to use an overarching formal model, which both offers a practical low-level execution model, but also offers a bridge to current higher-level formal frameworks.

II. Measurement Calculus

For our quantum programming virtual framework, we base ourselves on the formal framework of the Measurement Calculus (MC) by Danos et al. [2]. The Measurement Calculus brings together several interesting properties that make it highly suited for our above stated purposes. First, the formal basis for our framework should be *physically realistic*; when programmable quantum computers become available, our quantum programming framework should remain relevant. Physical scalability arguably poses the biggest threat to the building of physical quantum computers. The MC is a rigorous formalization of the Measurement-Based Quantum Computing (MBQC) model [3], [4]: a radically different paradigm of quantum computing which aims to achieve this physical scalability. Secondly, the operational foundation of the MC provides us with a small yet universal set of simple operations: a *quantum assembly language*. Its clean and rigorously defined operational semantics simplifies the task of constructing a practical measurement-based quantum computing simulator Thirdly, the MC introduces a *compositional* and modular abstraction on top its operational foundation: measurement patterns. An important contribution of the MC was to show that this abstraction enables local reasoning. Finally, the Measurement Calculus is *relevant*: it is currently being used as a research tool to drive new discoveries [5]–[7]. In summary, we use the Measurement Calculus because of its advantages as a conceptual framework, but also as a practical framework; building a quantum programming framework requires a combination of both.

We identify, for our purposes, two distinct abstraction layers in the Measurement Calculus: a pattern layer that concerns itself with composing patterns into larger wholes and a execution layer that deals with executing the individual operations. The execution layer deals with the manipulation *at runtime* of computational state, while the pattern layer manipulates measurement patterns *at design* time. This separation will form the basis of the layered architecture of our programming framework.

III. Framework

A. Layered Architecture

The structure of the proposed layered architecture is schematically represented in fig˙ 1. Each layer deals with different concerns:

- **Application layer:** MC programs need to be integrated into classical programming languages in order to build useful applications. In the application layer, the measurement patterns are connected to the programming environment at large. We envision and implement two ways in which this can happen: a graphical design tool for non-programmers and a pattern library extension for programmers.
- **Pattern layer:** The pattern layer provides pattern composition functionality, but also implements a pattern assembler to transform a given pattern into an concrete and executable command sequence.
- **Execution layer:** Given a command sequence, the execution layer orchestrates the execution of its MC commands, taking care of the required classical computations.
- **Realization layer:** The realization layer virtually or physically executes an individual quantum operation.

Each layer is developed and implemented separately, as a distinct abstraction, library or even executable computer program. It is fundamental to the layered architecture design that each layer implementation can be changed, extended and even substituted without affecting any other layers.

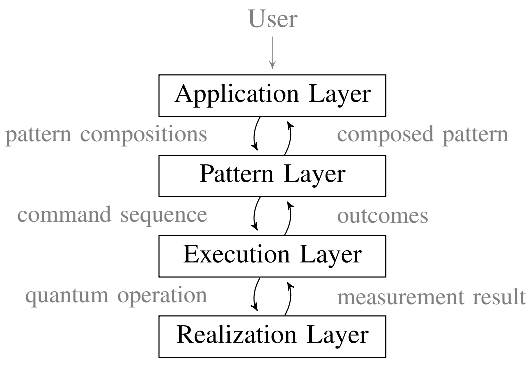

Figure 1. Overview of the layered architecture structure.

Given the scope of this text, we will briefly focus on the composition and execution layers: the layers that implement and automate the MC. More details, such as formalization and detailed semantics, we refer to Vandriessche's dissertation on the topic [8].

B. Composition

Measurement patterns and their composition rules were already introduced by the MC as an abstraction mechanism, but require some changes in order to be automated. The current notational convention works well for small patterns. However, performing and even defining larger pattern compositions by hand is a tedious and error-prone process; one needs to manually match qubit names correctly in an entire measurement pattern. The two rules for parallel and composite pattern composition from [2] cannot be trivially automated as they stand. It is implicit that qubits get renamed in order to achieve the correct result and to satisfy the rule's preconditions. We omit pattern standardization, an important aspect of the MC, in the discussion here; First, the standardization algorithm can be applied without fundamental changes. Second, as we will discuss in section III-C below, a measurement pattern is best left in a non-standardized (wild) state due to performance considerations.

We replace qubit names by logic variables and replace the renaming process by variable matching. While this may seem more complex, it allows for easier automation of the composition rules and separates the way the composition is expressed from the underlying composition rules. Indeed, this separation is crucial to allow several alternative ways of expressing patterns and their composition. We wish to stress that this formulation of measurement patterns is not meant to be used by humans, but rather by a computer program from the application layer. The composition functionality using this new basic representation is implemented by what we call the *generalized composition*, which provides a consistent pattern definition and composition abstraction to the application layer above. We present the composition functionality in three stages: *expressing pattern composition*, *renaming* and *merging*. Additionally, a *pattern assembly* process breaks down the pattern structures internal to the pattern layer into concrete command sequences, which can be passed to the execution layer for execution.

C. Execution

The execution layer deals with executing MC command sequences. One can create an interpreter or *quantum virtual machine* for executing command sequences by directly implementing MC's operational semantics. These have been defined with enough rigor that this poses no significant problems. In the context of our work, it is important to note that implementation here means a virtual or simulated execution environment, rather than a physical one. This gives rise to some practical concerns, on which we will focus here. As such, we introduce some amendments to the operational semantics of the MC, in order to deal with the practical concerns of a virtual execution environment. The main issue being: there is a practical upper limit to the number of entangled qubits one can represent in a simulated environment.

Taking the MC's operational semantics as-is, one would need to prepare a single, global and entangled quantum state before the start of the computation. Taking the amplitude-vector state representation of a quantum state[1], the size of the vector blows up exponentially by the number of entangled qubits represented. This limits in practice the total number of qubits that can be used in any command sequence, which is disastrous considering the number of qubits used in any typical measurement-based quantum computation. We work around this upper limit by the combining of three elements: *lazy state preparation*, *lazy tensor combination* and *wild patterns*. First, we only prepare (initialize) a qubit's quantum state when the qubit is first needed. Second, we represent the quantum state as a set of entangled quantum states, rather than a single entangled state. Finding and factorizing any given general state is a difficult problem. However, we can conservatively keep quantum states factorized by construction. Two states are combined (tensored) only when necessary: right before the execution of an entanglement operation targeting two of their qubits. Third, we keep patterns in a *wild* state. In fact, measurement (destruction) of a qubit should follow its first mention (creation) as early as possible. This last property of a pattern is the exact opposite of the standardization of a pattern[2]. The first two elements are small amendments to the MC's operational semantics, the third is a preference of input programs.

We implemented one such execution layer as an optimized and high-performance quantum virtual machine (QVM) in the C programming language. This artifact has been made public[3]. A large wild pattern, of 3288 commands, realizing the quantum Fourier transform on 16 qubits can be executed by the QVM in 0.5 seconds, using a modern Intel Xeon processor.

Dealing with the exponential blowup means on the one hand being performant, not wasting given computational resources such as to enable tackling larger problems in a realistic timescale. On the other hand, one can also throw more computational resources at the problem. Today, for both cases, this translates into providing an efficient parallel program.

IV. Parallel formulation

We provide a parallel formulation of the simulated execution of MC command sequences. This is achieved by manipulating the tensor formulation of the involved quantum operations and quantum states. The purpose of these manipulations are to put the operations in a form that will trivially map to a parallel computation model: the dataflow model. Dataflow started as a formal model which uses graphs to represent parallel computation [10], but it has grown into a widely accepted practical parallel programming model [11] as well, with applications in parallel computer architectures [12]–[14] and programming languages technology [15], [16]. It is important to note at this point that we are dealing here with *classical* parallel computations rather than quantum-level parallelism. Put differently, we want to run as much amplitude-level arithmetic operations simultaneously, instead of quantum-level operations.

The first step is to consider the tensor product formulation of single-qubit operations. Given an amplitude vector implemented using a column-vector, qubit positions in the tensor product combination become important. A MC command X_4 operating on a column-vector quantum state translates to some matrix operation $I^{\otimes m} \otimes X \otimes I^{\otimes n}$. Matrix operations formulated as such tensor or Kronecker products have along the diagonal a certain repeating block structure. The smaller such n, the smaller the repeating block's size and the more it is repeated. When X is completely at the 'right' side of the product ($n = 0$), one obtains a perfectly data-parallel formulation of the operator X. That is, applying the $I^m \otimes X$ operator on a vector state is equivalent to applying m-times the two-by-two X matrix operator *in parallel*, on contiguous two-element vectors of the input vector state. For this reason, we will refer to this preferred tensor product position as the *parallel position*.

By itself, the Kronecker product is not commutative. However, there is a class of permutations that does permute any given Kronecker product [17]. There is thus a permutation matrix P_n such that

$$I^{\otimes m} \otimes X \otimes I^{\otimes n} \equiv P_n^{-1} \left(I^{\otimes m+n} \otimes X \right) P_n \quad .$$

Put in other terms, there is an operation which realizes a cyclical shift of qubit positions. Each one-qubit and two-qubit operators in the MC have a simple dataflow graph realization. Given that a parallel position operator can be trivially obtained by concatenating the dataflow graph of the involved single (or dual) qubit operator a number of times, we thus obtain a parallel formulation of every necessary quantum operation in their parallel position form. The permutations necessary to put operations in the parallel position are realized on such dataflow graph by permuting the input output edges. As such, we can construct a highly-parallel dataflow graph for any given MC command sequence. An example of such dataflow graph is given in Figure 2.

This parallel formulation of the virtual execution of MC command sequences opens up formal analysis of the involved parallelism, but also provides practical benefits in the form of steering high-performance parallel implementations. Analysis of formal parallel metrics already reveal that theoretically, there is enough available parallelism to offset the exponential increase in computational cost with an exponential amount of parallel computing resources. In practice, we observe that parallel implementations are dominated by communication cost,

[1]We use the more general amplitude-vector state representation. A virtual execution environment based on an alternative, such as the stabilizer calculus [9], requires a fundamentally different approach, which is out of the scope of our current work.

[2]It would be interesting to explore an *anti-standardization* process to drive a given pattern as wild as possible. That is, a pattern transformation that keeps the width of the intermediate entangled quantum states as low as possible.

[3]https://github.com/yvdriess/qvm

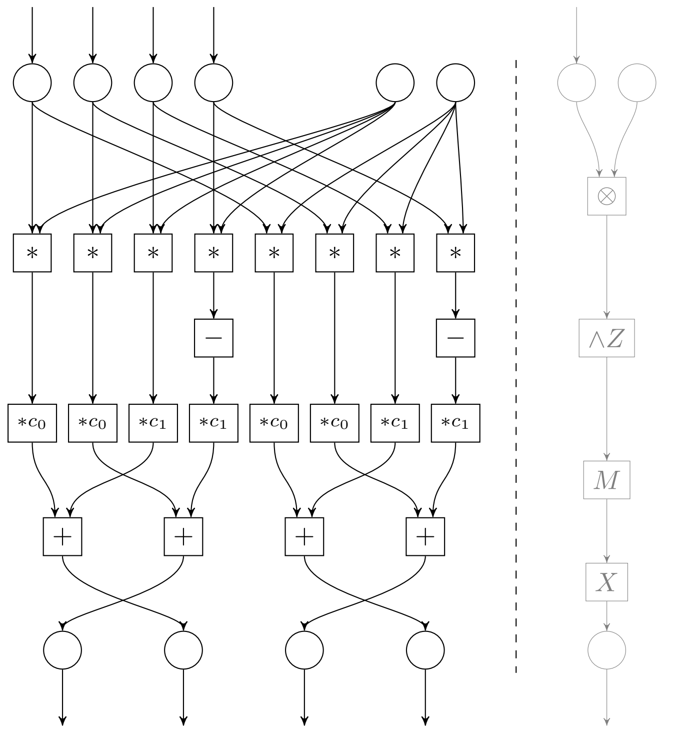

Figure 2. Fine-grained dataflow representation (left) of the command sequence $X_2^{s_1} M_1 E_{1,2}$ implementing the Hadamard transform, acting on a two-qubit input state. The schema on the right is a visual lead and a compact representation of the involved operators and quantum states.

either network communication in computer cluster configurations or memory bandwidth and latency in shared-memory multiprocessors. Here again, the dataflow formulation is key to optimizing communication and obtaining optimal parallel performance.

V. Discussion

We present here some of the results and insights gained by the construction of a practical top-to-bottom measurement-based quantum programming framework. The layered architecture of this framework was not only invaluable for developing the entire artifact itself, but lends itself particularly well as an experimental platform. We will discuss several acquired insights and avenues of experimentation here.

The limitations introduced by having to simulate the execution of measurement-based quantum operations has opened the door to changes or additions to some of the formal framework. Consider the standardization process for measurement patterns; a practical simulated execution of a MC program requires not only on a change to the MC's operational semantics (e.g. our lazy preparation and tensor combination amendment), but also requires patterns to be constructed in a way as to be as wild as possible. It is foreseeable that future physical measurement-based quantum computer prototypes have other such limitations that will have impact on multiple conceptual layers. For example, rather than the potentially infinite number of parallel operations that can currently be performed within the formal framework, it is more likely that a prototype physical implementation that is limited to a certain number of simultaneous quantum operations. Such operation schedule, a set of simultaneous operations, could very well have additional constraints, such as positional constraints where no two neighboring physical qubits can be manipulated simultaneously. Our layered framework provides a context where such changes to the formal and operational models can be explored: new constraints can be checked and enforced within the execution environment, new pattern-level transformations (e.g. an alternative standardization algorithm) can be added to the pattern layer. This type of research is prevalent in the programming languages community, where new language-level constructs, physical constraints and new assumptions are tested and validated by modifying an existing layered architecture.

As an example of this type of research support, we can point to previous work wherein a formal framework for distributed quantum computing [18] is supplemented by a practical framework. This practical framework was obtained with only minimal changes to the basic framework presented in this text. The pattern and execution layers were extended 'horizontally', that is, without breaking or changing existing functionality. We thus demonstrate that our practical framework can act as an experimentation platform where one can explore, iteratively work out and test new formal framework concepts.

Our main research vision, however, is one where new research results and practical insights are aggregated though integration with our practical framework. As mentioned in the introduction, the underlying agenda is to jump-start a measurement-based quantum programming paradigm, which needs to aggregates the knowledge of a wide expert community over time. Towards this goal, we have created a practical prototype with a heavy emphasis on extensibility, such that new concepts can be added with relative ease. We are confident that recent and useful research results can be added to the current framework: standardization [2], the automated calculation of depth complexity [6], finding the existence of flow to detecting unitary embedding [19] and generalized flow for deterministic patterns [20], detecting and executing parts of measurement patterns as stabilizer calculus operations, transforming measurement patterns to quantum circuits [21] and (the other way around) transforming quantum circuits into measurement patterns.

A more speculative future research direction lies in the unification of the dataflow model with the Measurement Calculus. Dataflow is a proven and popular model for parallel computing, which has been used both as a conceptual tool, supporting formal proofs, but also in practice, as an intermediate representation in optimizing compilers or even as execution technique in parallel processors. Our stepwise graph-based transformation of measurement patterns into a fine-grained dataflow graph has revealed similarities with the graph-based generalized flow algorithms. The foreseen benefit of such unification of Dataflow and MC is the exploration of a formal system that is still parallel by default, but which can deal with some of the limits and constraints introduced by physical parallel quantum computers.

References

[1] G. Steele, “Growing a language,” in *Higher-Order and Symbolic Computation*, 1999, pp. 221–236.

[2] V. Danos, E. Kashefi, and P. Panangaden, “The Measurement Calculus,” *Journal of the ACM (JACM*, vol. 54, no. 2, p. 8, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1219096

[3] R. Raussendorf and H. Briegel, “A one-way quantum computer,” *Physical Review Letters*, vol. 86, no. 22, pp. 5188–5191, 2001.

[4] R. Raussendorf, D. Browne, and H. Briegel, “Measurement-based quantum computation on cluster states,” *Physical Review A*, vol. 68, no. 2, p. 22312, 2003.

[5] A. Broadbent and E. Kashefi, “Parallelizing quantum circuits,” *Theoretical Computer Science*, vol. 410, no. 26, pp. 2489–2510, 2009.

[6] D. Browne, E. Kashefi, and S. Perdrix, “Computational Depth Complexity of Measurement-Based Quantum Computation,” *Theory of Quantum Computation*, vol. 6519, p. 35, 2011. [Online]. Available: http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2011LNCS.6519...35B&link_type=ABSTRACT

[7] S. Barz, E. Kashefi, A. Broadbent, J. F. Fitzsimons, A. Zeilinger, and P. Walther, “Demonstration of Blind Quantum Computing,” *Science*, vol. 335, no. 6066, pp. 303–308, Jan. 2012. [Online]. Available: http://www.sciencemag.org/cgi/doi/10.1126/science.1214707

[8] Y. Vandriessche, “*A foundation for quantum programming and its highly-parallel virtual execution*,” Ph.D. dissertation, Vrije Universiteit Brussel, Nov. 2012.

[9] D. Gottesman, “The Heisenberg Representation of Quantum Computers,” *Audio, Transactions of the IRE Professional Group on*, pp. –, Jun. 1998. [Online]. Available: http://pubget.com/paper/pgtmp_quant-ph9807006?institution=

[10] R. Karp and R. Miller, “Parallel program schemata: A mathematical model for parallel computation,” *IEEE Conference Record of the Eighth Annual Symposium on Switching and Automata Theory*, pp. 55–61, 1967.

[11] J. A. Sharp, *Data flow computing: theory and practice*. Norwood, NJ, USA: Ablex Publishing Corp., 1992.

[12] J. Dennis, “Data flow supercomputers,” *Computer*, vol. 13, no. 11, pp. 48–56, 1980.

[13] J. Gurd, C. Kirkham, and I. Watson, “The Manchester prototype dataflow computer,” *Communications of the ACM*, vol. 28, no. 1, pp. 34–52, 1985.

[14] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion, “Hybrid Dataflow/Von-Neumann Architectures,” *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–1. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6506076

[15] W. Johnston, J. Hanna, and R. Millar, “Advances in dataflow programming languages,” *ACM Computing Surveys (CSUR)*, vol. 36, no. 1, pp. 1–34, 2004.

[16] F. Allen and J. Cocke, “A program data flow analysis procedure,” *Communications of the ACM*, vol. 19, no. 3, p. 137, 1976.

[17] H. Henderson and S. Searle, “The vec-permutation matrix, the vec operator and Kronecker products: a review,” *Linear and Multilinear Algebra*, vol. 9, no. 4, pp. 271–288, 1981. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/03081088108817379

[18] E. D’Hondt and Y. Vandriessche, “Distributed quantum programming,” *Natural Computing*, vol. 10, no. 4, pp. 1313–1343, Dec. 2011.

[19] M. Pei and N. de Beaudrap, “An extremal result for geometries in the one-way measurement model,” *Quantum Information and Computation*, vol. 8, no. 5, pp. 0430–0437, 2008.

[20] D. Browne, E. Kashefi, M. Mhalla, and S. Perdrix, “Generalized flow and determinism in measurement-based quantum computation,” *New Journal of Physics*, vol. 9, p. 250, 2007.

[21] R. Duncan and S. Perdrix, “Rewriting measurement-based quantum computations with generalised flow,” *Automata, Languages and Programming*, pp. 285–296, 2010. [Online]. Available: http://www.springerlink.com/index/06g573327343gq75.pdf