ScaffCC: A Framework for Compilation and Analysis of Quantum Computing Programs

Ali JavadiAbhari, Shruti Patil,
Daniel Kudrow, Jeff Heckey, Alexey Lvov,
Frederic T. Chong, Margaret Martonosi

Princeton University, UC Santa Barbara, IBM

Quantum Computing Advantage

- A certain class of problems can be solved significantly faster by changing the paradigm of computing: use quantum mechanical systems to store and manipulate information.
- Example: Factoring a large *b*-bit number

	Asymptotic Complexity	232-digit number factoring
Best classical algorithm (GNFS) [Buhler 1994]	$O(\exp(\frac{64}{9}b)^{\frac{1}{3}}(\log b)^{\frac{2}{3}})$	2000 years on a single-core AMD Opteron [Kelinjung et al. 2010]
Shor's quantum algorithm	$O(b^3)$	*(technology dependent – theoretically large speedup)*

Background on Quantum Computers

- A quantum bit ***(qubit)*** can exist in a *superposition* of states:

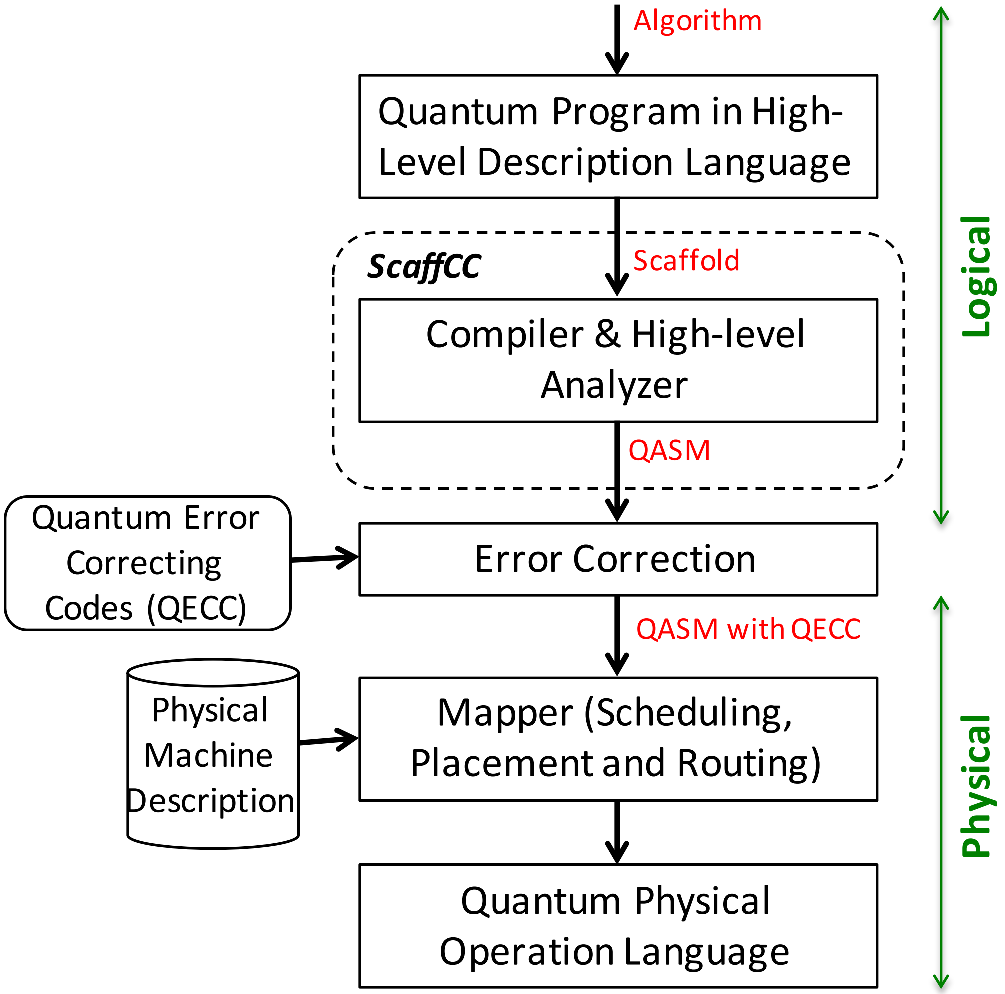
$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

- Quantum operations ***(gates)*** transform the state of qubits.
- **Measurement** (observation) collapses it to either $|0\rangle$ or $|1\rangle$.
- Quantum computation is reversible.

```
Quantum Assembly
qbit a[1], b[5];
H(b[0]);
H(b[1]);
H(b[2]);
H(b[3]);
H(b[4]);
Z(a[0]);
CNOT(a[0],b[1]);
```

Compiling Quantum Codes

- Data types and instructions in quantum computers:
  - Qubits, quantum gates
- Decoherence requires QECC
  - Logical vs. Physical Levels
- Efficiency crucial
  - Inefficiencies at logical level are amplified into greater physical level QECC requirements.



Goals and Contributions

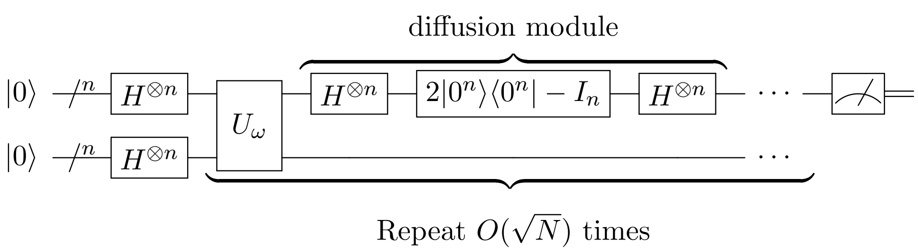
- 1) Identifying **differences in compiling** for quantum vs. classical computers
- 2) Providing **good scalability** to practical algorithm sizes
- 3) Automatically synthesizing **reversible computation** (e.g. for math functions)
- 4) Developing important program **analysis** passes

Benchmarks

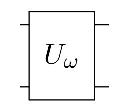
Benchmark	Classical Time Complexity	Quantum Time Complexity
Grover's Search	$O(n)$	$O(\sqrt{n})$
Binary Welded Tree (BWT)	$O(\frac{1}{4}2^{n/3})$	$O(\frac{1}{4}k^4n^9)$
Ground State Estimation (GSE)	$O(2^n)$	$O(n^5)$
Triangle Finding Problem (TFP)	$O(n^2)$	$O(n^{1.3})$
Boolean Formula (BF)	$O(n)$	$O(\sqrt{n})$
Class Number (CN)	$O((n\ln n)^{0.5})$	$O(\log(n)\log^*(n))$

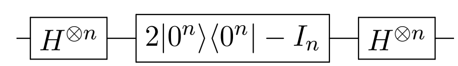
Scaffold Programs and Quantum Circuits

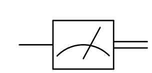
```
#include <math.h>
#define n 5
#define N pow(2,n)

// module prototypes
module Sqr(qbit a[n], qbit b[n]);
module EQxMark(qbit b[n], qbit t[1], int tF);

// diffusion module
module diffuse(qbit q[n]) {
    // allocate qubits local to module
    qbit x[n-1];

    // Hadamard applied to q
    for(j = 0; j < n; j++)
     H(q[j]);
    ...
}
```




```
// main module
module main() {
    // allocated qubits in main
    qbit a[n], b[n], t[1];
    // classical bits : measurement outcome
    cbit ma[n];
    // iteration bound
    int nstep = floor((pi/4)*sqrt(N))
    .
    .
    // Grover iteration: Repeat O(N^0.5) times
    for (istep=1; istep<=nstep; istep++) {
     Sqr(a,b);
     EQxMark(b, t, 0);
     Sqr(a,b);
     diffuse(a);
    }
    .
    .
    // measure a to find outcome
    for(i=0; i<n; i++)
     ma[i] = measZ(a[i]);
}
```

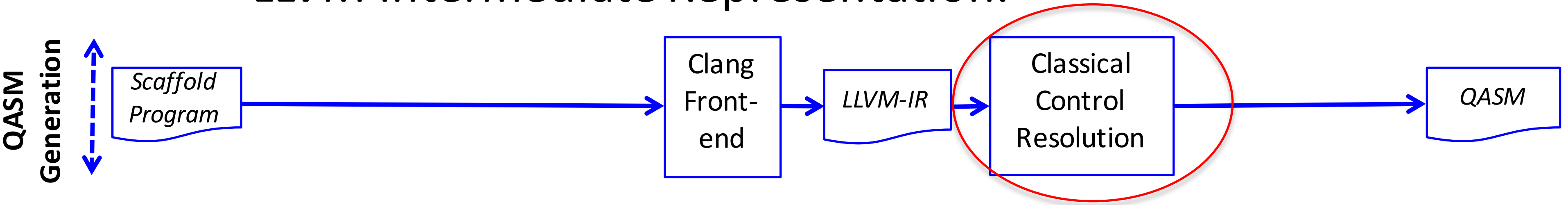






From Scaffold to QASM: Deep Optimization through LLVM

- ScaffCC translates from **Scaffold** Programming Language to **QASM** assembly language.
  - Implemented with **LLVM**, a rich and mature compiler framework.
  - Modified **Clang** front-end parses and converts ScaffCC to LLVM Intermediate Representation.



Scalability in Compilation and Analysis (1)

- Quantum circuits are typically specialized to one problem size, hence they are deeply and statically analyzable.
  - **Classical control resolution**
- Static classical control resolution using LLVM passes
  - May cause code explosion during code transformation of larger problems

Resolving Classical Controls in the Code

- Classical control surrounding quantum code must be resolved to disambiguate for the hardware the qubits and the exact set of gates

```
#define s_ 3000
module Oracle(qbit a[1], int j) {

    double theta=(-1)*pow(2,j)/100;
    Rz(a[0],theta)
}
module main() {
    qbit a[1];
    int i,j;
    for (i=0;i<=s_;i++)
     for (j=0;j<=3;j++)
       Oracle(a,j);
}
```

```
module Oracle_0(qbit a[1]) {
    Rz(a[0],-0.01);
}
```

.

.

```
module Oracle_3(qbit a[1]) {
    Rz(a[0],-0.08);
}
```

Classical Control Resolution

```
module EQxMark (qbit b[n], qbit t[1], int tF)
{
   .
   .
   if(tF==1)
      CNOT(t[0], x[n-2]);
   else
      Z(x[n-2]);
   .
   .
}
```

clone

```
module EQxMark_0 (qbit b[n], qbit t[1])
{
   .
   .
   Z(x[n-2]);
   .
   .
}
```

```
module EQxMark_1 (qbit b[n], qbit t[1])
{
   .
   .
   CNOT(t[0], x[n-2]);
   .
   .
}
```

inter-procedural
constant
propagation

```
module main (qbit b[n], qbit t[1])
{
   .
   for (i=0; i<2; i++)
      EQxMark(b,t,i);
   .
}
```

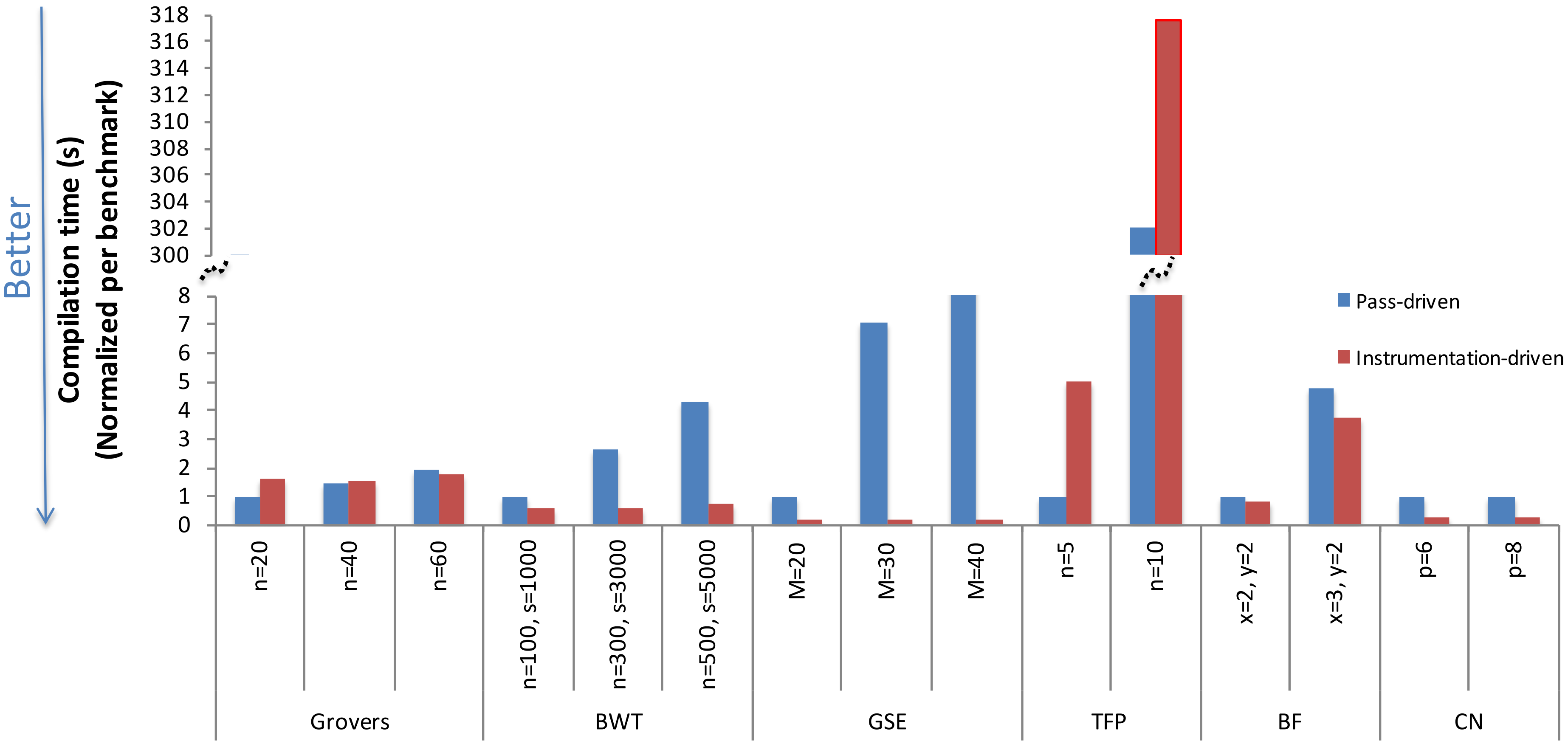
unroll

module main (qbit b[n], qbit t[1])
{
   .
   ~~EQxMark(b,t,0);~~ EQxMark_0(b,t);
   ~~EQxMark(b,t,1);~~ EQxMark_1(b,t);
   .
}

Pass-Driven Vs. Instrumentation-Driven

- **Pass-Driven:**
  - Loop unrolling
  - Procedure Cloning
  - Inter-procedural Constant Propagation
- **Instrumentation-Driven:**
  - Leveraging the dual nature of quantum programs
  - Instrument code such that a fast classical processor executes through the classical portion, collecting information regarding the quantum portion
  - Further speed-up by memoizing same module calls

The Instrumentation-Driven Approach Scales Better



Scalability in Compilation and Analysis (2)

- Traditional QASM:
  - No loops or modules: only sequences of qubits and gates
  - Used for small program representations
- Programs that we examined contained between 10^7 to 10^{12} gates
- We need a more scalable output format:
  - QASM with Hierarchy (QASM-H)
    - 200,000X smaller code
  - QASM with Hierarchy and Loops (QASM-HL)

Managing Scalability with QASM Format

Scaffold

```
#define n 1000
module foo(qbit q[n]) {
    for(int i = 0; i < n; i++)
     H(q[i]);
    CNOT(q[n-1],q[0]);
}
module main() {
    qbit b[n];
    foo(b);
}
```

QASM-H

```
module foo(qbit* q) {
    H(q[0]);
    H(q[1]);
    ..
    H(q[999]);
    CNOT(q[999],q[0]);
}
module main() {
    qbit b[1000];
    foo(b);
}
```

Flat QASM

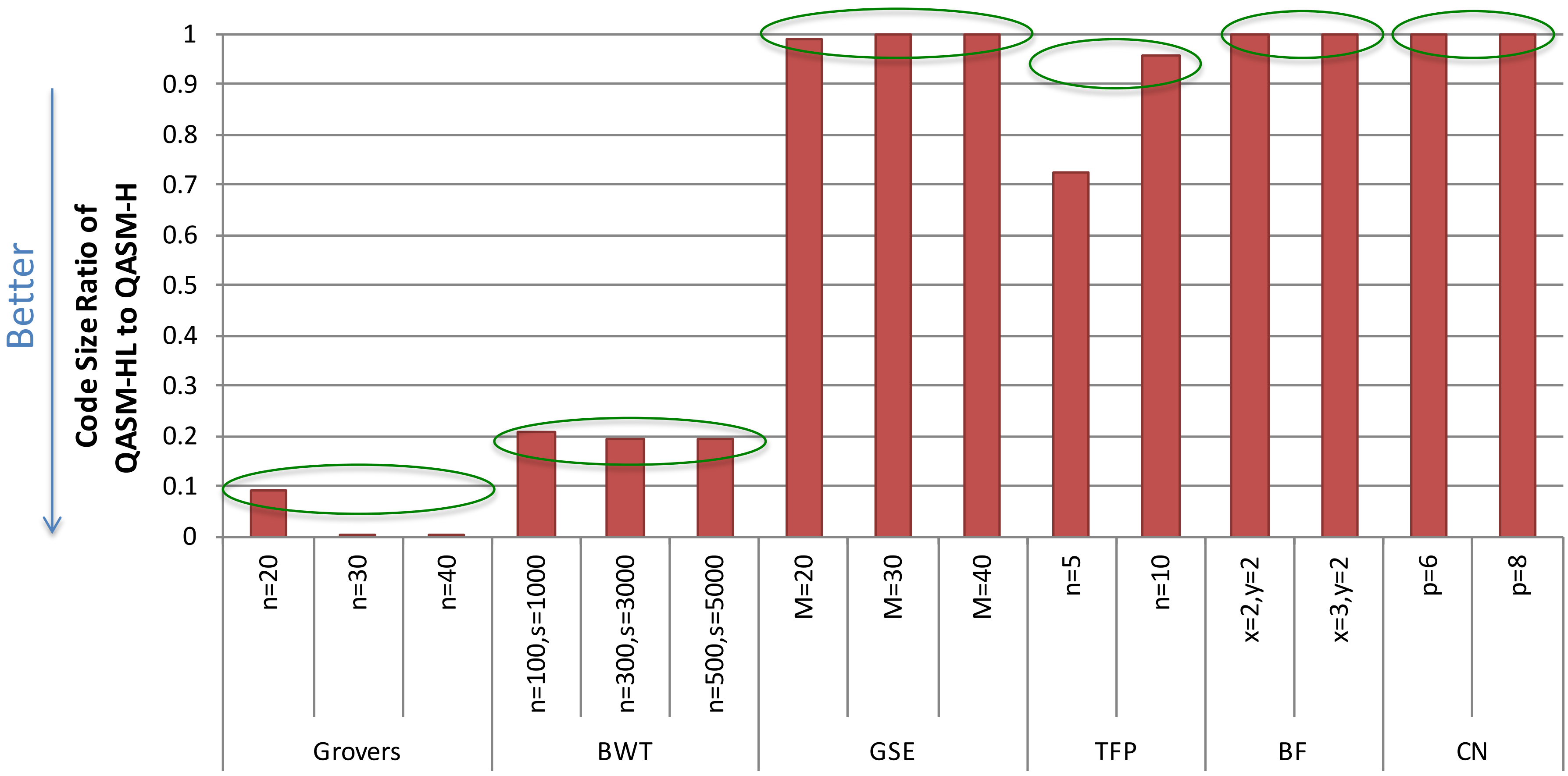
```
qbit b[1000];
H(b[0]);
H(b[1]);
.
.
H(b[999]);
CNOT(b[999],b[0]);
```

QASM-HL

```
module foo(qbit* q) {
    H(q[0:999]);
    CNOT(q[999],q[0]);
}
module main() {
    qbit b[1000];
    foo(b);
}
```

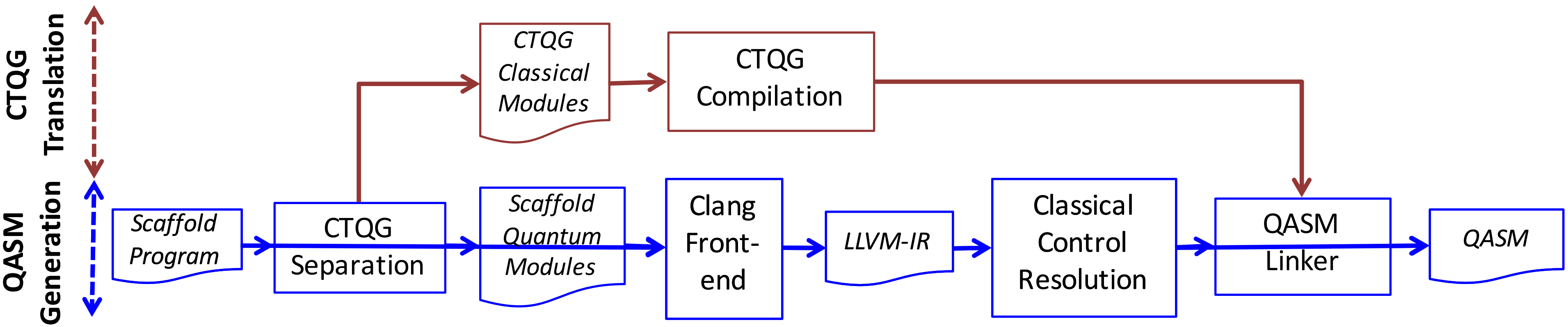
Comparison of QASM-H and QASM-HL

- A large reduction is already obtained from QASM-H over flat QASM.



Synthesizing Reversible Computation

- Classical-To-Quantum-Gate (**CTQG**): A ScaffCC feature for efficiently translating classical modules to quantum modules.

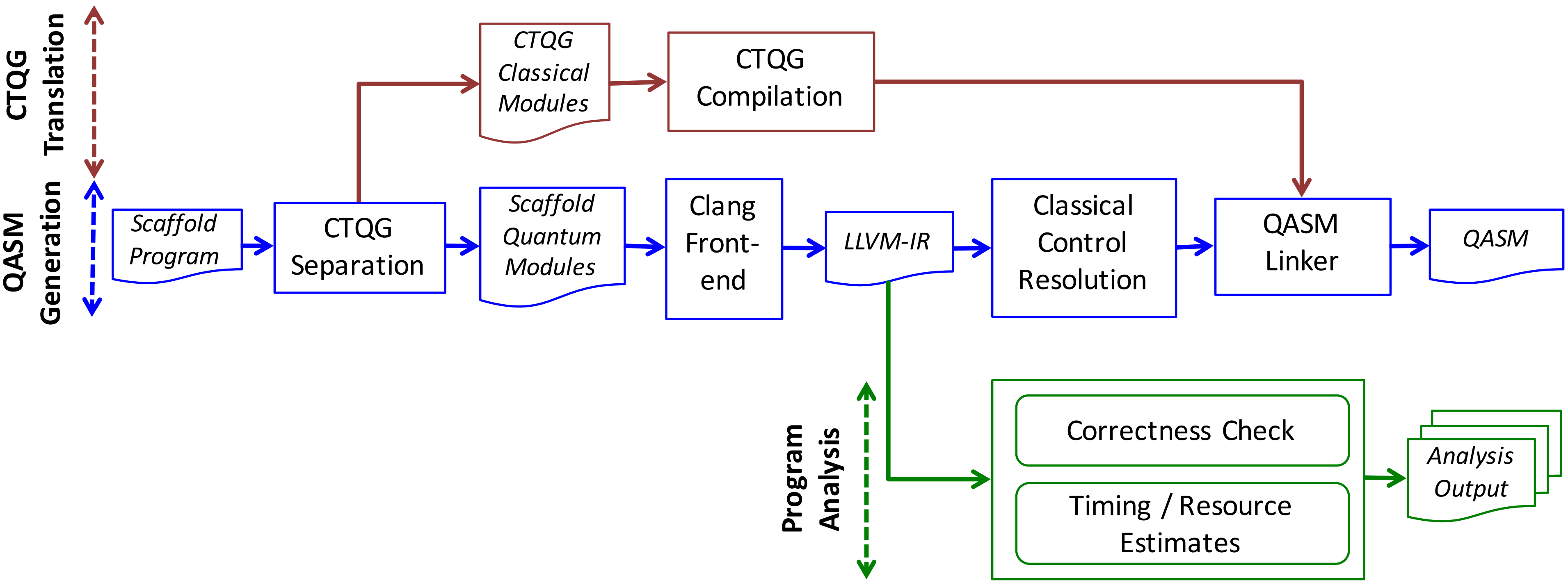


CTQG: Classical-To-Quantum-Gate

- Facilitates the synthesis of quantum circuits from classical mathematical expressions:
  - Basic **integer** arithmetic (a=a+b, a=a+bc, ...)
  - **Fixed-point** arithmetic (1/x, sin x, ...)
  - **Bit-wise** manipulations (shift operators, ...)
- State-of-the-art in reversible logic synthesis, minimizing the use of extra (*ancilla*) qubits
- Produces output gate-by-gate on the fly
  - Not limited by memory

Program Analysis

- Analysis passes:
  - Program correctness checks
  - Program estimates



Program Analysis

- ScaffCC supports a range of code analysis techniques:
  - Program correctness checks:
    - No-cloning checks
    - Entanglement and un-computation checks
  - Program estimates:
    - Resource estimation
    - Timing analysis (Parallel scheduling)

Program Correctness Checks

- No-Cloning:
  - Theorem: The state of one qubit cannot be copied into another (no fan-out)
  - Check that multi-qubit gates do not share qubits
- Entanglement:
  - The joint state of two qubits cannot be separated
  - Data-flow analyses to automate the tracking of entanglement and disentanglement

Quantum Program Analysis: Resource Analysis

- Obtaining estimates for the size of the circuit:
  - Qubits are expensive
  - More gates require more overall error correction and hence more cost
- The same pass-driven and instrumentation-driven approaches apply
- Dynamic memoization table records number of resources

Module	IntegerParam	DoubleParam	Resources				
			Qubit	X	Z	H	T
main	0	0	2	400	27800	54300	55100
Oracle	0	0	0	1	76	137	140
Oracle	1	0	0	1	65	130	132
Oracle	2	0	0	1	64	142	142
Oracle	3	0	0	1	73	134	137

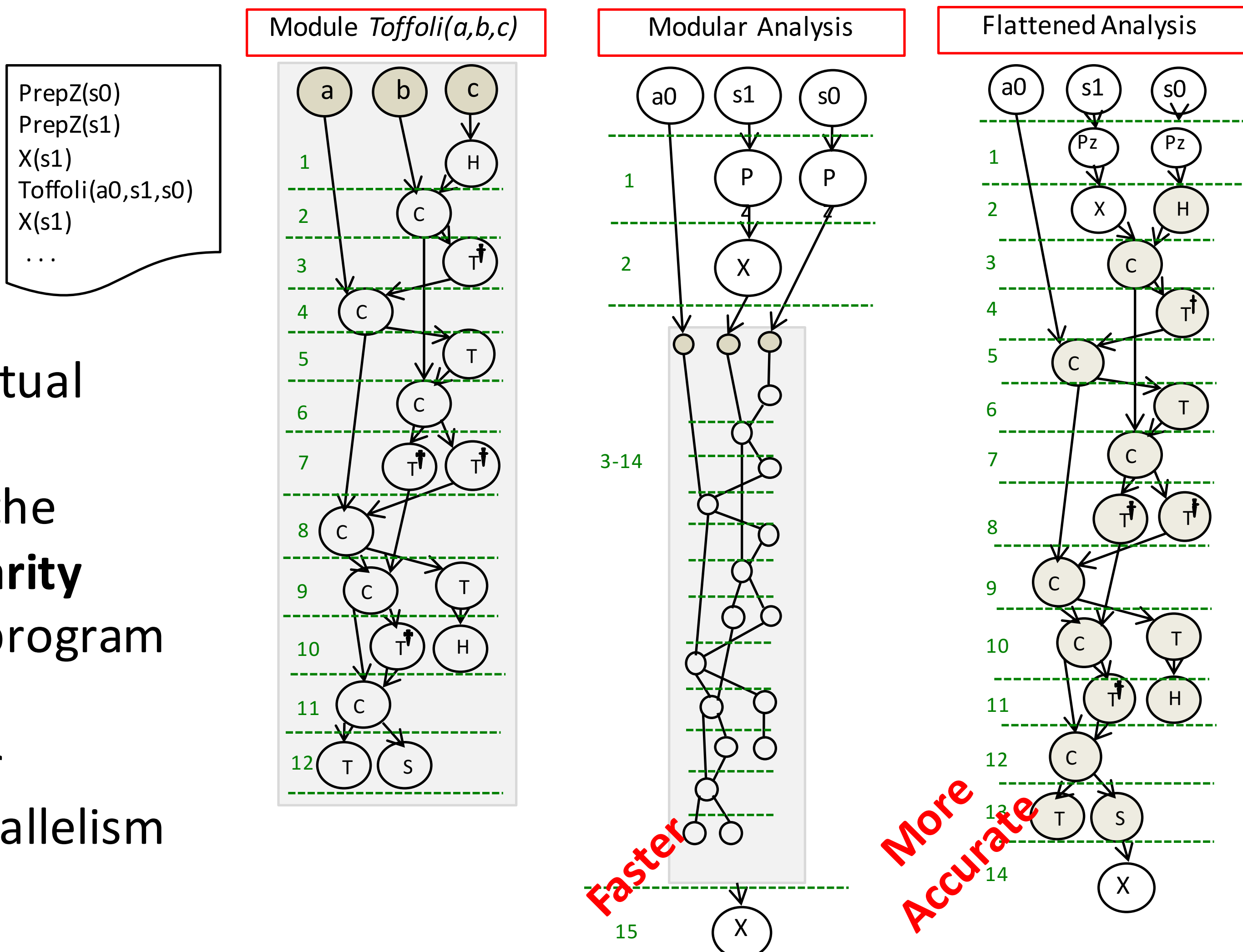
Timing Estimate

- Estimates the critical path length of the program
  - Assuming unlimited hardware capability for parallelization
- Scheduling based on **qubit data dependencies** between **operations**
- Hierarchical scheduling for tractability:
  - Obtain module critical paths separately and then treat them as black boxes.

Remodularization

- Analysis makes use of modularity
  - Avoid repetitive analysis
  - Reduce analysis time
- Results in loss of parallelism at module boundaries
  - Decreased schedule optimality
- Idea:
  - Inline small modules at call sites – larger flattened modules
  - Define threshold for “small” modules
  - Results in better critical path estimates

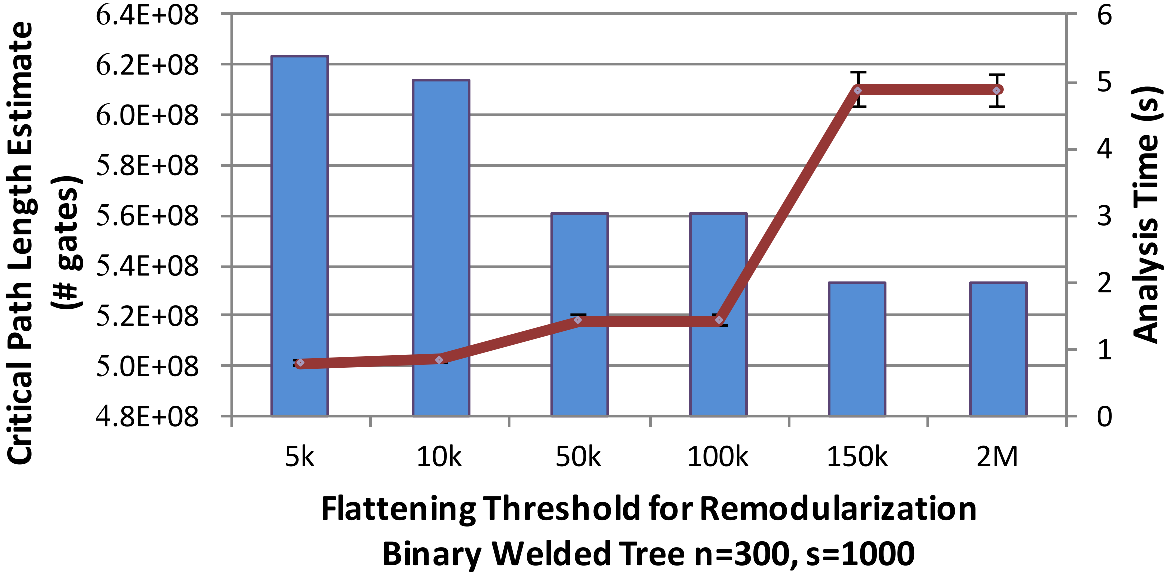
Hierarchical Approach Tradeoff



- Closeness to actual critical path is dependent on the level of **modularity**
- Flatter overall program means more opportunity for discovering parallelism

Effect of Remodularization

- Based on resource analysis, flatten modules with size less than a threshold
- Tradeoff between speed of analysis and its accuracy



Demo

Conclusion

- Extended LLVM's classical framework for quantum compilation at the logical level
- Managed scalability through:
  - Output format:
    - 200,000X on average + up to 90% for some benchmarks
  - Code generation approach:
    - Up to %70 for large problems
- CTQG: Automatic generation of efficient quantum programs from classical descriptions
- Developed a scalable program analysis toolbox
- ScaffCC can be used as a future research tool

Thank You