Fridge Compiler: Optimal Circuits From Molecular Inventories *

Lancelot Wathieu, Gus Smith, Luis Ceze, and Chris Thachuk

Paul G. Allen School of Computer Science & Engineering,
University of Washington, USA
{lwathieu,thachuk}@cs.washington.edu

Abstract. Rationally designed molecular circuits describable by well-mixed chemical reaction kinetics can realize arbitrary Boolean function computation yet differ significantly from their electronic counterparts. The design, preparation, and purification of new molecular components poses significant barriers. Consequently, it is desirable to synthesize circuits from an existing "fridge" inventory of distinguishable parts, while satisfying constraints such as component compatibility. Heuristic synthesis techniques intended for large electronic circuits often result in non-optimal molecular circuits, invalid circuits that violate domain-specific constraints, or circuits that cannot be built with the current inventory. Existing "exact" synthesis techniques are able to find minimal feedforward Boolean circuits with complex constraints, but do not map to distinguishable inventory components.

We present the Fridge Compiler, an SMT-based approach to find optimal Boolean circuits within a given molecular inventory. Empirical results demonstrate the Fridge Compiler's versatility in synthesizing arbitrary Boolean functions using three different molecular architectures, while satisfying user-specified constraints. We showcase the successful synthesis of all 256 three-bit and 65,536 four-bit predicate functions using a large custom inventory, with worst-case completion times of only seconds on a modern laptop. In addition, we introduce a unique class of cyclic molecular circuits that cover a larger number of Boolean functions than their conventional counterparts over a common inventory, often with significantly smaller implementations. Importantly, and absent in previous approaches specific to molecular circuits, the Fridge Compiler is logically sound, complete, and optimal for the user-specified cost function and component inventory.

Keywords: Molecular Computing · Exact Logic Synthesis · Cyclic Combinational Circuits.

1 Introduction

Molecular circuits are being explored and developed due to their natural interface with chemistry and biochemical systems. They can sense from their environment,

* Supported by an NSF grant (CCF 2106695) and a Faculty Early Career Development Award from NSF (CCF 2143227).

perform computation, and actuate a physical response *in situ*. Diffusive molecular circuits capable of arbitrary Boolean function computation—as described by rate-independent, well-mixed chemical reaction networks (CRNs) and our focus here—can be experimentally realized by DNA strand displacement (DSD) architectures.

DSD is a molecular primitive that can realize programmed behavior by the rational design of short DNA oligonucleotide strands. Due to designed sequence similarity 'invader' strands can compete to form Watson-Crick base-pairs with a substrate strand and displace an 'incumbent' strand that was initially hybridized. These reactions are often mediated and thermodynamically driven by the enthalpic gain of additional base pairs formed by an invader in a 'toehold' region of the substrate. Displaced strands can in turn act as invaders in downstream components, creating a network of cascading displacements in the presence of appropriate inputs strands [21]. DSD architectures, also referred to as DSD systems, use DSD to implement modular computing components often designed to robustly implement CRNs [21,4,2]. Figure 1 illustrates a detailed DSD pathway for the seminal two-domain architecture [4], and Section 2.2 outlines our approach for representing DSD architectures as Boolean Networks.

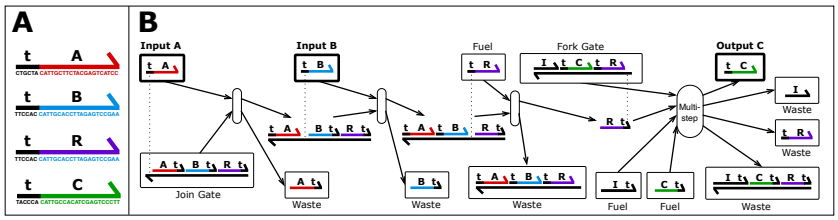

Fig. 1. Example strand displacement implementation of $A + B \to C$ using the two-domain architecture [4] – interpreted in this paper as $C := \texttt{AND}(A, B)$. DNA is represented with arrows indicating the 3' end. Each step exposes a 'toehold' region labeled `t` in the 'incumbent' strand where the 'invader' initiates binding along the dotted line. **A.** Sequence domains labeled `A`, `B`, `R`, `I`, and `C` serve as distinct signals in our well-mixed solution. Sequences and design insights are derived from [7]. **B.** The DSD pathway shows that output C is generated only if both inputs A and B are present. This process unfolds through a cascade of toehold-mediated reactions that deplete input, gate, and fuel components while generating output and waste components. Given that each circuit preparation can only process one input due to component consumption, we refer to this as a "one-shot" computation.

Given a target function and architecture a molecular programmer must first synthesize a compatible molecular parts list that will correctly compute the intended logic when well-mixed and in the presence of valid input. Manual synthesis is typical; however, automated synthesis becomes necessary to determine minimal size molecular circuits for all but the simplest of functions. Existing approaches have leveraged sophisticated electronic circuit synthesis tools by first

synthesizing optimal Boolean circuits over a functionally complete basis (*e.g.* using `AND`, `OR` and `NOT` gates) and then performing technology mapping into the target molecular architecture, as demonstrated in Figure 2. This strategy falls short for at least four reasons:

1. Traditional circuit synthesis tools are based on heuristic logic synthesis and ignore paradigms like cyclic combinational circuits that can be smaller than their acyclic counterparts.
2. Minimal circuits prior to technology mapping do not necessarily result in minimal molecular circuits; synthesis tools intended for electronic circuit optimization do not consider the unique properties of rate-independent DSD circuits such as it being infeasible to compute negation unless using a dual-rail input encoding, or the relative cost and/or limitation of fan-in and fan-out for particular gate architectures.
3. The molecular programmer must still determine if there is mapping from their existing inventory of molecular parts onto the synthesized circuit. If not, the laborious process of designing and preparing new molecular components would be necessary to realize the synthesized circuit. However, it is entirely possible that an *alternate* circuit could have been realized from an inventory of existing components.
4. Molecular circuits are often encumbered by additional constraints learned through a series of experiments (*e.g.*, spurious interactions between certain components). As these events are learned they must be considered as constraints by any synthesis tool.

This work advocates for an exact synthesis approach: given a Boolean function $\mathcal{F}$ and an existing "fridge of parts" inventory $\mathcal{I}$ choose a set of molecular components $\mathcal{I}' \subseteq \mathcal{I}$ that yields a valid and optimal implementation of $\mathcal{F}$ when mixed together, or give a proof that no such $\mathcal{I}'$ exists. This approach not only leads to a practical tool capable of handling arbitrary constraints and cost functions, it also avoids whenever possible the most costly solutions: those which require the design and/or preparation of molecular components not currently found in the inventory.

Our major contributions in this work are organized as follows. Section 2 defines the molecular circuit synthesis from an inventory problem. Section 3 details the Fridge Compiler tool and an overview of its implementation. Section 4 introduces a new class of cyclic circuits for "one-shot" molecular computation typical in DSD architectures, implementing Boolean functions with fewer or the same number of components than traditional (cyclic or acyclic) circuits. Section 5 provides a number of case studies that demonstrate the compiler is flexible and highly performant. Section 6 explores future directions necessary to meet the current needs of practitioners of molecular programming.

1.1 Relation to Previous Work

Molecular Circuit Synthesis. Tools and compilers exist that support the experimental implementation of arbitrary or specific DNA strand displacement

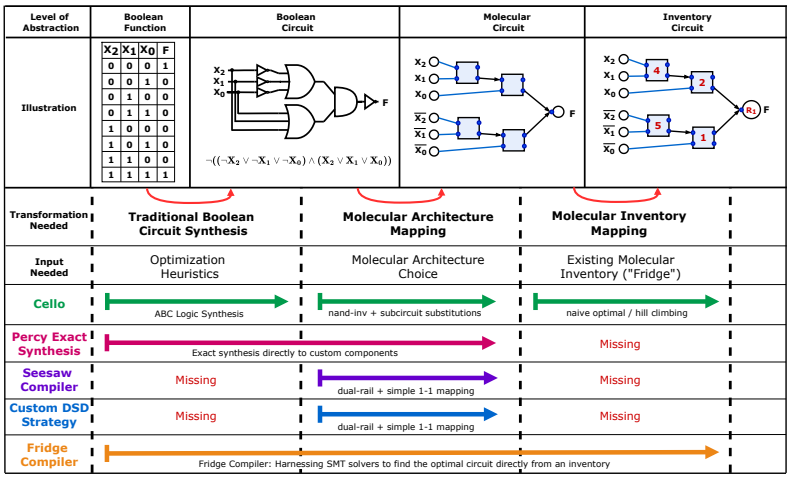

Fig. 2. Overview of circuit synthesis of Boolean functions, contrasting different approaches including Cello for genetic circuits, Percy exact synthesis, Seesaw Compiler, a custom strategy, and the Fridge Compiler. Given an inventory of distinguishable molecular parts and a function specification, other synthesis strategies shown take multiple synthesis steps to find a solution. During these steps optimality, completeness, and/or flexibility may be lost.

architectures, but they typically assume a target chemical reaction network or circuit [2,12,15]. As researchers develop larger and more advanced DSD circuit inventories, an unfulfilled need has arisen for the ability to use this existing material efficiently in a scalable manner while taking into account common DSD constraints. Our focus is automated synthesis of molecular circuits given a Boolean function description and an inventory of well-tested molecular parts.

The `Cello` tool [14,11] workflow is most comparable to our aim in this work, as it seeks to implement a Boolean function using a set of existing genetic circuit components. Genetic circuits use genetic regulatory networks to perform computation and therefore rely on reaction rate differences and repression to implement negation (*e.g.,* to realize NOR logic). In contrast, DSD architectures use rationally designed DNA strands to perform computation and cannot robustly implement negation, often relying on a dual-rail representation for functional completeness.

Both types of circuits share the common challenge of designing and testing limited-resource components. Cello utilizes a heuristic logic synthesis step to create an optimized intermediate circuit, making the technology mapping phase trivial as all gates in the intermediate network can be implemented. The heuristic fridge mapping phase in Cello is also more complex, using analog gate response characterization instead of simple digital abstraction. Although Cello's heuristic logic synthesis offers advantages in genetic circuit design, it cannot implement

optimal dual-rail circuits for DSD and lacks the completeness, and constraint-flexibility guarantees of the Fridge Compiler. Additionally, Cello does not support combinational cycles.

The loss of optimality and difficulty of satisfying arbitrary constraints make it challenging to use mainstream compilers for molecular computing, where small circuits with constrained architectures are the norm.

Logic Synthesis. Logic synthesis tools take high-level hardware language like Verilog, and map a design into technologies such as Application-Specific Integrated Circuits (ASICs) or Field-Programmable Gate Arrays (FPGAs). *Heuristic synthesis* approaches—including commercial tools such as Intel Quartus or Xilinix Vivado and open source alternatives [3,25,19]—prioritize large-circuit scaling by sacrificing flexibility and optimality. They can be modified in order to meet some constraints, such as pushing inversions out of the main compute path to inputs [22], although it is not always feasible to implement arbitrary constraints. *Exact synthesis* approaches prioritize flexibility and optimality with the help of constraint solvers [18]. Tools like Percy [9] combine logic synthesis and technology mapping into one optimal step. However, an optimal technology mapped circuit may not be realizable from a particular molecular inventory. Furthermore, both heuristic and exact logic synthesis tools often overlook paradigms such as cyclic combinational circuits, which are required for achieving minimal circuit sizes [16].

Program Synthesis. Program synthesis generates an implementation from its specifications and constraints, often employing SMT solvers such as Z3 [13] to define and search the space of valid solutions. *Satisfiability modulo theory* (SMT) problems, a superset of SAT problems, accommodate variables and constraints from domains (or *theories*) beyond Boolean algrebra, such as bitvectors and uninterpreted functions. SMT solvers have been used to synthesize and verify code in various domains, including DNA computation and synthetic biology for analyzing and verifying biological systems [26]. The synthesized circuits from these tools are also often acyclic, resulting in suboptimal combinational circuits (see Section 4).

2 Preliminaries

2.1 Molecular Circuit Synthesis from an Inventory

Molecular circuit synthesis from an inventory aims to find an optimal molecular circuit given a target function, user-defined constraints, and a fridge inventory of distinguishable, freely-diffusing components. Components must be *distinguishable* in diffusive computing, such as those implementations describable as a well-mixed CRN, since relationships between components must be programmed into their designed interactions (and lack of interactions) with other species components due to the lack of spatial organization. In contrast, two `AND` gates, for example, can be *indistinguishable* in an electronic circuit embedded on a surface since their connectivity to other components is entirely controlled by their spatial organization.

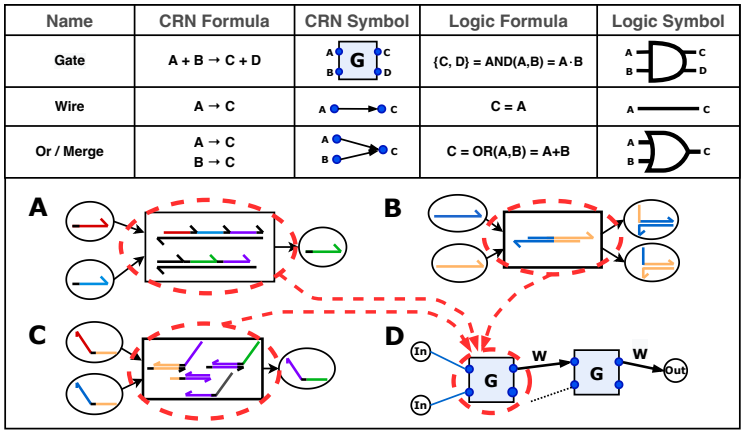

Fig. 3. (Top) Default circuit and inventory representation in this paper: CRNs with components as either 2-input 2-output CRNs (gates) or 1-input 1-output CRNs (wires). OR logic is commonly implemented by multiple wires converging to a common output. (Bottom) Fridge Compiler users can choose among a variety of pre-programmed DSD architectures including (A) two-domain [4], (B) leakless breadboard [23], (C) seesaw [15]. Alternatively, the user can create their own. The Fridge Compiler is capable of supporting a variety of architectures by abstracting the specific implementation into a generic circuit structure (D) while still retaining the important differences in the form of constraints.

2.2 Modelling Molecular Inventories

Rate-independent well-mixed CRNs, which compute correctly without reaction rate assumptions, are an appealing class of reactions due to their relative ease of engineering and ability to perform complex functions [5]. To represent arbitrary Boolean functions using rate-independent CRNs, we utilize a dual-rail input representation with distinct species for positive and negative literals, and consider outputs on upon reachability of output species – thereby treating the CRN representation like a Boolean network [5].

While our primary focus is on DSD-based CRNs our approach could be adapted to other molecular computing frameworks, including protein and genetic circuits, given appropriate consideration for managing complex constraints, inversions/negations, and cycles. The Fridge Compiler, detailed in the next section, is compatible with a variety of DSD architectures and abstracts specific implementation details to the rate-independent CRN level, while maintaining important differences as constraints to the SMT solver.

Figure 3 (top) shows the convention used for different nodes within this Boolean network interpretation of CRNs. A signal represents a single molecule; it is represented by dark blue circle gate ports. A gate represents a reaction/component with (multiple) inputs and (multiple) outputs. Gates in this interpretation perform `AND` logic, like a transition in a Petri net. In many architectures `OR` logic can be performed by mapping multiple distinct signals into a common signal. A wire

represents a translator from one signal to another. Figure 3 (bottom) shows the domain-level design of gate nodes in three different DSD architectures. Signals can be tagged as possible circuit outputs or inputs that correspond to fluorescent reporters on hand, or available DNA strands, respectively.

Since diffusive molecular circuits cannot rely on spatial organization, each component's output feeds into the inputs of other predetermined components. Therefore, all possible connections in the fridge are predetermined and an inventory can itself be represented by a rate-independent CRN. A particular circuit is a subset of this inventory. Only the components which form that circuit are enabled, and the corresponding induced subgraph of chemical reactions represents that circuit. Furthermore, the input nodes will be tagged with the function's input variables, or with `True` or with `False`. The output nodes will be tagged so that the correct function outputs appear at the tagged output nodes.

2.3 Desired Features of a (DSD) Fridge Compiler

Along with the distinguishability of components, an arduous development cycle is one of the major reasons for the need and development of an efficient "fridge compiler" with the following properties. *Soundness and Completeness.* An input query consists of a fridge and a Boolean function. Soundness guarantees that if the compiler returns a circuit, it will always correctly implement the input query. Completeness guarantees that if the compiler does not return a circuit, then there does not exist any circuit that fulfills the input query. Most alternative methods are sound, but not complete. *Flexibility of Architecture Choice.* A fridge compiler should be able to support any DSD architecture, and in the future any molecular architecture. *Handling practical constraints.* A fridge compiler should be able to handle arbitrary constraints such as the known incompatibility between stock components. *Optimality.* Optimality is important in molecular computing because molecular components are expensive in their preparation and the probability of spurious interactions increase with system size. Arbitrary cost functions should be possible, since each architecture has different comparative cost for each component. *Supporting Cyclic Circuits.* As shown in section 4, optimal circuits will often contain cycles.

2.4 A Naive Solution

Figure 2 shows the levels of abstraction that traditionally take place to transform a Boolean function description and an inventory into a final circuit. A naive solution to molecular circuit synthesis from an inventory would be to enumerate all possible circuits within that inventory and evaluate whether any implement the desired function. Each enumeration choice requires picking which input node to use for which literal, picking a subset of components to enable, and picking which output node(s) to interpret as output. Checking if the intended function was synthesized requires evaluating all possible input combinations in the worst case. This naive enumeration algorithm has the properties we seek, but is infeasible. The Fridge Compiler keeps these properties, but is efficient in practice.

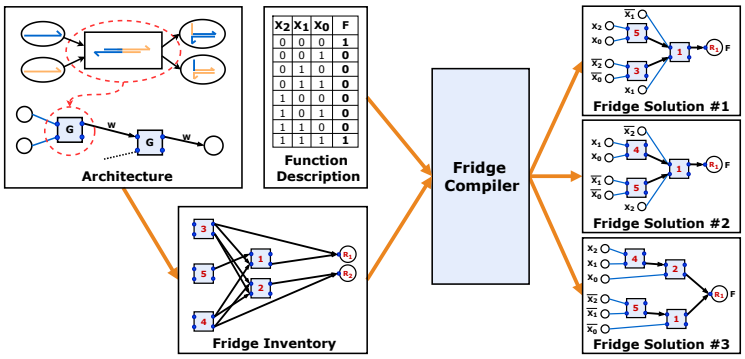

Fig. 4. Given a inventory and function specification, the Fridge Compiler can synthesize all valid circuit solutions (#1, #2, #3, ...), (size) optimal solutions (#1, #2), and optimal solutions that satisfy hard constraints (#1; "don't mix gates 4 and 5, nor gates 1 and 2"). Optimality, completeness, and flexibility are maintained.

3 The "Fridge" Compiler

3.1 General Overview

The Fridge Compiler is an exact synthesis tool similar to those in Section 1.1, providing an exact and optimal solution to a circuit description, over a given fridge inventory, and subject to constraints. It uses an SMT solver to describe the large set of enumerations using symbolic datatypes which replace the naive concrete enumerations. This approach fulfills all the desired features enumerated in Section 2.3: it is sound, complete, flexible, optimal, supports cycles, and is tractable in practice.

While many exact synthesis tools rely on custom-encoded SAT encoding [9], we build on program synthesis techniques that utilize more general SMT encodings because of their expressive power and ease-of-use [10]. We see this user-first approach as especially valuable in fields like molecular computing where the barrier to entry is already high. Our implementation borrows techniques from program synthesis and formal methods such as program sketches, symbolic execution, and SMT-based variable encoding [20,24,10]. We utilize a "sketching" approach where users with domain expertise write partial programs and leave "holes" for a solver to complete. The following section describes the Fridge Compiler's inputs and implementation.

3.2 Inputs

1. A inventory specified as: (a) A chemical reaction network without rates. (b) The subset of reactants that can be used as circuit inputs. (c) The subset of products that can be used as circuit outputs.

2. A desired output function, $F_n(X) = [F_0(X), F_1(X), ...]$, composed from one or more Boolean predicates.
3. A set of assumptions about the circuit including: (a) Architecture-specific constraints. (b) User-specified constraints.
4. A cost function.

It's worth drawing parallels to formal methods, where our input types above create a domain-tailored abstraction similar to a Domain-Specific Language (DSL). The inventory (1) forms our circuit (program) sketch, where symbolic variables specified below mark holes to be filled by the Fridge Compiler – effectively sculpting the final circuit. The desired function (2) specifies circuit behavior, similar to DSL specifications. Constraints (3), akin to DSL's semantic rules, limit the search space and ensure the circuit meets conditions. The cost function (4) ranks solutions to guide circuit selection.

3.3 Implementation

SMT Variable Encoding. Our approach to variable encoding in SMT is inspired by previous works [10]. *Logical input* variables, denoted as $X = [X_0, X_1, ...]$, serve as the Boolean inputs for the specification function (2). *Circuit selection* variables are Boolean values that dictate whether each component or reaction in the inventory is included in the circuit. Given that all components are distinguishable with pre-determined connections, selection variables define a complete circuit without I/O assignments. We also employ I/O location variables, which use integer values to map the input and output to the circuit [10]. *Input location* variables map our allowed circuit inputs (1b) above to a dual-rail version of our logical input variables. *Output location* variables map our function outputs (2) to our allowed circuit outputs (1b).

While a universal quantifier is performed over the *logical input* variables to ensure the equivalence of the circuit and function across all inputs, the remaining variables act as hole variables in the circuit sketch of the inventory. The solver fills these holes by assigning suitable values, defining a complete circuit including its I/O assignments.

Symbolic Interpreter for the Inventory. Interpreting a molecular circuit begins by assigning Boolean values to circuit inputs and triggering reactions until a steady state is reached, as exemplified in Figure 5.

To encode this into the solver, we apply symbolic execution [24] that transforms our concrete interpreter into a symbolic expression encapsulating all potential circuits. The interpreter starts from circuit inputs, integrates the SMT variables, and continuously simplifies the expression using Z3's *simplify* command.

In acyclic circuits, the interpreter accumulates logic from input to output, resulting in an expression proportional to the number of reactions in the CRN (1a). For cyclic circuits, which take more steps to reach the steady state depending on the number of cycles, we employ SMT verification to check the expression's equivalence after each gate, compressing the final expression – this step occurs only once per inventory and is typically not costly. Note that even for cyclic

circuits a steady state is always reached because once an output is produced it cannot be removed, by the monotonicity property of rate-independent CRNs [5].

This procedure yields SMT expressions for circuit outputs, depending on logical inputs X, circuit selection, and I/O location, designated as $CircuitF_n(X) = [CircuitF_0(X), CircuitF_1(X), ...]$.

Additional Constraints. Additional constraints further guide the solver, focusing the search and limiting the circuit space. These constraints are translated into SMT format and include architecture-specific constraints and user-defined requirements. Examples of architecture-specific constraints include uniqueness of location variables or forcing all outputs to be used by at most one enabled component. Examples of user-specified constraints include component incompatibility, restrictions on fan-in and fan-out, limitations on circuit depth, concentration splitting, or stock depletion. This flexibility in constraint definition allows for more customized and application-specific circuit designs.

Program Synthesis. The program synthesis task is to find an assignment of hole variables – *circuit selection, input location, output location* such that, assuming the *constraints* hold true, the function $F_n(X)$ matches $CircuitFn(X)$ for all possible X – **assuming** *constraints*, **forall** X: **assert** $F_n(X) == CircuitFn(X)$.

Existence vs Optimality. The system functions in two modes: existence mode, which finds the first circuit satisfying the query, and optimize mode, which seeks the most optimal circuit. Both modes are sound and complete. The cost function can be any function that can be evaluated for a specific circuit, allowing for extensive flexibility. For example, if a gate costs twice as much as a wire the cost function could be 2*W+1*G. However, performance is typically best with simple lexicographical cost functions rather than pareto tradeoff cost functions.

Implementation Details. For handling the universal (**forall**) quantifier in our implementation, we provide two options. By default, we use a brute-force mode where the expression $F_n(X) == CircuitFn(X)$ is resolved for each X value ($2^{N_{bits}}$ times), and each result is added as an assertion. This technique is effective for case studies in this paper with fewer input variables. However, when the number of logical inputs is large, we resort to CEGIS [20], a well-established synthesis technique for handling larger search spaces.

We use bitvector variables over integers for the *circuit selection*, *input location*, and *output location* variables, due to their superior performance especially given the relatively small range of these variables.

For optimization, we by default use the built-in SMT optimizer to minimize a set of cost functions. For larger problems where this method may time out, we provide an alternative implementation which uses a cycle of synthesize calls with a decreasing cost constraint.

The Fridge Compiler is currently implemented in Python 3, utilizing Z3 [13] as the SMT solver. An earlier version was implemented in Rosette [24].

3.4 Flexibility of Architecture Choice

The Fridge Compiler can in theory accommodate a variety of architectures, with three common preprogrammed examples serving as guiding templates (see Figure

3). These built-in architectures showcase the Fridge Compiler's ability to handle diverse DSD architectures and provide a blueprint for users to specify custom ones or support alternate molecular architectures.

4 One-shot Cyclic Circuit Synthesis

One-shot, or single-use, computation refers to computation where circuit components are consumed during evaluation of each specific input. While there is ongoing work towards implementing reusable DSD circuits, large-scale DSD systems have implemented one-shot computation in practice [8]. As depicted in Figure 1, input strands cause a sequence of cascading reactions leading to the depletion of circuit components along with an output readout. Consequently, the circuit must be newly prepared for each desired input.

Cyclic circuits, while usually associated with memory circuits like latches and flip-flops, can also be found in combinational circuits that only depend on current inputs. As early as 1953, Claude Shannon found that allowing cycles in circuits is necessary for obtaining optimal combinational circuits [17]. Significant advancements in the understanding of cyclic combinational circuits have been made by formalizing the terminology, providing comprehensive analysis strategies, proving optimality for some classes of circuits, and developing synthesis techniques to incorporate cycles into smaller combinational circuits [16,1,6]. Despite the potential of cyclic combinational circuits to optimize circuit size, they are not widely supported by mainstream compilers.

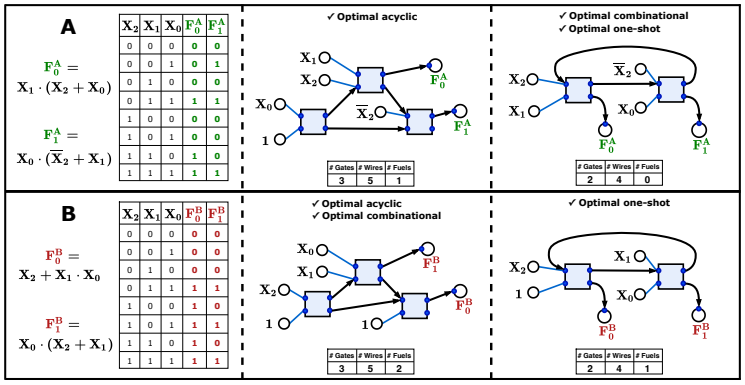

Fig. 5. **A.** Cyclic combinational circuits. *(Left)* Function definition for F^A; *(Middle)* its optimal acyclic implementation; *(Right)* its optimal cyclic combinational. **B.** *(Left)* Function definition for F^B; *(Middle)* its optimal acyclic implementation; *(Right)* its optimal one-shot cyclic implementation.

To analyze cyclic circuits we can assign an initial value C_i to a wire along each cycle i, and check if the circuit retains any values C_i once the circuit is

evaluated on an input. In conventional cyclic combinational circuits, every wire and output is assigned a specific value after circuit evaluation for all input values, ensuring no C_i values are left [16]. Figure 5A shows function F^A implemented through the smallest possible acyclic circuit (left) and through an even smaller combinational cyclic circuit (right). Going through the input combinations we can verify that the outputs are always forced to `True` or `False` – there is no input which leaves the cycle value unresolved, so this circuit is combinational.

We introduce one-shot cyclic circuits, which are cyclic circuits that leverage properties of one-shot circuits to create smaller combinational circuits not possible in traditional settings. These circuits are smaller than both conventional acyclic and cyclic combinational circuits.

CRN-based circuits, characterized by their one-shot assumption (that no previous input exists) have the unique property of initializing all cycle values C_i to `False`, as the component gates start un-reacted without an input. Therefore, any C_i values remaining on the output or along cycles in one-shot CRN-based circuits will actually result in a concrete output of `False`. Figure 5(B) shows function F^B implemented through the smallest possible acyclic circuit (left) and through an even smaller one-shot cyclic circuit (right). Going through the input combinations we find that both outputs F_0^B and F_1^B are dependent on the cycle value for the input $XYZ = 001$. Without the one-shot assumption, this circuit would not be combinational since a previous cycle value would influence the current output. However assuming a one-shot setting, the circuit in this case outputs the initial C_i value of `False`. The resulting truth table and functions are shown. Unlike for F^A, F^B does not have a cyclic combinational circuit smaller than the acyclic implementation, but does in fact have a smaller one-shot cyclic circuit.

This example shows that the one-shot assumption may allow for dual-rail circuits that are smaller than any traditional combinational circuit (acyclic or cyclic). As shown in Section 5.3, this property could be verified for any arbitrary inventory using the Fridge Compiler due to its completeness guarantee Any valid combinational circuit (acyclic or cyclic) is also a valid one-shot circuit, since it has outputs that are uninfluenced by their (possible) cycle values, meaning that initializing the cycle values to `False` in a one-shot setting will not impact the circuit's behavior. In contrast, one-shot cyclic circuits are distinct from cyclic combinational circuits as they rely on a previous circuit state, where all gates are initially `False`. Therefore the one-shot assumption creates a larger set of possible solutions, while encompassing both acyclic or cyclic combinational circuits. The Fridge compiler assumes the presence of cycles in its inventory and generates cycles in the resulting circuits when necessary to ensure optimality.

5 Case Studies & Empirical Evaluation

A natural question is: what functions can inventories cover? We chose three sets of Boolean function classes to test: all 256 3-bit predicates, all 65,536 4-bit predicates, and all 32,640 3-bit 2-predicate functions. The 3-bit predicate set was

Test Inventory	Gates	Wires	Function Set	# Functions	# SAT		MIN (sec)		50th % (sec)		99th % (sec)		MAX (sec)	
						Version	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
5G 12W [Thachuk]	5	12	3-bit 1-pred	256	246	*Existence*	0.01	0.06	0.02	0.06	0.09	0.10	0.10	0.10
						Optimal	0.01	0.02	0.02	0.03	0.11	0.05	0.20	0.05
5G 12W [Thachuk]	5	12	4-bit 1-pred	65536	22394	*Existence*	0.02	0.05	0.05	0.23	0.29	0.64	0.75	1.69
						Optimal	0.02	0.03	0.07	0.07	0.56	0.19	1.49	0.64
19G 27W (Large Custom)	19	27	4-bit 1-pred	65536	65536	*Existence*	0.04	N/A	0.05	N/A	0.13	N/A	0.52	N/A
						Optimal	0.04	N/A	0.39	N/A	21.60	N/A	181	N/A
3-Gate *Acyclic* Complete	3	15	3-bit 2-pred	32640	3936	*Existence*	0.02	0.02	0.02	0.02	0.02	0.03	0.08	0.09
						Optimal	0.01	0.01	0.01	0.01	0.03	0.02	0.05	0.1
3-Gate *Cyclic* Complete	3	26	3-bit 2-pred	32640	4824	*Existence*	0.02	0.02	0.02	0.02	0.03	0.03	0.04	0.11
						Optimal	0.01	0.01	0.02	0.01	0.03	0.02	0.16	0.07
4-Gate *Acyclic* Complete	4	28	3-bit 2-pred	32640	11384	*Existence*	0.02	0.03	0.03	0.10	0.09	0.25	0.22	0.51
						Optimal	0.02	0.02	0.12	0.07	0.50	0.16	4.13	0.51
4-Gate *Cyclic* Complete	4	50	3-bit 2-pred	32640	13604	*Existence*	0.03	0.05	0.04	0.17	0.18	0.88	0.46	6.66
						Optimal	0.02	0.03	0.14	0.09	2.02	0.53	46.57	3.45

Fig. 6. Runtimes and coverage results for case studies of various inventories and functions. For instance, the last row tells us that for the complete inventory of 4 gates and all possible wires (including feedback wires), the Fridge Compiler was run (in existence and optimality mode) with all functions of 3 input bits and 2 output predicates. Of those 32,640 target functions, 13,604 were implementable with the inventory. Of the 13,604 optimal circuits found, 99% of them were found in 2 seconds or less.

chosen in order to demonstrate a qualitative comparison to the results from the original Cello work [14] that attempted to synthesize (and experimentally realize) all 3-bit predicate functions. The 4-bit predicate set was chosen as a natural extension and in order to support known designs[15], and the 3-bit 2-predicate set was chosen to highlight the resource-sharing benefits of cyclic circuits. The primary goal of these case studies is to demonstrate a possible use-case of the Fridge Compiler. The current approach addresses problem instance sizes that are currently feasible to build experimentally. Although there is no present need for the Fridge Compiler to handle significantly larger problem instances, many techniques such as those discussed in Section 3.3 have shown to improve program synthesis runtime substantially without changing the underlying setup. Future work would include systematically characterizing larger inventories with a larger number of logical inputs.

5.1 Case Study: Synthesizing all k-bit Predicates

After running the Fridge Compiler on all 256 3-bit predicates, it turns out the breadboard inventory of Figure 4 covers 246/256 3-bit predicates. One of the functions this inventory cannot cover is odd parity. The Fridge Compiler guarantees that given the architecture, the inventory, the function, and the constraints, no circuit can represent this function. We designed a sum-of-product circuit that covers 3-bit odd parity as a new inventory; that inventory covers all 256 3-bit predicates. This makes intuitive sense: odd parity is one of the most complex functions in terms of circuit complexity.

5.2 Case Study: Inventory for all 4-bit Predicates

The 5-gate 12-wire inventory from the previous case study covers 22,394/65,536 4-bit predicates. A larger inventory was custom designed to cover all 65,536 4-bit

predicates: It contains 19 gates and 27 wires, and contains the odd parity circuit with a few extra gates. In order to test constraints, we allow this inventory to map input variable literals such as X_i or $\overline{X_i}$ to be mapped to more than one gate input. This constraint is valid in some architectures, and can often leads to more function coverage (see Figures 6, 7).

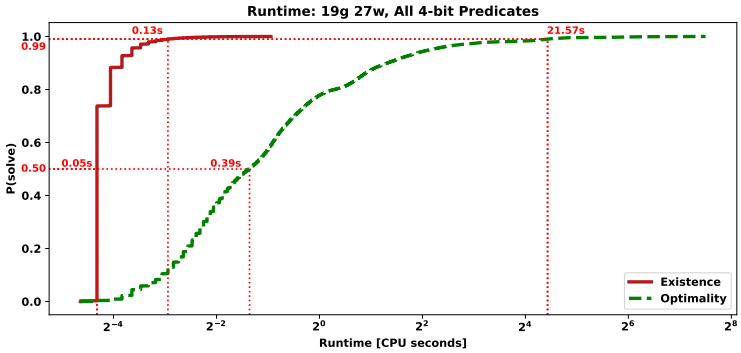

Fig. 7. A cumulative distribution function (CDF) curve illustrating runtime distribution for the test suite from Figure 6's third row. The X and Y axes represent runtime and proportion of tests completed in that time, respectively. This large custom inventory is able to synthesize all 65,536 4-bit functions. Existence finds any circuit that implements the function, and is typically solved faster than finding the optimal circuit.

5.3 Case study: Cyclic vs Acyclic

Cyclic circuits excel at optimizing resource-sharing between different parts of the circuit. To demonstrate this, we ask the Fridge Compiler to synthesize all 2-predicate functions with 3-bit inputs using a fridge of 3 Gates and any arbitrary wire connections. Figure 5 (Left) show 2 examples of functions in this set, and 2 implementations for each function. The Fridge Compiler finds 3,936 / 32,640 of the functions are implementable using 3 gates with acyclic circuits. By enabling permitting combinational cycles, an additional 72 functions can be realized, and 72 existing functions have smaller circuit implementations. For instance, Figure 5A shows a function that has a smaller circuit when cominational cycles are enabled. One-shot cycles can support even smaller circuits than combinational cyclic circuits. By enabling the Fridge Compiler to find one-shot cycles, we first observe that all 144 functions that were additionally found or improved with combinational cycles are still found, since combinational circuits are a subset of one-shot circuits. An additional 816 functions have circuit implementations, and 48 existing functions have smaller circuit implementations. Figure 5B shows a function where one-shot cycles enabled a smaller implementation while combinational cycles did

not. Synthesis over a a complete 4-gate inventory shows even better improvement. While 11,384 / 32640 functions are implementable using acyclic circuits, enabling one-shot cycles increases that amount to 13,604 / 32,640 functions synthesized.

5.4 Runtime Performance

Even while guaranteeing soundness, completeness, and optimality, Fridge Compiler runtimes are on the order of seconds on a personal laptop. Figure 6 shows the function coverage and runtimes for all case studies listed. In addition, the cumulative distribution function (CDF) in Figure 7 visually compares the runtimes of the 19G-27W inventory. All of the existence queries were completed within seconds, and the large majority of optimal solutions were found in less than a second. As expected with constraint solvers, there are a few combinations of inventories and circuits that take longer to optimize, as seen by the long tail in Figure 7. When comparing to the timeline of molecular programming, a matter of seconds to find an optimal solution is sufficiently fast.

6 Conclusion

We have proposed a new approach for molecular circuit synthesis that considers the available components "in the fridge", the Boolean function to be computed, and support for arbitrary constraints and user-defined cost functions. Our approach fills a need not met by the current menagerie of tools that support the development of molecular circuits, nor is the need met by existing circuit synthesis tools that were designed for electronic circuits; the latter misses important distinctions between electrical and molecular circuit properties. Importantly, our approach is sound, complete, and optimal. Empirical evaluations demonstrate that this approach is also efficient in practice with worst-case solutions found on the order of seconds. We have also introduced and supported synthesis of a new class of one-shot cyclic combinational circuits that can cover more functions and can yield smaller circuits for a given inventory.

In future work we intend to demonstrate the one-shot cyclic circuit in Figure 5B using a DSD circuit. We also plan to explore how to identify the minimum additional components to an existing inventory needed to support a set of functions, identify the smallest necessary inventory for a set of functions, and add a feature to balance stock depletion of fridge components. Furthermore, we plan to extend our work to include other molecular computing architectures such as genetic and protein circuits. We will also try adapting this methodology towards synthesizing analog, probabilistic, and/or continuous piecewise linear functions.

References

1. Backes, J.D., Riedel, M.D.: The synthesis of cyclic dependencies with Boolean satisfiability. ACM Trans. Design Autom. Electr. Syst. **17**, 44:1–44:24 (2012)

2. Badelt, S., Shin, S.W., Johnson, R.F., Dong, Q., Thachuk, C., Winfree, E.: A general-purpose CRN-to-DSD compiler with formal verification, optimization, and simulation capabilities. In: DNA Computing and Molecular Programming. pp. 232–248. Lecture Notes in Computer Science (2017)
3. Brayton, R.K., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: International Conference on Computer Aided Verification (2010)
4. Cardelli, L.: Two-domain DNA strand displacement. Mathematical Structures in Computer Science **23**, 247 – 271 (2010)
5. Chen, H.L., Doty, D., Soloveichik, D.: Rate-independent computation in continuous chemical reaction networks. In: Proceedings of the 5th Conference on Innovations in Theoretical Computer Science. p. 313–326. ITCS '14, Association for Computing Machinery (2014)
6. Chen, J.H., Chen, Y.C., Weng, W.C., Huang, C.Y., Wang, C.Y.: Synthesis and verification of cyclic combinational circuits. 2015 28th IEEE International System-on-Chip Conference (SOCC) pp. 257–262 (2015)
7. Chen, Y.J., Dalchau, N., Srinivas, N., Phillips, A., Cardelli, L., Soloveichik, D., Seelig, G.: Programmable chemical controllers made from DNA. Nature Nanotechnology **8**(10), 755–762 (2013). https://doi.org/10.1038/nnano.2013.189
8. Eshra, A., Shah, S., Song, T., Reif, J.H.: Renewable DNA hairpin-based logic circuits. IEEE Transactions on Nanotechnology **18**, 252–259 (2019)
9. Haaswijk, W., Soeken, M., Mishchenko, A., Micheli, G.D.: SAT-based exact synthesis: Encodings, topology families, and parallelism. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**, 871–884 (2020)
10. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. 2010 ACM/IEEE 32nd International Conference on Software Engineering **1**, 215–224 (2010)
11. Jones, T.S., Oliveira, S.M.D., Myers, C.J., Voigt, C.A., Densmore, D.M.: Genetic circuit design automation with Cello 2.0. Nature Protocols **17**, 1097 – 1113 (2022)
12. Lakin, M.R., Youssef, S., Polo, F., Emmott, S., Phillips, A.: Visual DSD: a design and analysis tool for DNA strand displacement systems. Bioinformatics **27**, 3211 – 3213 (2011)
13. de Moura, L.M., Bjørner, N.S.: Z3: An efficient SMT solver. In: International Conference on Tools and Algorithms for Construction and Analysis of Systems (2008)
14. Nielsen, A.A.K., Der, B.S., Shin, J., Vaidyanathan, P., Paralanov, V., Strychalski, E.A., Ross, D., Densmore, D.M., Voigt, C.A.: Genetic circuit design automation. Science **352** (2016)
15. Qian, L., Winfree, E.: Scaling up digital circuit computation with DNA strand displacement cascades. Science **332**, 1196 – 1201 (2011)
16. Riedel, M.D.: Cyclic combinational circuits. California Institute of Technology (2004)
17. Shannon, C.E.: Realization of All 16 Switching Functions of Two Variables Requires 18 Contacts: Bell Laboratories Memorandum, pp. 711–714. Wiley-IEEE Press (1953)
18. Soeken, M., Haaswijk, W., Testa, E., Mishchenko, A., Amarù, L.G., Brayton, R.K., Micheli, G.D.: Practical exact synthesis. 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE) pp. 309–314 (2018)
19. Soeken, M., Riener, H., Haaswijk, W., Micheli, G.D.: The EPFL logic synthesis libraries. ArXiv **abs/1805.05121** (2018)

20. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems. pp. 404–415 (2006)
21. Soloveichik, D., Seelig, G., Winfree, E.: DNA as a universal substrate for chemical kinetics. Proceedings of the National Academy of Sciences **107**, 5393 – 5398 (2009)
22. Testa, E., Zografos, O., Soeken, M., Vaysset, A., Manfrini, M., Lauwereins, R., Micheli, G.D.: Inverter propagation and fan-out constraints for beyond-CMOS majority-based technologies. 2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI) pp. 164–169 (2017)
23. Thachuk, C., Winfree, E.: A fast, robust, and reconfigurable molecular circuit breadboard. In: 15th Annual Conference on Foundations of Nanoscience (2018), `https://thachuk.com/talk/2018-fnano-invited/2018-FNANO-invited.pdf`, invited Talk
24. Torlak, E., Bodík, R.: Growing solver-aided languages with Rosette. In: SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software (2013)
25. Wolf, C.: Yosys open synthesis suite (2016)
26. Yordanov, B., Hamadi, Y., Kugler, H., Wintersteiger, C.M.: Z34Bio: An SMT-based framework for analyzing biological computation. In: SMT Workshop 2013 11th International Workshop on Satisfiability Modulo Theories (2013)