

# Design of Novel Analog Compute Paradigms with Ark

Yu-Neng Wang<sup>1</sup>, Glenn Cowan<sup>2</sup>, Ulrich Rührmair<sup>3,4</sup>, Sara Achour<sup>1</sup>

<sup>1</sup>Stanford University, <sup>2</sup>Concordia University, <sup>3</sup>TU Berlin, <sup>4</sup>University of Connecticut  
{wynwyn,sachour}@stanford.edu, gcowan@ece.concordia.ca, ruehrmair@ilo.de

## Abstract

*Previous efforts on reconfigurable analog circuits mostly focused on specialized analog circuits, produced through careful co-design, or on highly reconfigurable, but relatively resource inefficient, accelerators that implement analog compute paradigms. This work deals with an intermediate point in the design space: Specialized reconfigurable circuits for analog compute paradigms. This class of circuits requires new methodologies for performing co-design, as prior techniques are typically highly specialized to conventional circuit classes (e.g., filters, ADCs).*

*In this context, we present Ark, a programming language for describing analog compute paradigms. Ark enables progressive incorporation of analog behaviors into computations, and deploys a validator and dynamical system compiler for verifying and simulating computations. We use Ark to codify the design space for three different exemplary circuit design problems, and demonstrate that Ark helps exploring design trade-offs and evaluating the impact of nonidealities to the computation.*

## 1. Introduction

There has been significant interest in domain-specific reconfigurable analog computing platforms that perform in-sensor and near- or in-memory computation and solve computationally hard problems [40, 19, 33, 29, 41]. These analog architectures have applications in machine vision, medical devices, and robotics domains, [26, 35, 30] and reduce data movement across the analog/digital interface, enabling the use of more resource-efficient digital hardware [1, 34]. These analog architectures are implemented in circuits with good performance and typically incorporated into a larger domain-specific computation: examples include scientific computation and sensor processing pipelines. [11, 24, 42, 21]

**Classical Analog Circuits.** Designers have developed highly specialized classical analog circuits (e.g., filters, ADCs) with programmability and fidelity characteristics tailored for specific application use cases. These circuits typically offer limited programmability and target a specific fidelity and therefore can be engineered to be highly resource-efficient. Because these circuits are largely non-programmable, much of the specialization is done in the design stage. Fortunately, the design requirements for classical circuits can be described with standard figures of merit and functional specifications, and circuit designers have a good intuition on how to craft an implementation that meets the requirements. The existence of these standard interfaces be-

tween hardware designers and domain specialists is critical; otherwise, design-time specialization would be highly challenging.

**Unconventional Analog Accelerators.** Researchers have developed highly reconfigurable analog accelerators that faithfully implement radical new forms of computation, such as GPAC computing, oscillator-based computing, spiking neural networks, and cellular non-linear networks. [11, 24, 10, 20, 14, 47] Many of these accelerators require unconventional uses of analog circuits to implement novel computational operators or compute on different signal components (e.g., transient behavior). Therefore, these accelerators faithfully implement their respective analog compute paradigms and are highly programmable. Supporting this degree of generality comes at a substantial resource and performance cost in analog design, impacting the hardware platform’s scalability and efficiency. In contrast to specialized analog circuits, these accelerators are not typically developed through careful co-design, as that hardware is sufficiently flexible to support computations that are expressible in the target compute paradigm.

### 1.1. Domain-Specific Unconventional Analog Circuits

Thus far, prior work has focused on two extremes in the analog design space (1) highly specialized classical circuits and (2) highly programmable analog accelerators that target unconventional analog compute paradigms. We anticipate there is a largely untapped part of this design space: highly specialized analog circuits that perform computation using an unconventional analog compute paradigm (Figure 1). Because these circuits are specialized to a particular application domain, the circuit’s programmability and fidelity requirements can potentially be reduced, lowering resource costs and improving performance.

**Design Challenges.** To realize this new part of the design space, we need to specialize unconventional analog circuits to the target application domain. However, because these designs are typically non-standard, methods for specifying functional behavior and design requirements for classical circuits may not sufficiently capture the relevant behaviors and design requirements. Second, there is less design intuition for unconventional circuits, as they haven’t been studied as extensively as ADCs, filters, and other classical analog circuits. Therefore, it is highly challenging for the domain specialist and analog designer to navigate this space effectively and arrive at a promising design.

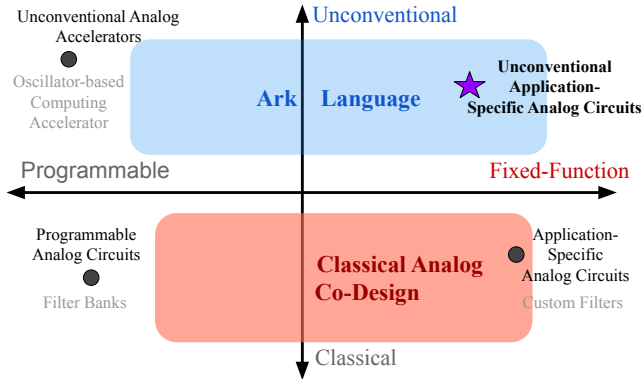


Figure 1: Analog design space exploration.

### 1.2. Analog Design with Ark

We introduce Ark, a programming language that enables design space exploration of unconventional analog circuits that leverage analog compute paradigms. With Ark, analog designers can expose different programmability and fidelity tradeoffs for domain specialists to explore within the context of their workload. The Ark language offers the following capabilities:

- **Analog Compute Paradigms as DSLs:** Ark supports the specification of new analog compute paradigms as domain-specific languages (DSLs), and deploys a compiler that enables the validation and simulation of computations written in an analog compute DSL.
- **Codifying Analog Design Spaces with DSLs:** With Ark, analog designers can specialize a compute paradigm DSL to incorporate new language constructs that capture bottom-up analog design constraints, model nonidealities, and codify different design tradeoffs.
- **Progressive Design of Analog Circuit Descriptions:** Ark supports the specification, configuration, and simulation of reconfigurable analog computations. Ark imposes strict inheritance rules to ensure computations written in the original compute paradigm can be progressively rewritten to selectively incorporate analog behaviors.

**Design Flow.** With Ark, both the domain specialist and the analog designer settle on an analog compute paradigm to serve as a basis for building both analog circuit designs and computations. Each analog compute paradigm offers basic computational operators that modify certain properties (e.g., phase) of the underlying analog signal and can be simulated with a system of differential equations. The domain specialist first develops reconfigurable analog computations in the agreed upon analog compute DSL for their target workloads, and the analog designer extends the analog compute DSL to codify the analog design space.

The domain specialist then progressively adapts computations to use designer-provided constructs to analyze the effect of analog nonidealities in their computation and explore different analog design options. The analog designer may then declare new constructs that codify new design points based on the domain specialist’s usage of their language extension.

Reconfigurable analog computations defined entirely with analog hardware constructs serve as a functional and requirements specification for unconventional analog circuits. We anticipate this flow enables iterative co-design of analog circuits.

### 1.3. Contributions

- **Dynamical Graph Representation:** We present a novel, unified intermediate representation termed a dynamical graph (DG) for both analog computations and analog circuit descriptions.
- **The Ark Language:** We present a programming language that supports the definition of domain-specific languages that codify analog compute paradigms and their associated hardware design spaces. Reconfigurable analog computations can then be written in DSLs in the defined language.
- **Ark Compiler and Validator:** We present a compiler that derives the system of differential equations that simulates the transient dynamics of a given Ark program and a validator that verifies that a given Ark program satisfies all of the constraints imposed by the domain-specific language.
- **Case Study and Evaluation:** We use Ark to codify the design tradeoffs associated with a transmission-line based analog PUF (TLN), a cellular non-linear network (CNN) analog accelerator, and an oscillator-based computing (OBC) analog accelerator. We demonstrate that Ark can capture nonidealities and design tradeoffs associated with these design problems and provide a detailed analysis for the PUF design problem.

## 2. Case Study: Transmission Line PUF

We start with an illustrative example to give an overview of the language components in Ark and demonstrate the design flow enabled by Ark. This case study focuses on the design of an unconventional analog circuit that implements a *physical unclonable function* (PUF) [22], a security primitive that leverages fabrication variations to produce a difficult-to-spoof hardware authentication device. Given a challenge bitvector, a PUF produces a response bitvector that is highly sensitive to fabrication variation. In a well-designed PUF, the mapping between challenge and response should be stable but maximally complex and hard to imitate or predict for cryptographic adversaries without physically possessing and interrogating the PUF. In an analog circuit PUF, the response is often naturally computed from voltage and current trajectories observed on a wire within a certain observation time window.

The security expert opts to target the *transmission line network (TLN) compute paradigm* [13] and selects the TLN DSL (Section 4.4) provided by Ark. To investigate the effect of fabrication variation, the expert uses the GmC-TLN extension of the TLN DSL (Section 4.5), which codifies the design space of fabrication-variation-sensitive GmC circuit implementations.

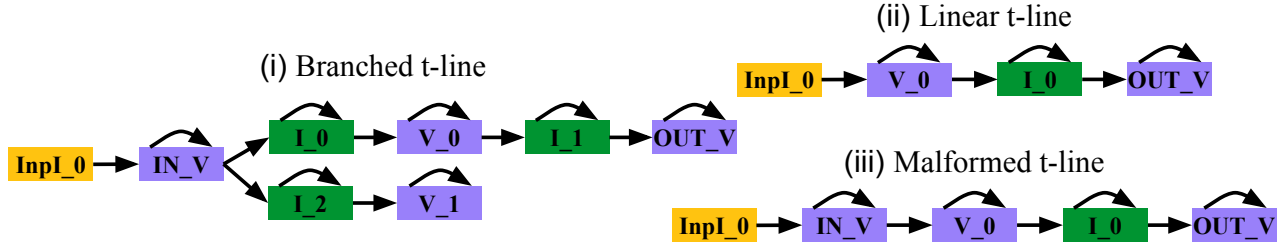


Figure 2: Dynamical graphs of branched, linear, and malformed t-lines.

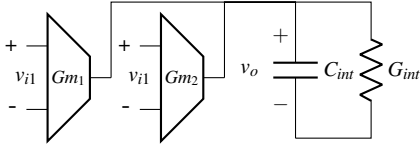


Figure 3: A GmC integrator schematic.

## 2.1. The TLN Compute Paradigm

A *transmission line* (t-line) is a channel that carries electromagnetic waves across some distance. The traversal of a wave through a line is modeled with the discretized *Telegrapher's equations* [25]:

$$\begin{cases} \frac{dV_i}{dt} = \frac{1}{C_i}(I_i - I_{i+1} - G \cdot V_i) \\ \frac{dI_i}{dt} = \frac{1}{L_i}(V_{i-1} - V_i - R \cdot I_i) \end{cases} \quad (1)$$

The t-line is segmented into  $0.., i.., n$  segments, where  $V_i$  and  $I_i$  models the voltage and current at each line segment  $i$ . The  $R$  and  $G$  parameters model impedance, which attenuates signals, and  $L$  and  $C$  parameters model propagation speeds, which delay signals and set the characteristic impedance. A *transmission line network* is a network of interconnected t-lines that route signals and leverage the delay, reflection, and transmission for computation. *Reflection* and *transmission* occur when the characteristic impedance is changed within a line, such as when branches are introduced or when a line is terminated.

## 2.2. Exploring TLN Topologies with Ark

Ark provides a TLN DSL that implements transmission line networks modeled with the Telegrapher's equations. Figure 2-(i) presents a branched t-line network implemented in the TLN DSL and formulated as a dynamical graph (Section 3). A dynamical graph is a typed, directed graph with nodes and edges that map to variables and dynamics in the underlying dynamical system, respectively. In the branched t-line, the  $\mathbf{v}$ ,  $\mathbf{I}$  node types (purple, green) map to  $V_i$  and  $I_i$  variables in the Telegrapher's equation model, and the  $\mathbf{InpI}$  node type (yellow) injects an external signal into the transmission line network.

**Attributes.** Each node and edge type defines attributes fixed to values at simulation time. The  $\mathbf{v}$  and  $\mathbf{I}$  node types define real-valued  $\mathbf{c/g}$  and  $\mathbf{1/r}$  attributes, respectively, which map to the  $C$ ,  $G$ ,  $L$  and  $R$  parameters in the Telegrapher's equations. In the above example, all  $\mathbf{1, c}$  attributes are set to  $\mathbf{1e-9}$ , and all  $\mathbf{g, r}$  attributes are set to  $\mathbf{0}$  for nodes in the middle of the line

and  $\mathbf{1}$  for the  $\mathbf{IN\_v}$  and  $\mathbf{OUT\_v}$ . The  $\mathbf{InpI}$  node type defines a function attribute that is assigned an input pulse function  $\mathbf{pulse(t, 0, 2e-8)}$  in the branched t-line. The node and edge types and attributes are defined in TLN DSL.

**Dynamics.** In the dynamical graph, interactions are *local*, and neighboring edges contribute terms to the differential equations. In branched t-line, the incoming, outgoing, and self-referencing edges contribute  $I/V.c$ ,  $-I/V.c$ ,  $-V.g/V.c \cdot V$  terms to each  $V$  node's differential equations, and contribute  $V/I.l$ ,  $-V/I.l$ ,  $-I.r/I.l \cdot I$  terms to each  $I$  node's differential equations. The TLN language defines the production rules for translating connections to algebraic terms.

Figure 2-(iii) presents a *malformed* TLN dynamical graph that is reported invalid by the TLN language because it includes a  $\mathbf{v-v}$  connection which introduces unexpected voltage terms into the underlying differential equations. The TLN language requires valid TLNs to have alternating  $\mathbf{I}$  and  $\mathbf{V}$  nodes to ensure the Telegrapher's equations are faithfully implemented.

**Analysis.** We simulate the voltage trajectory<sup>1</sup> at node  $\mathbf{OUT\_v}$  for the branched t-line (Figure 4a) and a linear, non-branched t-line (Figure 4b) using differential equations generated by the TLN dynamical system compiler. The branched t-line produces a weaker initial pulse ( $\mathbf{0.5} \rightarrow \mathbf{0.3}$ ) and an "echo" of the initial pulse after  $\mathbf{4e-8}$  seconds have elapsed (the shaded area in Figure 4a). This echo occurs because part of the injected pulse travels down the branch and reflects back to the main line, where the pulse then splits and travels to both the  $\mathbf{OUT\_v}$  and  $\mathbf{IN\_v}$  nodes. This echoing behavior can potentially be exploited to design PUFs with more complicated system dynamics.

These trajectories can be used to set signal observation windows. The linear t-line and branched t-line require observation windows of  $\mathbf{1e-8}$  to  $\mathbf{3e-8}$  seconds and  $\mathbf{1e-8}$  to  $\mathbf{8e-8}$  seconds respectively. The branched t-line is assigned a larger observation window to ensure that at least one of the signal echoes is captured in the response encoding.

## 2.3. GmC Circuit Implementation of TLN Computing

A transmission line network is efficiently emulatable with a network of GmC integrators [27]. Each GmC integrator (Figure 3) consists of transconductors  $Gm_1$  and  $Gm_2$  (implemented with several transistors) that convert input voltage signals,  $v_{i1}$  and  $v_{i2}$ , into currents  $i_{i1}$  and  $i_{i2}$ , where  $i_{i1} = Gm_1 \cdot v_{i1}$

<sup>1</sup>We simulate 53-node branched and linear lines (Figure 2-(i), (ii)).

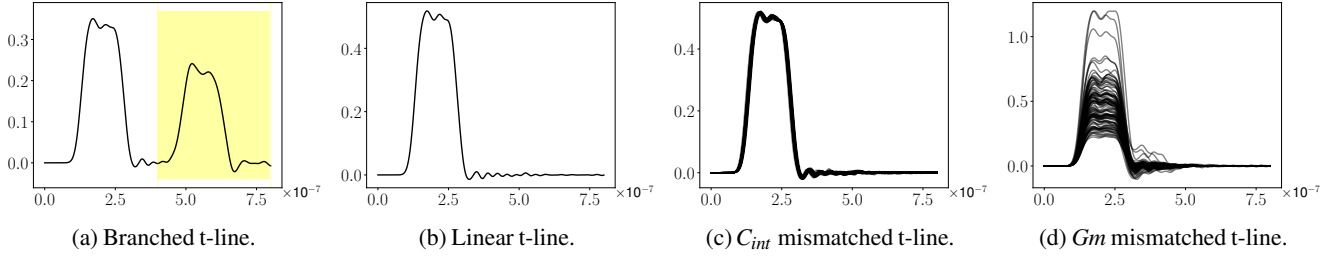


Figure 4: Dynamics of t-lines observed at `OUT_v`.

$i_{i2} = Gm_2 \cdot v_{i2}$ . The resistor with conductance  $G_{int}$  contributes to a current  $i_o$ , where  $i_o = G_{int} \cdot v_o$ . The  $i_{i1}$ ,  $i_{i2}$ , and  $i_o$  currents all flow out of the capacitor  $C_{int}$ , yielding the following dynamics:

$$\frac{dv_o}{dt} = \frac{1}{C_{int}} (-Gm_1 \cdot v_{i1} - Gm_2 \cdot v_{i2} - G_{int} \cdot v_o) \quad (2)$$

The above dynamics implement the  $\frac{dI_i}{dt}$  differential equation from the telegrapher's equation math model, provided  $-Gm_1 = Gm_2 = Gm$ . The  $v_o$ ,  $v_{i1}$ , and  $v_{i2}$  voltages map to  $I_i$ ,  $V_{i-1}$  and  $V_i$  respectively, and the  $Gm/G_{int}$  and  $C_{int}/Gm$  circuit quantities implement  $G$  and  $L$  parameters. The above dynamics also implement the  $\frac{dV_i}{dt}$  differential equation; the  $v_o$ ,  $v_{i1}$ , and  $v_{i2}$  voltages maps to  $V_i$ ,  $I_i$  and  $I_{i+1}$  variables and the  $G_{int}/Gm$  and  $C_{int}/Gm$  terms implement  $R$  and  $C$ .

**Modified Telegrapher's Equations.** If we relax the  $-Gm_1 = Gm_2 = Gm$  constraint to allow the device parameter magnitudes  $|gm_1|$ ,  $|gm_2|$  and  $|gm|$  to differ, then we can introduce  $ws$  and  $wt$  parameters in the TLN dynamics and implement a more flexible version of the telegrapher's equations:

$$\begin{cases} \frac{dV_i}{dt} = \frac{1}{C_i} (wt_i \cdot I_i - ws_{i+1} \cdot I_{i+1} - G_i \cdot V_i) \\ \frac{dI_i}{dt} = \frac{1}{L_i} (wt_{i-1} \cdot V_{i-1} - ws_i \cdot V_i - R_i \cdot I_i) \end{cases} \quad (3)$$

With this relaxed circuit usage, the  $C_{int}$  and  $G_{int}$  device parameters implement  $C/L$  and  $G/R$  respectively, and the  $-Gm_1$  and  $Gm_2$  device parameters implement  $wt$  and  $ws$ . When  $wt_i = ws_i = 1$ , the GmC circuit implements TLN computing.

#### 2.4. Exploring Analog Mismatch with Ark

The PUF's security properties come, in part, from its sensitivity to fabrication variations (e.g., device mismatch). In a variation-sensitive PUF design, fabricated instances of the same PUF behave differently when provided with the same input and can therefore be used to identify an individual uniquely.

We use the GmC-TLN extension to the TLN computing model, which codifies the design space of mismatch-sensitive

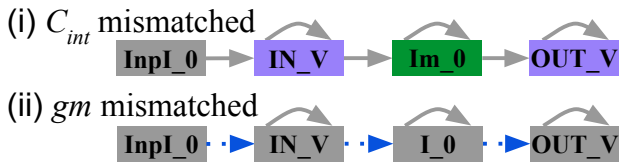


Figure 5: Dynamical graphs of mismatched t-lines.

GmC circuit implementations, to study the effects of process variation on a linear t-line. The node types  $\mathbf{Vm}$  and  $\mathbf{Im}$  inherit  $\mathbf{V}$  and  $\mathbf{I}$ , respectively, and override the  $\mathbf{C}$  and  $\mathbf{L}$  attributes to model the random mismatch associated with the corresponding  $C_{int}$  device parameter. The edge type  $\mathbf{Em}$  inherits  $\mathbf{E}$  and adds two new mismatched attributes  $ws$  and  $wt$ , corresponding to the  $Gm$  random mismatch. The GmC-TLN language uses the modified Telegraphers equations to model the dynamics of the  $\mathbf{Em}$  edges. In the GmC-TLN language, mismatched parameter values are sampled from a normal distribution with a 10% relative standard deviation before execution.

**Analysis.** We use the GmC-TLN language to explore the effects of different sources of process variation on the linear TLN. Ark's inheritance system ensures (1) the original linear t-line can be simulated in the GmC-TLN language and deliver the same dynamics, (2) nodes derived from TLN language nodes can be substituted into the dynamical graph. The process variation sensitive linear t-lines in Figures 5 substitute in  $\mathbf{Vm}/\mathbf{Im}$  node types (pink, green) and  $\mathbf{Em}$  edge types respectively to selectively model the effects of  $C_{int}$  and  $Gm$  mismatch.

Figures 4c and 4d present the `OUT_v` voltage trajectories over 100 sampled device mismatches for the  $C_{int}$ -sensitive t-line and the  $Gm$ -sensitive t-line respectively. We observe that within the  $1e-8$  to  $3e-8$  observation window, the  $Gm$ -sensitive t-line experiences a much greater degree of variation across trials than the  $C_{int}$ -sensitive t-line. This observation has several implications: (1) future TLN architectures should use  $Gm$  mismatch instead of  $C_{int}$  mismatch; (2) the analog designer should pursue GmC circuit variants that maximize  $Gm$  mismatch – these variants can be provided to the domain specialist as new edge types. Therefore, domain specialists can use Ark to assist in design space exploration and help set the direction of analog design efforts.

### 3. The Dynamical Graph Computational Model

Ark employs a higher order computational model, termed a Dynamical Graph (DG) that is specialized to implement different analog computational paradigms. The DG contains a set of nodes  $\mathbf{N}$  and a set of directed edges  $\mathbf{E}$  and can be interpreted as a system of differential equations that describes a computation. **Typed Nodes and Edges.** All elements in the DG are *typed*, meaning each node  $n \in \mathbf{N}$  and edge  $e \in \mathbf{E}$  belong to a node type  $NT$  and an edge type  $ET$  respectively. Node and edge types may define named attributes, and a node type  $NT$  has an

additional *reduction* operator  $\bigwedge_j \in \{\Sigma, \Pi\}$  and *order*  $p$  defined. The type, attributes, order, and reduction are necessary for deriving the graph dynamics.

**Dynamics.** Each node  $n \in \mathbb{N}$  maps to a variable  $Q$  in the underlying dynamical system, and each edge  $e \in \mathbb{E}$  contributes terms to the connected variables' dynamics. A node type with an order of 0 implements a pure function, and a node type with an order of  $p > 0$  implements  $p^{\text{th}}$  order differential equations. The DG works with a set of production functions  $\mathfrak{P}_{\text{in}}$ ,  $\mathfrak{P}_{\text{out}}$ , and  $\mathfrak{P}_{\text{self}}$  for incoming edges ( $\hat{n} \rightarrow n$ ), outgoing edges ( $\hat{n} \leftarrow n$ ), and self-referencing edges ( $\odot n$ ) respectively. Given a node  $n$  of node type  $NT$  with  $m$  incoming edges,  $q$  outgoing edges, and  $r$  self-reference edges, the production function takes input as the types of an edge and the source and destination nodes. The production function finds the production rule that matches the given types and returns an algebraic expression based on the rule and types. The expressions aggregate over edges with the reduction operator  $\bigwedge$ . Overall, the dynamics of the variable associated with a node are defined as follows:

$$\frac{d^p Q}{dt^p} = \bigwedge_{i=0}^{m-1} \mathfrak{P}_{\text{in}}(ET_i, NT_i, NT) + \bigwedge_{i=m}^{m+q-1} \mathfrak{P}_{\text{out}}(ET_i, NT, NT_i) + \bigwedge_{i=m+q}^{m+q+r-1} \mathfrak{P}_{\text{self}}(ET_i, NT) \quad (4)$$

The dynamics of a DG are fully specified with the differential equations collectively from all variables.

## 4. The Ark Programming Language

Figure 6 presents the Ark programming language. Ark languages specialize the DG computational model to implement different classes of analog computations, Ark functions generate dynamical graphs and can be invoked to generate DGs with different topologies and attribute parametrizations. Sections 4.1-4.2 describes the basic Ark language and function definition constructs, and 4.3 describes the Ark hardware extensions. Section 4.4-4.5 provides an illustrative example with TLN.

**Expressions.** Expressions are defined over variables, which include nodes, edges, and function arguments ( $v$ ), simulation time (**time**), and node and edge attributes  $v.v'$  (attribute  $v'$  of node  $v$ ). Boolean expressions  $b$  use logical and comparison operators and return boolean values. Math expressions ( $e$ ) use linear/non-linear math operators (e.g.,  $+$ ,  $\times$ ,  $\sin$ ,  $\cos$ , etc) and if-then-else statements (e.g. **if**  $b$  **then**  $e$  **else**  $e'$ ) and return real values. Ark checks that all variables in the variable are in scope and that expressions evaluate to their expected type.

**Datatypes.** Ark supports bounded real and integer datatypes (**real** $[x_0, x_1]$  and **int** $[i_0, i_1]$ ), and function datatypes **lambd**( $v^*$ ) that accept real-valued arguments and compute real values. Ark checks variable assignments to ensure the value matches the variable's datatype, and is contained within the datatype's value range  $[x_0, x_1]$ . All assigned functions

$x \in \mathbb{R}, s \in \mathbb{R}^{++}, i \in \mathbb{Z}, p \in \mathbb{N}_0$   
 $v \in \text{Literals}, e \in \text{Expressions}, b \in \text{BoolExpressions}$   
**inherit**( $(Rule)$ ) ::=  $Rule$  **inherits**  $v \mid Rule$

```

1  SigT      ::=  real[x0, x1] | real[x0, x1] mm(s0, s1)
2           | int[i0, i1] | lambd(v*)
3  SigTProg ::=  SigT | SigT const
4  Attr     ::=  attr v = SigTProg | init(i) SigTProg
5  Reduc    ::=  sum | mul
6  Type     ::=  node-type(p, Reduc) | edge-type
7           | edge-type fixed
8  ProdExpr ::=  v <= e | v <= e off
9  ProdRule ::=  prod(v0 : v1, v2 : v3 -> v4 : v5) ProdExpr
10 VAtom    ::=  p | inf
11 VMatch   ::=  match(VAtom, VAtom, v0, vn -> [v1*])
12           | match(VAtom, VAtom, v0, [v1*] -> vn)
13           | match(VAtom, VAtom, v0)
14 ValExpr  ::=  acc VMatch* | rej VMatch*
15 ValRule  ::=  cstr vn : v1 { ValExpr* } | extern-func v
16 LangSt   ::=  inherit(Type v {Attr*}) | ProdRule | ValRule
17 LangDef  ::=  inherit(lang v {LangSt*})
18
19 Val      ::=  x | lambd(v*) : e
20 FuncVal  ::=  Val | v
21 FuncSt   ::=  node v0 : v1 | edge < v0, v1 > v2 : v3
22           | set-attr v0.v1 = FuncVal
23           | set-edge v when b
24           | set-init v(i) = FuncVal
25 FuncArg  ::=  v : SigT | v0.v1 : SigT
26
27 FuncDef  ::=  func v0 (FuncArg*) uses v1 {FuncSt*}
28
29 Stmt     ::=  FuncDef | LangDef
30 Prog    ::=  Stmt*

```

Figure 6: Basic grammar for Ark language. Shaded expressions model hardware behavior.

**lambd**( $v^*$ ) :  $e$  must have the same number of arguments as the function datatype,

### 4.1. Language Definitions (Lines 1-17)

Each language definition declares node and edge types and defines production rules and validation rules over node and edge types. The typing information and production rules are used to specialize the dynamical graph computational model to implement the desired analog computational paradigm.

**Types.** [Lines 1-7] Each **node-type**  $v(p, Reduc)$  **Attr**\* statement defines a node type  $v$  with a variable order  $p$  and a reduction operator  $Reduc$ . The node type  $v$  contains attributes and initial value definitions **Attr**\* that specify the names and datatypes of attributes and initial values. Each **edge-type**  $v$  **Attr**\* statement defines an edge type named  $v$  with attributes **Attr**\*. Each attribute and initial value may optionally be assigned a constant value.

**Production Rules.** [Lines 8-9] Each **prod** statement defines a new production rule for a connection with an edge  $v_0$  of type  $v_1$ , a source node  $v_1$  of type  $v_3$ , and a destination node  $v_4$  of type  $v_5$ . The production expression ( $v <= e$ ) applies a term to the node  $v$ . A production rule is a self-referencing rule if the source and destination node have the same name ( $v_2=v_4$ ).

**Local Validity Rules.** [Lines 10-14] Ark supports the definition of local validity rules over the cardinalities of connected node and edge types (**cstr**). Each local validity rule **cstr**  $v_n : v_1$  **ValExpr\*** operates on a target node named  $v_n$  of node type  $v_1$  and returns true if the accepted validity expression (**acc**  $VMatch^*$ ) returns true and the rejected validity expression (**rej**  $VMatch^*$ ) returns false. Both expressions return true if the target node  $v_n$  is *described* by the pattern  $VMatch^*$ . Here, we say that a node is described by a pattern if we find an assignment of edges to clauses  $VMatch$  in the pattern such that every clause is assigned between  $VAtom_1$  and  $VAtom_2$  edges of edge type  $v_0$  connected to a node type contained in  $[v_1^*]$ . Ark provides rejected validity expression to support restricting the design space as an accepted expression can become invalid once a language extension is introduced.

**Global Validity Rules.** [Line 14] Ark supports the definition of a global validity check that is evaluated over the entire graph topology. The **extern-func**  $v$  statement binds an external validity checking algorithm  $v$ . Global connectivity checks are required to ensure the DG implements certain topologies, such as grid topologies.

**Semantic Checks.** Ark ensures all node and edge types have unique names, that each node type contains an initial value declaration **init**  $(i)$   $SigTProg$  for derivatives  $0 \dots i \dots p - 1$  of the node type, and ensures edge types contain only attribute statements. For each production rule, Ark checks the expression  $e$  only references variables instantiated in the **prod**  $(.)$  clause and that the node  $v$  in the production expression is either source node  $v_2$  or the destination node  $v_4$  from the clause. For each validity rule, Ark ensures match clauses reference the target node  $v_n$  and that all node and edge types are declared in the language.

#### 4.1.1. Inheritance

Ark supports single inheritance of languages, where the derived language inherits node and edge types, production rules, and validation rules from a parent language and may define node and edge types that inherit from types in the parent language. Ark constrains the derived language to ensure the parent and derived languages are compatible:

- Derived node and edge types inherit the parent type’s node order and reduction operator, and inherit all attributes and initial value declarations from the parent type.
- Inherited attributes and initial values can be redefined in the derived node or edge type but must retain the same datatype (real, integer, lambda) and operate on a smaller value range than the parent attribute.
- Production and validation rules from the parent class cannot be overridden or removed. Any new production or validation rules must include one new type from the derived class.
- For each connection, the most specific production rule is applied for a given combination of derived nodes and edge types. If no production rule exists, Ark falls back to a production rule that applies to the parent node and edge types. Ambiguities also produce an error.

The above properties ensure dynamic graphs comprised of derived types can be cast to the parent type, and dynamical graphs written in the parent language can be faithfully executed in the derived language and progressively rewritten to use node and edge types in the derived language.

#### 4.2. Function Declarations (Lines 19-27)

Ark supports defining functions that procedurally generate dynamic graphs from a set of function inputs. Each function declaration **func**  $v_0$  ( $FuncArg^*$ ) **uses**  $v_1$  specifies the Ark language  $v_1$  to use and a list of typed arguments  $FuncArg^*$  accepted by the function.

The function body contains statements that construct a dynamical graph parametrized over the function arguments. The **node**  $v_0 : v_1$  statement constructs a node named  $v_0$  of type  $v_1$ , and the **edge**  $\langle v_0, v_1 \rangle v_2 : v_3$  statement constructs an edge with name  $v_2$  of type  $v_3$  that connects the source node  $v_0$  to the destination node  $v_1$ . The **set-attr**  $v_0.v_1 = FuncVal$  statement sets an attribute  $v_1$  of node  $v_0$  to a value or function argument, and the **set-init**  $v(i) = FuncVal$  statement sets the initial value of the  $i^{\text{th}}$  derivative of node  $v$  to value or function argument. The **set-switch**  $v$  **when**  $b$  statement turns the switchable edge  $v$  on when  $b$  is true, where  $b$  is a boolean expression over function arguments.

**Semantic Checks** Ark checks that all referenced node and edge types are defined in the language, all referenced nodes/edges are defined within the function body, and all attributes and initial values defined in the node/edge type are set for each node, and all datatype assignments are valid.

#### 4.3. Ark Hardware Extensions

Ark offers language extensions (grey highlighted in Figure 6) for modeling analog-specific behaviors not already captured in the basic language features. The **real**  $[x_0, x_1]$  **mm**  $(s_0, s_1)$  datatype models process variation-sensitive attributes and initial values, and samples a mismatched value  $\hat{x}$  from a normal distribution  $N(x, x \cdot s_0 + s_1)$  when an attribute or initial value is assigned to a nominal value  $x$ . Each function invocation sets the random seed used to produce the same mismatched values. The seed can be varied across invocations to model multiple fabricated instances of a particular design.

Ark offers language constructs for defining non-programmable attributes and initial values (**attr**  $v \dots$  **const** and **init-val**  $(i) \dots$  **const**) and fixed edge types (**edge-type fixed**  $v_1$ ). Non-programmable switches are always on, and non-programmable attributes must be assigned to a constant value or math function on instantiation. Ark also supports defining production rules (**prod**  $\dots v \leftarrow e$  **off**) that model nonidealities associated with edges that are switched off. **Semantic Checks** For attributes and initial values that are assigned to function arguments, Ark checks that the associated definition in the type declaration is not **const**. Ark validates that all **set-switch** statements are applied to edges that are not **fixed**.

```

lang tln {
  ntyp(1, sum) V {attr c=real[1e-10, 1e-08],
    attr g=real[0, inf]};
  ntyp(1, sum) I {attr l=real[1e-10, 1e-08],
    attr r=real[0, inf]};
  ntyp(0, sum) InpV {attr fn=fn(a0), attr r=real[0, inf]};
  ntyp(0, sum) InpI {attr fn=fn(a0), attr g=real[0, inf]};
  etyp E {};
  prod(e:E, s:V->t:I) s<--var(t)/s.c;
  prod(e:E, s:V->t:I) t<=var(s)/t.l;
  ...
  cstr V {acc[
    match(0, inf, E, V->[I]), match(0, inf, E, [I]->V),
    match(0, inf, E, [InpV]->V),
    match(0, inf, E, [InpI]->V),
    match(1, 1, E, V)]}
  cstr I {acc[match(0, 1, E, I->[V]),
    match(0, 1, E, [V, InpV, InpI]->I),
    match(1, 1, E, I)]} ... }

```

Figure 7: Ark TLN language definition snippet. The `ntyp` and `etyp` are abbreviations of `node-type` and `edge-type` respectively

```

func br-func ( br:int[0,1]) uses tln{
  node IN_V:V;
  node I_0:I; node I_1:I;
  ...
  edge <IN_V, I_0>E_0:E; edge <I_0, V_1>E_1:E;
  edge <V_3, V_3>E_10:E;
  ...
  set-switch E_6 when br; set-attr IN_V.c=1e-09;
  set-attr IN_V.g=0.0;
  set-attr I_0.l=1e-09; ... }

```

Figure 8: Ark function snippet of branched T-Line

#### 4.4. Illustrative Example: TLN Language

We present an example TLN language definition in Figure 7. The TLN language declares `V` and `I` node types, which map to  $V$  and  $I$  terms in the discretized Telegrapher’s equations, and `InpV` and `InpI` node types, which provide external input voltage and currents into the TLN computation. The `V` and `I` node types contain real-valued `c/g` and `l/r` attributes which map to the  $C$ ,  $G$ ,  $L$ , and  $R$  parameters in the Telegrapher’s equations.

The TLN language defines production rules that derive the Telegrapher’s equations from node and edge types. For example, the `prod(e:E, s:V->t:I)` production rule matches an edge type `E` that connects a source node `s` of type `V` to a destination node `t` of type `I`, and contributes  $-var(t)/s.c$  term to the source node `s`’s dynamics<sup>2</sup>. The `V` node type implements a first-order differential equation with a summing reduction operator, so the  $-var(t)/s.c$  term is added to the derivative of  $s$ .

The TLN language definition includes validation rules that ensure the TLN computational model is faithfully implemented. The validation rule `cstr I` disallows connections between `I` and `I` node types and the validation rule `cstr V` disallows connections between `V` and `V` node types.

**Branched T-Line Function.** Figure 8 presents an Ark function

<sup>2</sup>`var(.)` is a convenient function that returns the state variable associated with the node.

that implements a programmable version of the branched t-line from Figure 2-(i). The `br-func` function accepts a `br` branch bit and enables the edge connecting the `IN_V` and `I_2` nodes together if the bit is set. Invoking the function `br-func` with `br=0` and returns the dynamical graph of a linear t-line (Figure 2-(ii)) and a branched t-line respectively. The body of the `br-func` function uses the TLN language to construct the branched t-line topology (`uses tln`). In the above function, all resistances and conductances are set to zero, all capacitances and inductances are set to  $1e-09$ , and the `InpI_0` is configured to provide a trapezoidal pulse function with width  $2e-8$  at time  $t=0$ .

#### 4.5. The GmC-TLN Language

Figure 9 presents the GmC-TLN language, which models the nonidealities of a mismatch-sensitive GmC network. The GmC-TLN language inherits all types and rules from the TLN language. All TLN computations are implementable in the GmC-TLN language and deliver the same dynamics.

The GmC-TLN language defines `Vm` and `Im` node types that inherit from the `V` and `I` node types and incorporate the effects of device mismatch on the  $C_{int}$  parameter in the corresponding GmC circuit. The `Vm` and `Im` node types override the `c` and `l` attributes to accept values between  $1e-10$  and  $1e-08$  and are subject to 10% mismatch. When a value  $v$  is written to an `Im` node’s mismatched attribute `c`, `c` is set to a value sampled from a normal distribution  $N(v, 0.1v)$ . Because the `Vm` and `Im` node types inherit from the `V` and `O` node types, they can be used anywhere a `V` node type was originally used.

The GmC-TLN language also defines a mismatched edge `Em` that both incorporates the effects of device mismatch on the  $gm_1$  and  $gm_2$  device parameters in the GmC circuit. The `Em` edge inherits from the `E` edge and defines 10% mismatched `ws` and `wt` attributes that map to the  $gm_1$  and  $gm_2$  device parameters. Because the GmC circuit dynamics change when  $gm_1 \neq gm_2$ , the GmC-TLN language defines new production rules for the `Em` edge. These production rules implement the modified Telegrapher’s equations presented in Section 2.3.

**Empirical Validation.** We randomly generate 1000 valid GmC-TLN DGs and generate SPICE netlists from these models with a simple algorithm. We observe (1) all valid DGs successfully map to a spice-level netlist, (2) the generated transient dynamics of the DG match the transient dynamics of the spice-level netlist within a root-mean-squared error of 1%. Therefore, empirically, the GmC-TLN language captures the dynamics of the spice-level circuit.

#### 4.6. The Ark Framework

Given an Ark program containing language and function definitions, an end user may invoke any of the defined functions with Ark. Ark executes the function with the provided arguments to build the associated dynamic graph and then validates that the dynamic graph satisfies the local and global validation rules in the associated language (Section 6). If the dynamic graph validates, Ark generates differential equations (Section 5) that

```

lang gmc-tln inherits tln{
  ntyp(1, sum) Vm inherit V
  {attr c=real[1e-10, 1e-08] mm(0, 0.1),
   attr g=real[0, inf]};
  ntyp(1, sum) Im inherit I
  {attr l=real[1e-10, 1e-08] mm(0, 0.1),
   attr r=real[0, inf]};
  etyp Em inherit E {attr ws=real[0.5, 2] mm(0, 0.1),
   attr wt=real[0.5, 2] mm(0, 0.1)
};
prod(e:Em, s:V->t:I) s<=-e.ws*var(t)/s.c;
prod(e:Em, s:V->t:I) t<=e.wt*var(s)/t.l; ... }

```

Figure 9: Gm-C-TLN language definition snippet.

---

### Algorithm 1 Differential Equation Compilation

---

**Input:**  $dg, langDef$

**Output:** A system of equations describes the  $dg$  dynamics

```

1: eqs ← []
2: eq.append(InitState(dg))
3: for n in Nodes(dg) do
4:   eqs.append(LowOrdEqs(langDef, n))
5:   rhs ← []
6:   for e in Edges(n) do
7:     rule ← LookupProdRule(lagnDef, n, e)
8:     expr ← Rewrite(rule, n, e)
9:     rhs.append(expr)
10:  eq ← FormEq(n, rhs)
11:  eqs.append(eq)
12: return eqs

```

---

simulate the transient behavior of the graph.

## 5. Ark Dynamical System Compiler

The Ark compiler processes a dynamical graph and a language definition as input and generate differential equations. For each node with order  $p$ , the compiler generate  $p$  differential equations with each state variable  $n_i, i = 1, \dots, p$  corresponding to the  $i^{\text{th}}$  derivative of the node  $n$ . The compilation process, outlined in Algorithm 1, proceeds as follows: First, the compiler creates all the necessary state variables with initial values specified in the dynamical graph. The `LowOrdEqs` function generates equations  $\frac{dn_i}{dt} = n_{i+1}, i = 1, \dots, p - 1$  which describe the node dynamics for orders less than  $p$ .

Subsequently, the compiler iterates through each edge associated with the nodes to determine the  $p^{\text{th}}$  derivative of the node. The `LookupProdRule` function examines the node and edge to find an associate production rule (`prod(v0:v1, v2:v3->v4:v5) ProdExpr`) in the language definition. Specifically, the edge's type determines  $v_1$ , and the edge's direction and the terminal nodes' types determine  $v_3$  and  $v_5$ , uniquely identifying the production rule to apply. If no rule matches immediately, the compiler will trace the inheritance relation and look up parent types recursively to find the closest production rule to apply. The `Rewrite` function retrieves the expression (`ProdExpr`) from the rule and returns new expression with the nodes and edges in the `ProdExpr`

---

### Algorithm 2 Pattern Matching

---

```

1: function ISDESCRIBED( $n, pattern$ )
2:   cstrs ← []
3:   vars ← ILPVars(len(edges) × len(pattern))
4:   edges ← EdgesOf( $n$ )
5:   for  $i, e$  in enumerate(edges) do
6:     for  $j, cls$  in enumerate(pattern) do
7:       if Matched( $n, e, cls$ ) then
8:         cstrs.append(ZeroOrOne(vars[i][j]))
9:       else
10:        cstrs.append(Zero(vars[i][j]))
11:   cstrs.append(UnityRowSum(vars))
12:   cstrs.append(RangedColSum(pattern, vars))
13:   return ILPSolve(cstrs)

```

---

substituted by the ones currently processed. The expressions of all edges of the node  $n$  are aggregated in `FormEq` using the reduction operator specified in the node type, yielding the equation representing the  $p^{\text{th}}$  derivative of the node. Following the procedure, the compiler returns differential equations  $eqs$ , which fully specify the computation of the given dynamical graph and can be used for transient simulation.

## 6. Ark Dynamical Graph Validator

The Ark validator takes input as a dynamical graph and a language definition, performing the validation required outlined in Section 4.1. The dynamical graph is provided as an input to the external function (`extern-func v`) specified in the language definition, which validates that the graph satisfies all global validity rules. The graph validates if the function `v` returns true. The validator verifies the local validity rules by iterating through all nodes and checks if they are described by at least one accepted pattern (`acc VMatch*`) and not described by any of the rejected patterns (`rej VMatch*`).

We formulate and solve the `described` relation as an Integer Linear Programming (ILP) problem in Algorithm 2. The algorithm accepts as input a node  $n$  and a pattern which is essentially a list of clauses (`VMatch*`), and returns whether the pattern describes the node. The procedure initializes an empty list of constraints  $cstrs$  and ILP variables  $vars$ . The variables denote the assignment of edges to clauses, meaning if  $vars[i][j] = 1$ , the  $i^{\text{th}}$  edge of the node  $n$  is assigned to the  $j^{\text{th}}$  clause of the pattern. Every edge-clause pair is inspected using the `Matched` function, which returns true if the edge is of edge type  $v_0$  connected to a node type contained in  $[v_1 *]$  specified in `VMatch`. If true,  $var[i][j]$  can be assigned to the clause, meaning it is constrained to be either 0 and 1;  $var[i][j]$  is constrained to equal 0 otherwise. The constraints specified in `UnityRowSum` ensure that each edge is assigned to only one clause, i.e.,  $\forall i. \sum_j var[i][j] = 1$ . The `RangeColSum` encodes the cardinality constraints specified with the `VAtom` terms hold for each clause, i.e.,  $\forall j. VAtom_{j,1} \leq \sum_i var[i][j] \leq VAtom_{j,2}$ . The constraints are passed to an ILP solver which returns true if a satisfying assignment is found and returns false otherwise.



## 7. Evaluation

We present two case studies where we formalize analog compute paradigms and the tradeoffs associated with different analog realizations with Ark. In each case, we analyze the effect of incorporating a subset of analog behaviors into the computation and formulate topological constraints that guide different analog design problems. Ark’s goal is to enable iterative co-design flow of domain-specific unconventional compute paradigms and analog circuits. Therefore, expressiveness and extensibility are the overarching objectives of the system.

### 7.1. Cellular Nonlinear Network (CNN)

The cellular nonlinear network [10] analog compute paradigm performs computation over a topology of locally interconnected cells, and has applications in image processing, pattern recognition, PDE solving, and security primitives [10, 8, 18, 13]. Researchers have previously developed analog accelerators that implement the CNN computational paradigm [12, 16]. The following differential equation describes the dynamics of a CNN with cells  $x_{ij}$ :

$$\frac{dx_{ij}}{dt} = -x_{ij} + \sum_{(k,l) \in N(i,j)} (A_{ij,kl} \cdot f(x_{kl}) + B_{ij,kl} \cdot u_{kl}) + z \quad (5)$$

Each cell  $x_{ij}$  and accepts an external input  $u_{ij}$ . The neighboring cells  $(k, l \in N(i, j))$  and associated external inputs flow into cell  $x_{ij}$ . The  $A_{ij,kl}$  and  $B_{ij,kl}$  matrices apply weights to the neighboring signals and external inputs, and the nonlinear activation function  $f$  transforms neighboring signals. Each  $x_{i,j}$  also has negative feedback and is subject to a constant bias  $z$ . The cell’s non-linear activation function  $f$  is typically a saturating function (blue line, Figure 11a).

**The CNN Language.** Figure 10a presents the Ark CNN DSL. The **V** and **Inp** node types map to the  $x_{i,j}$  and  $u_{i,j}$  variables, and the **Out** node type applies the nonlinearity dynamic **act** to an incoming  $x_{i,j}$  signal. The **iE** and **fE** edge types implement the CNN dynamics. The **v** node type defines the **z** parameter, and the **fE** edge type defines the **g** attribute which implements the  $A$  and  $B$  parameters.

**Hardware Extensions.** Figure 10b presents the **hw-cnn** extension to the CNN language that codifies the analog CNN design space and models circuit nonidealities.[17, 38] The **vm**, and **fEm** node types extend **v** and **fE** respectively and override the **g** and **z** attributes to incorporate mismatch in the hardware realization – this nonideality is reported to affect system convergence in analog implementations [17]. The **OutNL** node type inherits from the **Out** node type and applies a non-ideal saturation function **sat\_ni** with nonlinear dynamics near the saturation points (orange line, Figure 11a). These non-idealities arise because analog CNN realizations implement saturation with a MOS differential pair which introduces non-linearities due to the MOS transistors’ large signal behavior [38].

**Edge detection.** We implement an edge detector [9] in the **cnn** language and then use the **hw-cnn** language extension

```
lang cnn {
  ntyp(1, sum) V {attr z=real[-10,10]};
  ntyp(0, sum) Out {};
  ntyp(0, sum) Inp {};
  etyp iE {};
  etyp fE {attr g=real[-10,10]};
  prod(e:fE, s:Inp->t:V) t<=e.g*var(s);
  prod(e:iE, s:V->t:Out) t<=sat(var(s));
  prod(e:iE, s:V->s:V) s<=s.z-var(s);
  prod(e:fE, s:Out->t:V) t<=e.g*var(s);
  cstr V {acc[match(1,1,iE,V->[Out]),
    match(4,9,fE,[Out]->V),
    match(1,1,fE,V)]};
  cstr Out {acc[match(4,9,fE,Out->[V]),
    match(1,1,iE,[V]->Out)]};
  cstr Inp {acc[match(4,9,fE,Inp->[V])]} }
```

(a) CNN language.

```
lang hw-cnn inherits cnn{
  ntyp(0, sum) OutNL inherit Out {};
  ntyp(1, sum) Vm inherit V {attr z=real[-10,10],
  attr mm=real[1,1] mm(0,0.1)};
  etyp fEm inherit fE {attr g=real[-10,10] mm(0,0.1)};
  prod(e:fE, s:Inp->t:Vm) t<=e.g*t.mm*var(s);
  prod(e:iE, s:Vm->s:Vm) s<=s.mm*(s.z-var(s));
  prod(e:fE, s:Out->t:Vm)
  t<=e.g*t.mm*var(s);
  prod(e:iE, s:V->t:OutNL) t<=sat_ni(var(s)); }
```

(b) HW-CNN language.

Figure 10: Ark CNN language definition and extension.

to explore the effect of different analog non-idealities. The edge detector CNN is provided input image pixels as an external  $u_{i,j}$  input. The value of each cell  $x_{i,j}$  at steady state computes the output pixel  $p_{i,j}$ , which is black if an edge is detected. Figure 11b presents the input image and column A of Figure 11c presents the expected output image for the edge detector. Columns B-D present the CNN edge detector’s behavior with integrator bias (**z** mismatch), **g** parameter mismatch, and non-ideal saturation behavior. These behaviors are modeled by selectively substituting **v**, **Out**, and **fE** nodes in the original implementation with non-ideal nodes from **hw-cnn**. The rows of Figure 11c capture the evolution of time. **Analysis.** We observe all analog nonidealities substantially affect the transient dynamics of the edge detector. Notably, designs with mismatched **z** and **g** parameters (B, C) converged more slowly, where **g** mismatch also yielded an incorrect output image. The non-ideal saturation function (D) produced the correct result and actually *improved* convergence time, suggesting that some nonidealities have a potentially positive effect on the computation model.

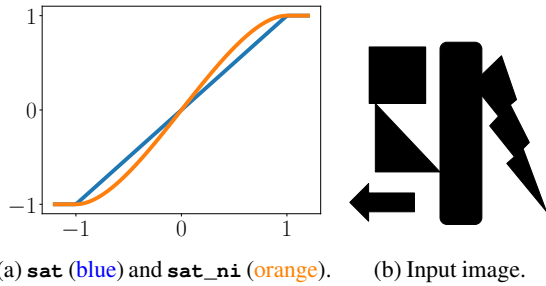
From this analysis, we can conclude (1) analog CNN designs should prioritize reducing **g** mismatch over **z** mismatch, (2) introducing certain nonidealities into the non-linearity function is acceptable, and in some case beneficial. The first point informs designers of where to expend effort to improve fidelity,

and the second point suggests a potential axis to explore in the CNN design space. Overall, this analysis underscores the importance of early-stage modeling of nonidealities in the design process, and how studying these nonidealities can aid in design-space exploration.

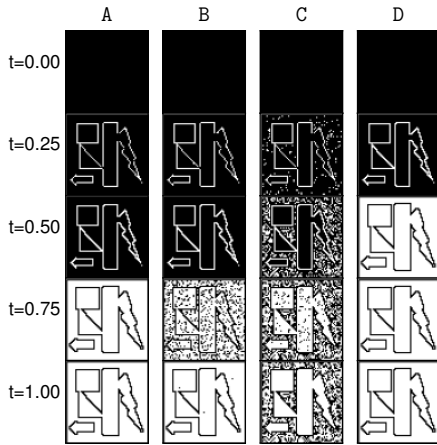
## 7.2. Oscillator-Based Computing (OBC)

Oscillator-based computing (OBC) is an emerging unconventional compute paradigm that has recently been used to solve max-cut [7, 36, 32] and graph coloring problems [32] and perform signal processing tasks like filtering and pattern recognition [44, 14]. In oscillator-based computing, a network of coupled oscillators performs computation. The coupling strength encodes the program inputs, and the synchronization behavior between oscillators implements the computation. The phase dynamics of the oscillators are described by the *modified Kuramoto model* [36]:

$$\frac{d\phi_i}{dt} = -C_1 \cdot \sum_{j=1}^n K_{ij} \cdot \sin(\phi_i - \phi_j) - C_2 \cdot \sin(2\phi_i) \quad (6)$$



(a) `sat` (blue) and `sat_ni` (orange). (b) Input image.



(c) Simulation results of an edge detection CNN with different types in `cnn` and `hw-cnn` language modeling hardware non-idealities. A: Ideal CNN. B: 10% random mismatch in integrator bias. C: 10% random mismatch in parameters. D: Non-ideal saturation function

Figure 11: Saturation functions and input image to CNN in the experiments and the observed transient dynamics under different non-ideal conditions.

```
lang obc {
  ntyp(1, sum) Osc {};
  etyp Cpl {attr k=real[-8,8]};
  prod(e:Cpl, s:Osc->t:Osc) s<=-1.6e9*e.k*sin(var(s)-var(t));
  prod(e:Cpl, s:Osc->t:Osc) t<=-1.6e9*e.k*sin(-var(s)+var(t));
  prod(e:Cpl, s:Osc->s:Osc) s<=-1e9*sin(2*var(s)); }
```

(a) OBC language.

```
lang ofs-obc inherits obc{
  etyp Cpl_ofs inherit Cpl
  {attr k=real[-8,8],
  attr offset=real[0,0] mm(0.02,0)
  };
  prod(e:Cpl_ofs, s:Osc->t:Osc)
  s<=-1.6e9*e.k*(e.offset+sin(var(s)-var(t)));
  prod(e:Cpl_ofs, s:Osc->t:Osc)
  t<=-1.6e9*e.k*(e.offset+sin(-var(s)+var(t))); }
```

(b) Ofs-OBC language.

Figure 12: Ark OBC language definition and extension.

$d$	obc		offset-obc	
	sync prob. (%)	slvd prob (%)	sync prob. (%)	slvd prob. (%)
$0.01\pi$	94.1	94.1	54.1	54.1
$0.1\pi$	94.2	94.1	94.8	94.6

Table 1: Probability of successful synchronization and solving max-cut problems with OBC and OBC with offset.

The  $\phi_i$  variable is the phase of oscillator  $i$ , and the  $K_{ij} \cdot \sin(\phi_i - \phi_j)$  term models the coupling between oscillators  $i$  and  $j$ , with  $K_{ij}$  denoting the coupling strength.  $C_1$  and  $C_2$  are constant scaling factors. We use  $1.6e9$  and  $1e9$  respectively in the evaluation.

**The OBC Language** Figure 12a defines an OBC language with Ark. The `Osc` node type and `Cpl` edge type models the oscillator phase  $\phi$  and the oscillator coupling behavior respectively. The `Cpl` defines an attribute `k` that maps to the coupling strength  $K$ . The production rules implement the modified Kuramoto model where the constants  $C_i$  and  $C_j$  are embedded in the expression.

**Integrator Bias Hardware Extension** Figure 12b presents the ofs-OBC hardware extension to the OBC language, which models the effect of analog integrator offset on an integrator-based OBC accelerator design. Prior work uses integrator and nonlinear conductors to implement the phase dynamics [36], which is possible to experience a non-zero offset when fed by currents emulated the coupling function. The `Cpl_ofs` node type inherits the `Cpl` node type and defines a mismatched offset attribute `offset` that biases the coupling terms. The ofs-con language defines new production rules that capture the effect of this offset nonideality on the system dynamics.

**Max-Cut Solver.** We implement a max-cut solver as a function using the OBC language, and then substitute `Cpl` with `Cpl_ofs` nodes to study the effect of integrator bias on the max-cut solution. We invoke the maxcut solver function over 1000 unweighted 4-vertex graphs, where the max-cut solver

```

lang intercon-obc inherits obc{
  ntyp(1,sum) Osc_G0 inherit Osc {};
  ntyp(1,sum) Osc_G1 inherit Osc {};
  etyp Cpl_1 inherit Cpl {attr k=real[-8,8],
    attr cost=int[1,1]};
  etyp Cpl_g inherit Cpl {attr k=real[-8,8],
    attr cost=int[10,10]};
  cstr Osc_G0 {acc[match(1,1,Cpl_1,Osc_G0),
    match(0,inf,Cpl_1,Osc_G0->[Osc_G0]),
    match(0,inf,Cpl_1,[Osc_G0]->Osc_G0),
    match(0,inf,Cpl_g,Osc_G0->[Osc]),
    match(0,inf,Cpl_g,[Osc]->Osc_G0)
  ]}
  cstr Osc_G1 {acc[match(1,1,Cpl_1,Osc_G1),
    match(0,inf,Cpl_1,Osc_G1->[Osc_G1]),
    match(0,inf,Cpl_1,[Osc_G1]->Osc_G1),
    match(0,inf,Cpl_g,Osc_G1->[Osc]),
    match(0,inf,Cpl_g,[Osc]->Osc_G1) ]} }

```

Figure 13: Intercon-OBC language definition.

maps input edges to coupling strengths and graph nodes to oscillators. For each graph, the oscillator phases are extracted from the max-cut solver simulation at steady-state. Oscillator nodes with phases that are within  $d$  radians of 0 and  $\pi$  are placed in partition 0 and partition 1, respectively; oscillators with other phases are marked unknown. The  $d$  deviation tolerance parameter is configurable and external to the analog circuit.

Table 1 summarizes the simulation results. For the **obc** max-cut solver, the correct partition is returned 94% of the time. Conversely, the max-cut solver with the integrator bias nonideality only successfully partitions graphs 54.1% percent of the time  $d$ . We study the dynamics of the nonideal max-cut solver and find the oscillator phase experiences slight jitter. We increase the phase tolerance from  $0.01\pi$  to  $0.1\pi$  to absorb this jitter and find the non-ideal max-cut solver attains 94% accuracy with this new parametrization. In this study, we used Ark to analyze analog non-idealities, which were then attenuated by applying a compensation technique external to the circuit. This mitigation approach allowed us to significantly improve the max-cut solver without actually improving the fidelity of the underlying circuit.

**Modeling Routing Tradeoffs.** Figure 14 presents the intercon-OBC extension to the OBC language, which captures the design tradeoffs associated with different kinds of coupled oscillator interconnect – a critical aspect in OBC design. The language lets end-users intermix *all-to-all* connections [32] and neighboring connections in the OBC computations. [5] All-to-all connections offer exceptional programmability but require significantly larger area as more programmable interconnect is required. Neighboring connections are less flexible but significantly more resource-efficient. The impact of this particular design decision is significant: the all-to-all chip [32] implemented 30 oscillators and devoted most area to routing circuitry, while the neighboring connection chip [5] implemented 560 oscillators with minimal routing circuitry. Both chips were fabricated with the CMOS 65nm technology and used 1.44 mm<sup>2</sup> chip size.

The intercon-OBC language formalizes the trade-off between programmability and resource utilization. This language

```

lang mm-tln inherits tln{
  ntyp(1,sum) Vm inherit IdealV
  {attr c=real[1e-10,1e-08] mm(0,0.1),
    attr g=real[0,inf]};
  ntyp(1,sum) Im inherit IdealI
  {attr l=real[1e-10,1e-08] mm(0,0.1),
    attr r=real[0,inf]};
  etyp Em inherit IdealE {attr ws=real[0.5,2] mm(0,0.1),
    attr wt=real[0.5,2] mm(0,0.1)
  };
  prod(e:Em,s:IdealV->t:IdealI) s<=-e.ws*var(t)/s.c;
  t<=e.wt*var(s)/t.l;
  prod(e:Em,s:IdealI->t:IdealV)
  s<=-e.ws*var(t)/s.l;
  prod(e:Em,s:IdealI->t:IdealV)
  t<=e.wt*var(s)/t.c;
  prod(e:Em,s:InpV->t:IdealV)
  t<=e.wt*(-var(t)+s.fn(times))/(s.r*t.c);
  prod(e:Em,s:InpV->t:IdealI)
  t<=e.wt*(-s.r*var(t)+s.fn(times))/t.l;
  prod(e:Em,s:InpI->t:IdealV)
  t<=e.wt*(-s.g*var(t)+s.fn(times))/t.c;
  prod(e:Em,s:InpI->t:IdealI)
  t<=e.wt*(-var(t)+s.fn(times))/(s.g*t.l);
}

```

Figure 14: Intercon-OBC language definition.

defines two edge types **Cp1\_1** and **Cp1\_g**, which implement edges for local and global connections, respectively. The **Cp1\_1** and **Cp1\_g** edges define a **cost** attribute, which specifies the resource cost of a connection. Local edges are assigned a lower cost than global edges. The OBC language defines validation rules that mandate connections across groups to be realized using **Cp1\_g** edges. These rules ensure the connectivity restrictions associated with local and global edge types are enforced at compile-time. With intercon-OBC, we can soundly architect global-local interconnect topologies that capture programmability/efficiency trade-off points in the design space.

## 8. Related Works

**Empirical Studies:** Researchers have characterized the impact of non-idealities on solution quality, convergence time, and stability for cellular nonlinear networks [17] and oscillator-based computing [44, 7, 5, 36]. These studies involve experts with a deep understanding of both the computing paradigm and analog circuitry and present the paper as a technical artifact. Our work complements this research and focuses on codifying both analog compute paradigms and the analog circuit constraints with a unified representation accessible to both parties, enabling collaboration between domain specialists and analog designers.

**Analog Models.** Analog behavioral modeling languages such as Verilog-A and Verilog-AMS [37, 31, 46, 39] and Analog MacroModels [45, 15, 6] are typically used to model analog behavior for system-level design. These models are usually constructed from a transistor-level circuit design and apply simplifications or elide certain analog behaviors (e.g., transient behavior) to improve the performance of the model and focus

on the modeling of classical circuits for circuit designer’s use. In contrast, our method targets early design-space exploration for the co-design of novel compute paradigms and analog circuits, which require accessible models for non-circuit domain specialists and support of iterative modeling over circuit specification before a transistor-level circuit is presented.

**Structured Math Abstractions.** Many fields use structured math representations of continuous-time systems, such as signal-flow graphs and block diagrams [23, 28, 43]. These representations compose basic math elements (e.g., multiplication, filtering) together to construct a higher-level computation. These approaches are hardware agnostic and do not support easy incorporation of analog design constraints and nonidealities. In contrast, Ark captures both analog design tradeoffs and restrictions and models the overall dynamics of the continuous-time system.

**Analog Computer Specifications.** Compiler-writers have also developed analog hardware specification languages to capture the capabilities of GPAC analog computing platforms [4, 2, 3]. These languages cannot effectively model analog compute paradigms with interacting elements (e.g., OBC, CNN models), and rely on algorithms to implement certain hardware constraints (e.g., current fanout). Our language can support a range of compute models, including compute models with interacting elements, and supports the specification of the aforementioned interconnect restrictions.

## 9. Conclusion

Reconfigurable analog circuits are promising substrates for unconventional compute paradigms tailored to specific domains. We present the Ark programming language for specifying these novel compute paradigms while incorporating analog constraints and trade-off space. Ark enables progressive modeling of analog behaviors with computations and provides a validator and dynamical system compiler for verifying and simulating the computations. We describe three unconventional computing paradigms with Ark and demonstrate that Ark aids design space exploration and co-design of the computation with the analog circuits. Therefore, Ark takes a key step towards an agile design of novel compute paradigms, enhancing collaboration among experts from diverse backgrounds, and thereby stimulating innovation.

## Acknowledgement

We would like to thank Boris Murmann and Luke Sammarone for their input and expertise during development of the Ark language. This work is supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number DE-FOA-0002950 and the Stanford Graduate Fellowship.

## References

[1] The decadal plan for semiconductors, a pivotal roadmap outlining research priorities. <https://www.src.org/about/decadal-plan>.

Accessed: 2022-11-10.

[2] Sara Achour and Martin Rinard. Time dilation and contraction for programmable analog devices with jaunt. *ACM SIGPLAN Notices*, 53(2):229–242, 2018.

[3] Sara Achour and Martin Rinard. Noise-aware dynamical system compilation for analog devices with legno. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 149–166, 2020.

[4] Sara Achour, Rahul Sarpeshkar, and Martin C Rinard. Configuration synthesis for programmable analog devices with arco. *ACM SIGPLAN Notices*, 51(6):177–193, 2016.

[5] Ibrahim Ahmed, Po-Wei Chiu, William Moy, and Chris H Kim. A probabilistic compute fabric based on coupled ring oscillators for solving combinatorial optimization problems. *IEEE Journal of Solid-State Circuits*, 56(9):2870–2880, 2021.

[6] Tutu Ajayi, Sumanth Kamineni, Yaswanth K Cherivirala, Morteza Fayazi, Kyumin Kwon, Mehdi Saligane, Shourya Gupta, Chien-Hen Chen, Dennis Sylvester, David Blaauw, et al. An open-source framework for autonomous SoC design with analog block generation. In *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*, pages 141–146. IEEE, 2020.

[7] Jeffrey Chou, Suraj Bramhavar, Siddhartha Ghosh, and William Herzog. Analog coupled oscillator based weighted ising machine. *Scientific reports*, 9(1):14786, 2019.

[8] Leon O Chua, Martin Hasler, George S Moschytz, and Jacques Neirynck. Autonomous cellular neural networks: a unified paradigm for pattern formation and active wave propagation. *IEEE Transactions on Circuits and Systems I: Fundamental theory and applications*, 42(10):559–577, 1995.

[9] Leon O Chua and Lin Yang. Cellular neural networks: Applications. *IEEE Transactions on circuits and systems*, 35(10):1273–1290, 1988.

[10] Leon O Chua and Lin Yang. Cellular neural networks: Theory. *IEEE Transactions on circuits and systems*, 35(10):1257–1272, 1988.

[11] Glenn ER Cowan, Robert C Melville, and Yannis P Tsividis. A VLSI analog computer/digital computer accelerator. *IEEE Journal of Solid-State Circuits*, 41(1):42–53, 2005.

[12] JM Cruz and LO Chua. A  $16 \times 16$  cellular neural network universal chip: The first complete single-chip dynamic computer array with distributed memory and with gray-scale input-output. *Analog Integrated Circuits and Signal Processing*, 15:227–237, 1998.

[13] György Csaba, Xueming Ju, Qingqing Chen, Wolfgang Porod, Jürgen Schmidhuber, Ulf Schlichtmann, Paolo Lugli, and Ulrich Rührmair. On-chip electric waves: An analog circuit approach to physical uncloneable functions. *Cryptology ePrint Archive*, 2009.

[14] György Csaba and Wolfgang Porod. Coupled oscillators for computing: A review and perspective. *Applied physics reviews*, 7(1):011302, 2020.

[15] Fernando De Bernardinis, Pierluigi Nuzzo, and A Sangiovanni Vincentelli. Mixed signal design space exploration through analog platforms. In *Proceedings of the 42nd annual Design Automation Conference*, pages 875–880, 2005.

[16] Shukai Duan, Xiaofang Hu, Zhekang Dong, Lidan Wang, and Pinaki Mazumder. Memristor-based cellular nonlinear/neural network: design, analysis, and applications. *IEEE transactions on neural networks and learning systems*, 26(6):1202–1213, 2014.

[17] Jorge Fernández-Berni and Ricardo Carmona-Galán. On the implementation of linear diffusion in transconductance-based cellular nonlinear networks. *International Journal of Circuit Theory and Applications*, 37(4):543–567, 2009.

[18] Luigi Fortuna, Paolo Arena, David Balya, and Akos Zarandy. Cellular neural networks: a paradigm for nonlinear spatio-temporal processing. *IEEE Circuits and Systems magazine*, 1(4):6–21, 2001.

[19] Daibashish Gangopadhyay, Emily G Allstot, Anna MR Dixon, Karthik Natarajan, Subhanshu Gupta, and David J Allstot. Compressed sensing analog front-end for bio-sensor applications. *IEEE Journal of Solid-State Circuits*, 49(2):426–438, 2014.

[20] Samanwoy Ghosh-Dastidar and Hojjat Adeli. Spiking neural networks. *International journal of neural systems*, 19(04):295–308, 2009.

[21] Ning Guo, Yipeng Huang, Tao Mai, Sharvil Patil, Chi Cao, Mingoo Seok, Simha Sethumadhavan, and Yannis Tsividis. Energy-efficient hybrid analog/digital approximate computation in continuous time. *IEEE Journal of Solid-State Circuits*, 51(7):1514–1524, 2016.

[22] Charles Herder, Meng-Day Yu, Farinaz Koushanfar, and Srinivas Devadas. Physical uncloneable functions and applications: A tutorial. *Proceedings of the IEEE*, 102(8):1126–1141, 2014.

[23] Joachim Holtz. The representation of ac machine dynamics by complex signal flow graphs. *IEEE transactions on industrial electronics*, 42(3):263–271, 1995.

- [24] Yipeng Huang, Ning Guo, Mingoo Seok, Yannis Tsvividis, Kyle Mandli, and Simha Sethumadhavan. Hybrid analog-digital solution of nonlinear partial differential equations. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 665–678. IEEE, 2017.
- [25] Umran S Inan, Aziz S Inan, and Ryan K Said. Engineering electromagnetics and waves. (*No Title*), 2016.
- [26] Houk Jang, Chengye Liu, Henry Hinton, Min-Hyun Lee, Haeryong Kim, Minsu Seol, Hyeon-Jin Shin, Seongjun Park, and Donhee Ham. An atomically thin optoelectronic machine vision processor. *Advanced Materials*, 32(36):2002431, 2020.
- [27] Haideh Khorramabadi and Paul R Gray. High-frequency cmos continuous-time filters. *IEEE Journal of Solid-State Circuits*, 19(6):939–948, 1984.
- [28] Wing-Hung Ki. Signal flow graph in loop gain analysis of dc-dc pwm ccm switching converters. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 45(6):644–655, 1998.
- [29] Saikrishna Reddy Konatham, Reza Maram, Luis Romero Cortés, Jun Ho Chang, Leslie Rusch, Sophie LaRochelle, Hugues Guillet de Chatellus, and José Azaña. Real-time gap-free dynamic waveform spectral analysis with nanosecond resolutions through analog signal processing. *Nature communications*, 11(1):1–12, 2020.
- [30] Scott Koziol, Paul Hasler, and Mike Stilman. Robot path planning using field programmable analog arrays. In *2012 IEEE international conference on robotics and automation*, pages 1747–1752. IEEE, 2012.
- [31] Sabrina Liao and Mark Horowitz. A verilog piecewise-linear analog behavior model for mixed-signal validation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(8):2229–2235, 2014.
- [32] Antik Mallick, Mohammad Khairul Bashar, Daniel S Truesdell, Benton H Calhoun, Siddharth Joshi, and Nikhil Shukla. Graph coloring using coupled oscillator-based dynamical systems. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2021.
- [33] Adnan Mehonic, Abu Sebastian, Bipin Rajendran, Osvaldo Simeone, Eleni Vasilaki, and Anthony J Kenyon. Memristors—from in-memory computing, deep learning acceleration, and spiking neural networks to the future of neuromorphic and bio-inspired computing. *Advanced Intelligent Systems*, 2(11):2000085, 2020.
- [34] Boris Murmann, Marian Verhelst, and Yiannos Manoli. Analog-to-information conversion. *NANO-CHIPS 2030: On-Chip AI for an Efficient Data-Driven World*, pages 275–292, 2020.
- [35] Robert B Northrop. *Analysis and application of analog electronic circuits to biomedical instrumentation*. CRC press, 2003.
- [36] Karlheinz Ochs, Bakr Al Beattie, and Sebastian Jenderny. An ising machine solving max-cut problems based on the circuit synthesis of the phase dynamics of a modified kuramoto model. In *2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 982–985. IEEE, 2021.
- [37] François Pêcheux, Christophe Lallement, and Alain Vachoux. Vhdl-ams and verilog-ams as alternative hardware description languages for efficient modeling of multidiscipline systems. *IEEE transactions on Computer-Aided design of integrated Circuits and Systems*, 24(2):204–225, 2005.
- [38] Behzad Razavi. *Design of Analog CMOS Integrated Circuits*. McGraw-Hill, second edition, 2005.
- [39] Jinsoo Rhim, Yoojin Ban, Byung-Min Yu, Jeong-Min Lee, and Woo-Young Choi. Verilog-a behavioral model for resonance-modulated silicon micro-ring modulator. *Optics express*, 23(7):8762–8772, 2015.
- [40] Jussi Ryynanen, Kalle Kivekas, Jarkko Jussila, Aarno Parssinen, and Kari Al Halonen. A dual-band rf front-end for wcdma and gsm applications. *IEEE Journal of Solid-State Circuits*, 36(8):1198–1204, 2001.
- [41] Abu Sebastian, Manuel Le Gallo, Riduan Khaddam-Aljameh, and Evangelos Eleftheriou. Memory devices and applications for in-memory computing. *Nature nanotechnology*, 15(7):529–544, 2020.
- [42] Yannis Tsvividis. Not your father’s analog computer. *IEEE Spectrum*, 55(2):38–43, 2018.
- [43] Mummadi Veerachary. General rules for signal flow graph modeling and analysis of dc-dc converters. *IEEE transactions on Aerospace and Electronic Systems*, 40(1):259–271, 2004.
- [44] Damir Vodencarevic, Nicolas Locatelli, Flavio Abreu Araujo, Julie Grollier, and Damien Querlioz. A nanotechnology-ready computing scheme based on a weakly coupled oscillator network. *Scientific reports*, 7(1):44772, 2017.
- [45] Jian Wang, Xin Li, and Lawrence T Pileggi. Parameterized macromodeling for analog system-level design exploration. In *Proceedings of the 44th annual Design Automation Conference*, pages 940–943, 2007.
- [46] Yi Wang, Yikai Wang, and Lenian He. Behavioral modeling for operational amplifier in sigma-delta modulators with verilog-a. In *APCCAS 2008-2008 IEEE Asia Pacific Conference on Circuits and Systems*, pages 1612–1615. IEEE, 2008.
- [47] Alex Yakovlev and Victor Pacheco-Peña. Enabling high-speed computing with electromagnetic pulse switching. *Advanced Materials Technologies*, 5(12):2000796, 2020.