Noise-Aware Dynamical System Compilation for Analog Devices with Legno

Sara Achour
MIT EECS & CSAIL
sachour@csail.mit.edu

Martin Rinard
MIT EECS & CSAIL
rinard@lcs.mit.edu

Abstract

Reconfigurable analog devices are a powerful new computing substrate especially appropriate for executing computationally intensive dynamical system computations in an energy efficient manner. We present Legno, a compilation toolchain for programmable analog devices. Legno targets the HCDCv2, a programmable analog device designed to execute general nonlinear dynamical systems. To the best of our knowledge, Legno is the first compiler to successfully target a physical (as opposed to simulated) programmable analog device for dynamical systems and this paper is the first to present experimental results for any compiled computation executing on any physical programmable analog device of this class. The Legno compiler synthesizes analog circuits from parametric and specialized blocks and account for analog noise, quantization error, and manufacturing variations within the device. We evaluate the compiled configurations on the Sendyne S100Asy RevU development board on twelve benchmarks from physics, controls, and biology. Our results show that Legno produces accurate computations on the analog device. The computations execute in 0.50-5.92 ms and consume 0.28-5.67 μJ of energy.

CCS Concepts. • **Computer systems organization → Analog computers**; • **Software and its engineering → Compilers**.

Keywords. Compilers, Analog Computing, Languages

ACM Reference Format:
Sara Achour and Martin Rinard. 2020. 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 (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3373376.3378449

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03.
https://doi.org/10.1145/3373376.3378449

1 Introduction

Programmable analog devices are a promising new class of low-power computing substrates well suited for executing complex dynamical systems [4, 7, 8, 32, 34, 37, 40, 41, 45]. Dynamical systems are used to model a variety of physical processes from domains such as biology, chemistry, and physics. Applications include dosage prediction, optimization, and stability for these processes. Dynamical systems may also be used to implement filters, control systems, and state estimators over external signals.

In contrast to digital systems, which simulate dynamical systems by discretizing time, analog devices execute dynamical systems continuously, using the physics of the device to model the behavior of the dynamical system. This execution model eliminates time scale issues digital devices suffer from and enables the analog hardware to execute nonlinear dynamical systems using micro-joules of energy. Some analog devices are able to seamlessly process external analog inputs (such as sensor inputs). Our target devices are digitally programmable with digitally settable connections and digitally configurable analog computing blocks.

1.1 Legno

We present Legno, a new compiler for programmable analog devices. Legno targets the HCDCv2 [15, 18, 43], a programmable analog device designed to execute general nonlinear dynamical systems. To the best of our knowledge, Legno is the first compiler to successfully target a physical (as opposed to simulated) programmable analog device for dynamical systems and this paper is the first to present experimental results for any compiled computation executing on any physical programmable analog device of this class.

Given a dynamical system and target analog device specification, Legno automatically configures the device to execute the dynamical system. Our target class of devices exhibit several properties that shape the compilation:

- **Computation with Physical Signals**: Analog devices, including the HCDCv2, exploit the device physics to implement computation directly using physical signals such as analog currents. While this direct computation is the key to the energy efficiency of analog devices, it also requires computations to operate successfully in the presence of challenging physical phenomena such as noise, quantization error, frequency and operating range limitations, and manufacturing variations.

Legno manages these issues by scaling the computation to respect the physical limitations of the device and compensate for variations within the device while delivering acceptably accurate computations in the presence of noise.

- **Parametric Blocks**: The HCDCv2 features digitally configurable analog computation blocks, with different configurations implementing different analog functions. Such *parametric blocks* enhance the programmability of the device and enable it to support a wider range of computations with a given number of blocks.
  To exploit the full capabilities of parametric blocks, Legno deploys a staged compilation algorithm that first partially configures blocks to select the desired computation operation (such as multiply or integrate), then completes the configuration when scaling to ensure the final configuration satisfies the physical constraints of the device.
- **Leveraging Properties of Physical Signals:** To support efficient analog computations that leverage physical properties of the signals on the device, Legno generates configurations that add signals by connecting the wires that carry these signals in the HCDCv2.
- **Specialized Blocks**: Physical signals often have properties (signals represented with currents, for example, cannot be routed to multiple input ports) that constrain their use within the device. The HCDCv2 therefore provides specialized blocks that copy and route signals globally within the device. Legno produces configurations that incorporate these specialized blocks as necessary to map the computation onto the device.
- **Locally Prioritized Interconnectivity**: Complete interconnectivity is impractical in programmable analog devices. The HCDCv2 prioritizes local interconnectivity, specifically by providing denser interconnectivity between colocated analog blocks. To effectively utilize the device, Legno therefore incorporates a *spatially aware* placement algorithm that places interconnected blocks close to each other in the device.

Other Devices: In addition to the HCDCv2, previous programmable analog devices share these properties [41, 45]. Given the engineering tradeoffs that shape the design space, these properties are also likely to appear in future devices — noise and other physical phenomena are inherent in energy-efficient computing with physical devices, leveraging the physical properties of signals is one of the keys to efficient analog computing, parametric blocks effectively support programmability which maximizing potential utilization of hardware resources, and sparse interconnectivity is a fact of life given routing constraints on modern hardware platforms. We therefore anticipate the basic Legno compilation approach will generalize to include future programmable analog hardware platforms.

1.2 Contributions

This paper presents the following contributions:

- **Languages:** It presents the Legno analog device specification language, dynamical system specification language, and analog device program language. These languages specify the target analog device, the input dynamical system, and the configuration to write to the analog device.
- **Compiler:** It presents the Legno compiler, a compilation toolchain for programmable analog devices.
- **LGraph:** It presents the LGraph graph synthesis engine. LGraph synthesizes an analog device program that implements the input dynamical system. LGraph adopts a staged synthesis pipeline (generation, assembly, and routing) that inserts relevant specialized blocks at each stage and configures the parametric behavior of the used blocks to implement the desired functionality.
- **LScale:** It presents the LScale scaling engine. Given an analog device program and an analog and digital quality measure, LScale completes the configuration of the parametric behavior of the blocks and scales the signals and values in the program. The emitted analog device program respects the frequency and current range limitations of the device block, ensures that the noise and quantization errors fall within the provided measures, and compensates for manufacturing variations found in the device. The experimental results show that LScale delivers realistic scaling transforms that produce good quality computations on our target physical analog device.
- **Experimental Results:** It presents, to the best of our knowledge, the first empirical evaluation of an analog compilation toolchain on a physical programmable pure analog device for solving differential equations.

Broader Implications: Driven by trends such as the need for increased performance, reduced energy consumption, and growing markets for focused classes of computations, specialized but still programmable computing platforms will play an important role in the future of computing. These platforms will offer unprecedented combinations of energy efficiency and performance, but at the cost of increasingly challenging programming models. Reconfigurable analog devices comprise one compelling example of this class of platforms. Legno highlights the important role that new compilation technology and, more generally, new software toolchains, can play in enabling developers to successfully exploit the potential of these promising platforms.

2 Previous Compilers for Analog Devices

Researchers have previously developed software tooling and compilation techniques for mapping general-purpose computations to programmable analog neural network architectures [5, 21, 27, 35, 36, 40]. Our research, in contrast, targets programmable analog devices for dynamical systems.

category	feature	Arco	Jaunt	Legno
block	complex	✓	✓	
	parametric			✓
	special-use	△	△	✓
device	current-mode	△	△	✓
	voltage-mode	△	△	
	spatial layout	△	△	✓
property	operating range		✓	✓
	frequency		△	✓
	noise			✓
	quantization error			✓
	manufacturing deviation			✓

Figure 1. Comparison of Legno with prior approaches. ✓: full support. △: partial support.

Arco [2] and Jaunt [1] both target simulated programmable analog devices for dynamical systems. Legno, in contrast, targets a physical analog device and must effectively deal with all of the physical phenomena (noise, frequency, and operating range constraints, quantization errors, manufacturing variations) present in physical analog devices. Figure 1 presents a comparison of the architectural features and properties supported by Legno compared to Jaunt and Arco.

Block Features: Legno provides both language and compiler support for parametric, routing, and copy blocks, including separating blocks by purpose and exploiting specialized blocks in dedicated compilation stages. Arco and Jaunt, in contrast, do not support parametric blocks in their device description languages and compilers, do not distinguish routing or copy blocks in the compilation, and therefore do not support the effective utilization of parametric, routing, or copy blocks. Legno uses a pattern-based unification scheme which works best with blocks that implement simple functions. Legno can easily be adapted to use a different unification algorithm, such as the one deployed by Arco.

Device Features: Legno natively supports analog currents, including implementing addition by connecting wires carrying the signals to add and introducing blocks to copy currents as necessary. While both Arco and Jaunt can target voltage- and current-mode devices, they do not implement specialized mechanisms for performing computation with currents. To our knowledge, analog voltages require less specialized handling than currents — voltage-mode devices implement computation with dedicated circuits and voltages may be used multiple times without the aid of a copier.

To support devices that prioritize local connections, Legno deploys a block placement algorithm that colocates connected blocks. Arco and Jaunt, in contrast, use a layout-agnostic placement procedure that does not produce efficient placements for devices that prioritize local connections.

3 Languages

The Legno compiler works with a dynamical system specification (**DSS**) and an analog device specification (**ADS**). The

```
body(<rule>) ::= <rule> | seq<rule>;<rule>
seq(<rule>) ::= <rule> | seq<rule>,<rule>
tup(<rule>) ::= (seq<rule>)
lst(<rule>) ::= [seq<rule>]
pat(<rule>) ::= | <rule> | pat(<rule>) | <r1>
match(<r1>,<r2>) ::= (pat(<r1>) -> <r2>)*
multi(<r1>,<r2>) ::= <r2> | func match(<r1>,<r2>)
```

Figure 2. Shortcuts and macros for grammars

```
x ∈ RealNumbers, v ∈ Literals
n ∈ NaturalNumbers,
x+ ∈ RealNumbers ≥ 0,
G ::= sgn | ln | exp | cos | sin  | abs
H ::= min | max | pow
I  ::= [x,x]
E  ::= E1 + E2 | E1*E2 | x | v
     | integ(E1,E2) | (E) | call(v,lst(E))
F  ::= E | F1/F2 | G (F) | H (F1, F2)
```

Figure 3. Math expressions

```
Stmt ::= var v = E | emit E as v | extern v
| interval seq(v) = I | func v(seq(v)) = F | time x
Prog ::= prog v { body(Stmt)}
```

Figure 4. Dynamical system specification language

```
prog cosc {
  var v = integ(-0.22*v - 0.84*p, -2.0);
  var p = integ(1*v, 9.0);
  interval p,v = [-15,15];
  emit p as pos; time 20; }
```

Figure 5. DSS for dampened oscillator

Legno compiler produces, as output, an analog device program (**ADP**) that specifies a configuration of the analog device that executes the dynamical system. The analog device program is, itself, not directly executable. It is instead converted to a low-level executable script, written in the Grendel scripting language (Section 3.4) for execution on the target device.

3.1 Dynamical System Specification Language

Figure 4 presents the dynamical system specification language (DSSL). The language supports binding symbolic expressions to dynamical system variables (`var v = E`) and defining named functions (`func v = F`) that the `call(v,E1, ..., En)` construct can invoke. Every statement of the form `var v = E` defines a *relation* in the specification. Figure 3 presents the mathematical operators for expressions and function bodies. An `integ(E1,E2)` expression denotes the integral of E_1 with respect to time given an initial value E_2. Function bodies (F) support a richer set of mathematical operations than dynamical system expressions (E). `emit` statements observe variables and expressions in the dynamical system. `extern` statements provide external signals to the dynamical system.

Each variable in the specification must be annotated with an interval bound (`interval` statements). Before compilation, an interval propagation algorithm computes bounds for all variables in the system. A well-formedness check ensures

that all non-external variables have defined behavior and can be bounded. The **time** statement describes how much time (in simulation units) to run the computation for.

Dampened Oscillator: Figure 5 presents the dynamical system specification (DSS) for a dampened oscillator. This system models the position p and velocity v of a dampened spring oscillator over time starting with an initial position of 9 cm and an initial velocity of -2 cm/s. The DSS (Figure 5) implements the following differential equations:

$$\dot{v} = -0.22 \cdot v - 0.84 \cdot p \quad \dot{p} = v \quad v(0) = -2 \quad p(0) = 9$$

The **var** statements declare and define the dynamics of the p and v variables; the **interval** statements annotate each variable with the range of values it may take on. The **v** and **p** variables are annotated to be between [-15,15] m/s and [-15,15] m respectively. The **pos** variable corresponds to the observation of the position **p** over time. The oscillator executes for twenty simulation units.

3.2 Analog Device Specification Language

The analog device specification, written in the analog device specification language (ADSL), comprises a collection of block specifications, available connections, and the layout of the target analog device.

3.2.1 Block Specification Language: Figure 6 presents the block specification language. Each named block either computes (**compute**), copies signals (**copy**), or routes signals (**route**) through the chip. Each block has a set of associated input (**in**) and output (**out**) ports. Each port is either an **analog** or a **digital** port. Analog ports use analog currents to represent values in the computation. Analog currents are added together by joining wires and cannot be used more than once without the aid of a copier block.

Ports may be externally accessible (**extern**). External input ports accept externally provided signals. External output ports may be observed with an external measurement device such as an oscilloscope. Each block may also have a collection of digitally settable data fields (**data**) that can be statically set by the compiler. The data fields may be constant values (**const**) or expressions (**expr**). Each block has a set of digitally programmable *modes*. Each mode is a tuple of literals, which together set the behavior of the block.

Figure 7 presents the block specification for an analog multiplier. The **type** clause identifies the multiplier as a **compute** block. The **modes** clause specifies eleven modes, **(m,m,m)** through **(x,h,h)**. The block has two analog inputs (**x** and **y**), one constant data field (**c**), and one analog output (**z**). The **rel** statement specifies the *basic expression* of the analog output port **z** as a function of the block mode.

When the block mode matches the pattern **(x,*,*)**, the block multiplies the analog input signal **x** by the digital parameter **c** so that the basic expression of **z** is **c*x**. Otherwise, the block multiplies the two analog input ports together so that the basic expression of **z** is **x*y**.

```
Mode = tuple(1), ModeR = tuple(1|*)
BlockT ::= compute | copy | route
SigT ::= analog | digital
DataT ::= const | expr vars seq(v)
PortT ::= in | out
QuantT ::= linear d
IFace ::= PortT seq(v) sigT (extern)?
        | data v DataT
Impl   ::= rel v = multi(ModeR,E)
        | coeff v = multi(ModeR,x+)
        | interval seq(v) = multi(ModeR,I)
        | quantize v = multi(ModeR,QuantT)
        | maxfreq v = multi(ModeR,n)
Def ::= block v BlockT modes lst(Mode)
Stmt ::= IFace | Impl
Spec ::= Def {body(Stmt)}
```

Figure 6. Block specification language

```
1  block mult type compute modes [(m,m,m),
2      (m,m,h),(h,m,h),(m,h,h),(h,h,h),
3     ,(x,m,m),(x,m,h),(x,h,m),(x,h,h)] {
4     in x,y analog; out z analog;
5     data c const;
6     rel z = func |(x,*,*) -> c*x
7          |(*,*,*) -> x*y;
8     coeff z = func |(m,m,h) -> 5,
9          |(m,m,m)|(h,m,h)|(m,h,h) -> 0.5
10         |(h,m,m)|(m,h,m)|(h,h,h) -> 0.05
11         |(x,m,h)->10 |(x,h,m) -> 0.1
12         |(x,h,h)|(x,m,m) -> 1;
13     quantize c = linear 256;
14     interval c = [-1,1];
15     interval z = func |(*,*,m) -> [-2,2]
16                  |(*,*,h) -> [-20,20];
17     interval x = func |(*,m,*) -> [-2,2]
18                  |(*,h,*) -> [-20,20];
19     interval y = func
20                  |(h,*,*) -> [-20,20]
21                  |(*,*,*) -> [-2,2]; }
```

Figure 7. Block specification for multiplier

Blocks can also specify a positive, constant coefficient for each port. The **coeff** statement in Figure 7 defines the coefficient of port **z** as a function of the block mode. The value of an output port **p** that implements basic expression **E** is the basic expression with the port coefficients applied (**coeff(p)E[v⇒coeff(v)v]**). For example, with mode **(m,m,m)**, the value of **z** is **0.5*x*y**; with mode **(x,m,h)**, the value of **z** is **10*c*x**.

Each port also has an *operating range*. Each port's value must fall within the port's operating range for the block to function correctly. The **interval** statements in Figure 7 define the operating ranges for ports **z**, **x**, and **y** as a function of the block mode. For example, the value of port **z** must remain between **-2** and **2** μA for modes matching **(*,*,m)** and between **-20** and **20** μA for modes matching **(*,*,h)**. The **maxfreq** statement (not in Figure 7) identifies the maximum frequency supported by the block ports.

```
Loc = v tuple(n), AbsLoc = v tuple(l|n)
RegLoc = v tuple(l|v|_);
Ports = lst(v) @ RegLoc (port v)?
Stmt ::= loc lst(n) |block lst(v) @ Loc
       | for AbsLoc blk lst(v) @ AbsLoc
       | view v (in v)? | freq n
       | (for AbsLoc)? conn Ports with Ports
Spec ::= device v {block(Stmt)}
```

Figure 8. Device specification language

```
1  device hcdcv2 {
2   view chip; view tile in chip;
3   view slice in tile; view idx in tile;
4   loc 0,1 in chip; loc 0,1,2,3 in tile;
5   loc 0,1,2,3 in slice; loc 0,1,2,3 in idx;
6   for slice(x,y,z) blk int,mul,fan@ idx(x,y,z,0);
7   for slice(x,y,z) blk mul,fan@ idx(x,y,z,1);
8   for idx(x,y,z,w) blk tout,tin@ idx(x,y,z,w);
9   blk cout @ idx(0,3,2,0); freq 126000;
10  for tile(x,y)
11    conn int,mul,fan,tin @ idx(x,y,_,_)
12    with int,mul,fan,tout @ idx(x,y,_,_);
13  for chip(x) conn tout @ idx(x,_,_,_)
14    with tin @ idx(x,_,_,_,_);
15  conn tout @ idx(0,_,_,_)
16    with cout @ idx(0,3,2,0); }
```

Figure 9. Example device specification

Digital ports are quantized into a finite set of values as specified by their corresponding `quantize` and `interval` statements. The block defined in Figure 7, for example, quantizes **c** into **256** digital values between **-1** and **1** (with the values spaced 0.0071825 apart).

`interval` statements are required for all ports and data, `quantize` statements are required only for digital ports, and `maxfreq` statements are optional. Together, the `interval`, `maxfreq`, and `quantize` statements specify the *physical limitations* of the block, i.e., the range and frequency limitations and quantization effects.

In comparison with `compute` blocks, `copy` and `route` blocks are specialized blocks with additional constraints. `copy` blocks may only copy or negate signals with only unity (**-1** or **1**) coefficients. `route` blocks have a single mode, only one input port and one output port, and only unity coefficients. `route` blocks exist to enable successful signal routing in the presence of constraints on connections between output and input ports from different blocks.

3.2.2 Device Specification Language: Figure 8 presents the device specification language, which identifies the blocks and connections available on the analog device. `blk` statements attach blocks in the device to locations (**Loc**). A block at a location is called a *block instance*. The block name and location uniquely identify each block instance. `conn` statements identify settable connections. The hardware time constant as specified by the `freq` statement defines the baseline integration speed of the device.

```
Loc = v tuple(n), Port = v.v  @ Loc
Assigns = seq(seq(v) = x)
CStmt ::= set v = x | expr v = E
        | set v vars lst(v) = F
        | scale Assigns | coeff Assigns
Stmt ::= config block v @ Loc { body(CStmt) }
       | conn Port with Port | timescale x
Prog ::= body(Stmt)
```

Figure 10. Analog device program language

```
timescale 40000;
conn mul.z @ idx(0,3,0,0) to int.x @ idx(0,3,0,0);
conn mul.z @ idx(0,3,0,1) to int.x @ idx(0,3,0,0);
```

Figure 11. Example connections and time constant

The location of a block instance is a named tuple of numbers (**v tuple(n)**). The locations in the device are divided into sequentially organized views. The *name* of the location (**v**) is the device `view` the location belongs to. Every location in a particular view also belongs to a location in its parent view. Block instances may only be bound to locations from the most specific view (the leaf view).

Figure 9 presents an example device specification. The device has four views (chip, tile, slice, and index). There are two chips (line 4), 4 tiles per chip (line 4), 4 slices per tile (line 5), and 4 indices per slice (line 5). Each block location identifies the chip, tile, slice, and index of the block. For example, a **mul** block at **idx(0,3,2,1)** is on chip **0**, tile **3**, slice **2**, and index **1** and therefore also belongs to **slice(0,3,2)**, **tile(0,3)**, and **chip(0)**.

This specification attaches one integrator (**int**), two multipliers (**mul**), two copiers (**fan**), and eight routing blocks (**tin** and **tout**) per slice (lines 6-8). The device has exactly one external output (**cout**), which resides on chip **0**, tile **3**, slice **2**, and index **0**. All blocks may be connected to all other blocks within a tile (lines 10-12), but routing blocks (**tin** and **tout**) must be used to connect blocks belonging to different tiles (lines 13-15). In the above specification, the baseline time constant is 126000 hz – so that one unit of integrator time corresponds to 7.93μs of wall clock time.

3.3 Analog Device Program Language

An analog device program (ADP) specifies a configuration of the analog device as generated by the Legno compiler. The ADP is itself not directly executable on the device — it is instead translated to a physical device configuration and loaded into the device via the Grendel scripting language. Figure 10 presents the analog device program language.

Device Configuration: A device configuration specifies the digitally settable connections to enable and the time scale of the computation. Figure 11 presents an excerpt of a device configuration. `conn` statements specify the connections to enable — each `conn` statement identifies an input port and an output port to connect. The **timescale** statement specifies the time scale of the computation.

```
config block mult @ idx(0,3,0,0) {
   set c=0.22; expr z=c*x; }
```

Figure 12. Basic configuration for multiplier

```
config block mult @ idx(0,3,0,0) {
   set c=0.22; expr z=c*x;
   mode=(x,m,m);
   scale x=0.063,c=4.2,z=0.1999; }
```

Figure 13. Full configuration for multiplier

Block Configurations: A *basic block configuration* specifies the name and location of the block, values for constants, and basic expressions for output ports. Figure 12 presents a basic configuration for a multiply block from Figure 7. Here the block name is **`mult`**, the block location is **`idx(0,3,0,0)`**, the value of **`c`** is **`0.22`**, and the basic expression for **`z`** is **`c*x`**.

A *full block configuration* also specifies the mode, which determines the full expression for each port, and a scaling transform, which scales the computation to ensure it respects the physical constraints of the device. Figure 13 presents a full configuration for a multiply block from Figure 7. In addition to setting **`c`** to **`0.22`** and the basic expression for **`z`** to **`c*x`**, the full configuration also sets the mode to **`(x,m,m)`** and defines scaling factors for the ports and data fields.

To reconstruct the dynamics of **`z`** in the units of the original dynamical system, take the observed signal from the computation, divide it by **`0.1999`** to invert the scaling transform applied to **`z`**, then multiply by the time constant **`40000`** from the device configuration (Figure 11).

3.4 Grendel Scripting Language

Grendel is an executable, low-level scripting language used for programming and executing experiments on the HCDCv2. It is able to calibrate and profile blocks, configure the analog chip, retrieve debugging information, and set up external measurement devices. The Grendel source generator generates Grendel scripts from ADPs, which contain all the information necessary to configure the analog device.

4 Overview of the Legno Compiler

Figure 14 presents a high-level overview of the Legno compiler. The **Dynamical System Specification** (DSS) and **Analog Device Specification** (ADS) respectively specify the input dynamical system and the available programmable blocks and connections on the target analog device.

The **Empirical Model Database** is a repository of empirically derived models that describe how each individual block instance deviates from the block specification in the ADS. These models specify the empirically measured gain and noise for each data field and port in the block instance. The *gain* is the ratio of the measured value to the expected value. The *noise* is the standard deviation of the measured value. Figure 15 presents an example empirical model for the multiplier at **`idx(0,3,0,0)`**.

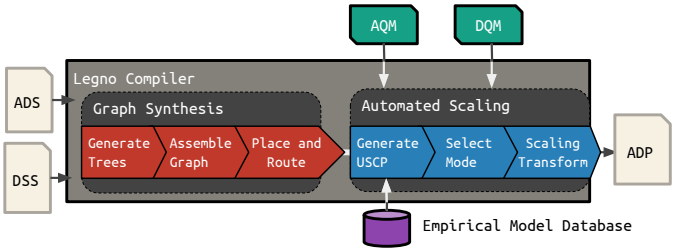


Figure 14. Overview of the Legno Compiler

```
{block:'mult', loc:'idx(0,3,0,0)',
(x,m,m): {z:{gain:0.755781,noise:0.01},
          c:{gain:1.0,noise:0.0},
          x:{gain:1.0,noise:0.0}},
(x,m,h): {z:{gain:0.660414,noise:0.1},
          ...}, ...}
```

Figure 15. Empirical model for multiplier at **`idx(0,3,0,0)`**

The ***Analog Quality Measure*** (AQM) is a parameter that specifies the minimum allowed signal-to-noise ratio for the analog signals in the ADP. For constant analog signals, the AQM specifies the upper bound for the ratio of the noise to the magnitude of the signal (**`noise/|signal|` ≤ `AQM`**). For dynamic analog signals, the AQM specifies the upper bound for the ratio of the noise to the dynamic range of the signal (**`noise/(max(signal)-min(signal))` ≤ `AQM`**).

The ***Digital Quality Measure*** (DQM) is a parameter that specifies the minimum allowed signal-to-noise ratio for the digital signals in the ADP. For constant digital signals, the DQM specifies the upper bound for the ratio of the error to the magnitude of the signal (**`error/|signal|` ≤ `DQM`**). For dynamic digital signals, the DQM specifies the upper bound for the ratio of the error to the dynamic range of the signal (**`error/(max(signal)-min(signal))` ≤ `DQM`**).

Compilation Choice Points: The Legno compiler encounters various **nondeterministic** choice points during the compilation process. Different decisions at these choice points typically produce different analog device programs (ADPs). Legno has the capability to enumerate the full range of decisions at each choice point and to therefore enumerate the full range of ADPs for the input DSS. Our current Legno implementation uses iterated random sampling at each choice point to generate a subset of the full range of ADPs possible at that point. It is possible for a phase of the compiler to fail for one or more of the current ADPs. In this case Legno continues on with the compilation of the remaining ADPs.

Here we present the compilation process for a single ADP, identifying nondeterministic choice points as they are encountered during compilation. The two primary phases are:

LGraph [Section 5]: LGraph synthesizes an ADP that implements the input DSS. The ADP contains connections and basic block configurations. Recall a basic block configuration assigns basic expressions to output ports and values/expressions to data fields, but does not include scaling information.

LScale [Section 6]: LScale completes the configurations in the input ADP by setting the scale factors and completing the mode in each block instance configuration. It also sets the time constant of the ADP. LScale derives a universal scaling constraint problem (USCP) from the input ADP, the analog and digital quality measures (AQM and DQM), and the empirical model database. Legno solves the USCP to select the modes and scaling transform for the ADP. The scaling transform accounts for the range, frequency, and quality constraints imposed by the ADS, compensates for manufacturing deviations, and ensures that the original dynamics of the input DSS can be recovered by scaling each signal by statically derived coefficients.

5 LGraph

LGraph first generates a subcircuit for each relation in the DSS (relation subcircuit generation, Section 5.2). Recall that a relation is a statement of the form `var v = e`. It then assembles these relation subcircuits into a circuit that implements the DSS, inserting copy blocks as necessary (assembly, Section 5.3). Finally, it assigns blocks to locations on the device, inserting route blocks as necessary (place and route, Section 5.4). Internally, LGraph works with a flexible representation of the ADP called the VADP (Section 5.1).

5.1 Virtual Analog Device Program (VADP)

Figure 16 presents an example VADP that implements the relation `var c=(0.1*(-a))*b`. The VADP identifies block instances with unique integer *identifiers* which are replaced with locations during the place and route stage. The VADP supports subcircuits with input *sources* and output *sinks*. Sources indicate where positive or negative DSS variables are needed in the circuit. Sinks map output ports to DSS variables. Both sources and sinks are resolved to output ports during the assembly stage.

5.2 Relation Subcircuit Generation

For each relation `var v=e` in the DSS, the relation subcircuit generation procedure produces a subcircuit (implemented as a VADP) that implements that relation. Figure 18a presents an example synthesized subcircuit for the relation `v = integ(-0.22*v-0.84*p,-2)`.

Algorithm: The **gen** algorithm (Figure 17) produces a VADP that implements the desired relation. Inputs to the algorithm include the VADP to populate, the expression to implement, and a *destination* (input) port or sink. The algorithm populates the VADP with the basic block instance configurations and connections necessary to implement the provided expression. It then connects the port carrying that expression to the provided destination. For a relation `var v = e`, LGraph invokes the **gen** algorithm with a new VADP, an expression **e**, and the destination **sink(v)**. We next describe the functions invoked by the **gen** algorithm.

```
config mult @ 1 {set c=0.1;expr z=c*x;}
config mult @ 2 {expr z=x*y;}
conn source(a,-) with mult.x @ 1
conn mult.z @ 1 with mult.x @ 2
conn source(b,+) @ 1 with mult.x @ 2
conn mult.z @ 2 with sink(c)
```

Figure 16. Example VADP implementing **c=0.1*(-a)*b**

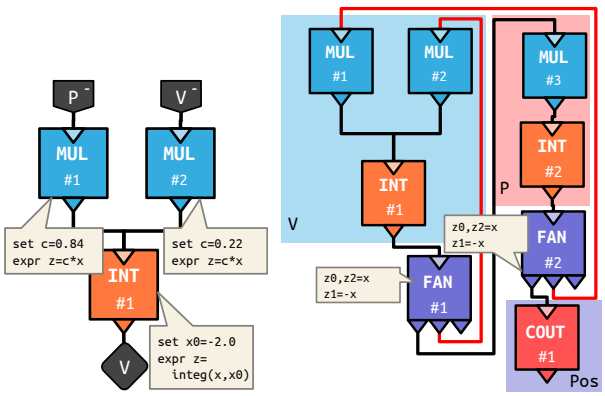
```
1  function gen(vadp,expr,dest):
2   match expr with
3   | v -> vadp.connect(source(v,+),dest)
4   | -1*v -> vadp.connect(source(v,-),dest)
5   | e1+e2 -> gen(vadp,e1,dest);gen(vadp,e2,dest)
6   | e -> blk,idx = select_block(e)
7     op,cfg,assigns=unify(blk,idx,e)
8     vadp.add(cfg);vadp.conn(port(blk,idx,op),dest)
9     for ip,ie in assigns:
10      gen(vadp,ie,port(blk,idx,ip))
```

Figure 17. gen algorithm

unify(blk,idx,expr): The unification function **nondeterministically** unifies the input expression against one of the relations in the block specification. It accepts a block instance and expression, and returns the selected output port, the computed basic configuration, and a set of input port-expression assignments. LGraph performs unification using pattern matching, but other algorithms could be used.
select_block*(e)*: The selection function **nondeterministically** selects a compute block from the ADS that is unifiable (can be unified with the expression) and creates a fresh identifier for that block. It returns the name and identifier of the chosen block instance. If none of the compute blocks are unifiable, the compilation process fails.

We next describe the basic operation of the **gen** algorithm:

Insertion of Source Nodes [Lines 3-4]: **gen** maps positive and negative variables to positive and negative sources.
Addition with Kirchhoff's Law [Line 5]: **gen** leverages *Kirchhoff's Law* to add variables together, as current-mode analog devices do not provide compute blocks that perform addition. Kirchhoff's law states that at any point in a circuit, the sum of incoming currents equals the sum of outgoing currents. **gen** connects the synthesized subcircuits that implement **e1** and **e2** to destination port/sink **dest**. Because the currents carrying **e1** and **e2** both flow into **dest**, the current flowing out of dest equals **e1+e2**.
Synthesis of Block Configurations [Lines 6-10]: For the remaining expressions, **gen** uses block selection and unification to implement the provided expression with a compute block instance. It first selects a compute block instance and unifies that block with the provided expression (Line 6-7). **gen** adds the returned basic configuration to the VADP and connects the unified output port to the provided destination (Line 8). **gen** then generates the rest of the subcircuit by recursively synthesizing each input port-expression assignment returned by the **unify** function (Line 9-10).



(a) generation **(b)** assembly

Figure 18. Relation subcircuit generation and assembly for dampened oscillator.

5.3 Assembly

LGraph next assembles the collection of subcircuits implementing the individual relations from the DSS into a circuit that implements the input DSS. It accepts, as input, the set of VADPs produced in the relation subcircuit generation stage, and produces, a output, a VADP of the assembled circuit.

Example: Figure 18b presents the assembled circuit for the dampened oscillator (see Section3.1). It introduces two current copiers (**FAN**) configured to produce one positive and one negative signal. These positive **(black)** and negative **(red)** signals are connected to the appropriate input ports.

Algorithm: The assembly algorithm matches the sources found in each VADP to ports that implement the desired variable with the desired sign. Because analog currents cannot be used more than once, the algorithm inserts copy blocks when necessary to replicate signals. LGraph first collates all the VADPs and counts all the positive and negative sources for each variable. If only one copy of a variable is needed, it directly connects the source and sink together. If more than one copy of a variable is needed, it **nondeterministically** builds a tree of copy blocks and configures the copy blocks (with basic configurations) to produce the required number of positive and negative signals. These signals are then connected to the appropriate sources. After all the sources are matched to the appropriate ports, the source and sink nodes are eliminated from the VADP.

5.4 Place and Route

Given a VADP that implements the assembled circuit and an ADS, LGraph assigns block instances to locations (placement) and maps connections to paths resident on the analog device, inserting route blocks as necessary (route). It produces, as output, an ADP comprised of basic configurations.

The placement algorithm (Section 5.4.3) incrementally resolves the location of each block instance by successively solving a sequence of *view placement problems* (VPP).

This decomposition reduces the complexity and size of the placement problem, enabling LGraph to tractably arrive at a satisfying set of location assignments. Because VPPs also account for the device's routing restrictions, LGraph is able to use a lightweight routing algorithm (Section 5.4) to assign VADP connections to paths on the analog device.

5.4.1 View Placement Problem (VPP): The view placement problem (VPP) is an integer linear programming problem that assigns VADP block instances to locations in a target view, subject to a set of restricting location assignments. Recall views are sequentially organized groups of locations that correspond to spatial regions on the device (Section 3.2.2). The VPP **nondeterministically** generates location assignments that respect the block instance and connectivity limitations imposed by the ADS. For each block instance with a restricting location assignment, the VPP ensures the assigned location is a child of the restricting location. The VPP contains two types of binary membership variables [38]:

Instance variables [`inst(block,id,loc)`] assign VADP block instances to locations in the view. LGraph derives location assignments from enabled (set to 1) instance variables.

Path variables [`path(conn,loc`$_1$`,loc`$_2$`,path_id)`] assign VADP connections to distinct paths comprised of route blocks and digitally programmable connections. The VPP only models paths between distinct locations because analog devices typically provide fewer connections between spatially distant blocks (distinct locations) than spatially colocated blocks (same location). This simplification reduces the complexity of the VPP while producing likely routable placements.

5.4.2 VPP Generation: The VPP generation algorithm accepts an input VADP, an ADS, and a set of restricting assignments and generates a VPP that, when solved, produces location assignments with the properties described in Section 5.4.1. It generates the following constraints:

Block Location Assignment: Each block instance in the VADP is assigned to exactly one instance.

Example: The sum of instance variables for multiplier #1 must equal one: $[\sum_{x,y}$ `inst(mul,1,tile(x,y)) = 1`$]$

Block Availability: The number of block instances mapped to each location does not exceed the number of available blocks of that type at that location.

Example: **`tile(0,1)`** has eight multipliers. Only eight multiplier instance variables for **`tile(0,1)`** may be enabled: $[\sum_i$ `inst(mul,i,tile(0,1))` $\leq$ `8`$]$

Path Assignment: If two connected blocks are assigned to different locations, than the connection must be mapped to a path between the two locations.

Example: If multiplier #1 is mapped to **`tile(0,1)`** and integrator #1 is connected to multiplier #1 (with connection **`c`**) and mapped to **`tile(0,0)`**, exactly one path variable assigning **`c`** to a path between those tiles is enabled: $[$`inst(mul,1,tile(0,1))` $\wedge$ `inst(int,1,tile(0,0)) =` $\sum_i$ `path(c, tile(0,1),tile(0,0),i)`$]$

```
timescale 40000;
config block mult @ idx(0,3,0,0) {
   set c=0.22;
   expr = c*x;
   mode = (x,m,m);
   scale x=0.063;
   scale c=4.2;
   scale z=0.1999;
}
```

Figure 19. Excerpt of full configuration

Path Restrictions: The number of enabled paths that share resources (route blocks) must not exceed the number of distinct paths supported by these resources.
Example: **tile(0,0)** requires outgoing connections use one of 16 route blocks. **tile(0,0)** therefore supports a maximum of sixteen distinct outgoing paths:
$[\sum_{j,i,x,y}$ **path(i,tile(0,0),tile(x,y),j)** $\leq$ **16**$]$
Restricting Locations: Each block instance may only be assigned to a child of its assigned restricting location.
Example: If multiplier #1 is restricted to location **chip(0)**, it is only mapped to locations matching **tile(0,**$*$**)**:
$[\sum_{x}$ **inst(mul,1,tile(1,x)) = 0**$]$

5.4.3 Placement Algorithm: The placement algorithm accepts a VADP and ADS and produces a set of location assignments that spatially colocate densely connected blocks. This is desirable as analog device microarchitectures typically prioritize providing programmable connections between blocks that are close together. The algorithm produces assignments by incrementally refining the spatial location of each block instance. It first solves the VPP for the most general (root) view to attain an initial set of restricting assignments. It then solves the VPP for each child view, using the parent view assignments as restricting assignments. If no solution is found, the algorithm backtracks. Generally, backtracking is not necessary as early placement decisions assign blocks to larger, more sparsely connected device structures. However backtracking may occur more often for VADPs that exhaust hardware resources.

5.4.4 Routing Algorithm: The routing algorithm maps connections in the VADP to paths in the ADS. It assigns a candidate path that implements each connection in the VADP. After each path assignment, it removes any intersecting paths from the set of candidate paths. If the algorithm encounters a connection with no candidate paths, it backtracks to an earlier assignment. This lightweight algorithm is often able to find satisfying routing solutions because the VPPs solved during placement restrict the selection of paths that share resources.

6 LScale

LScale computes a set of mode assignments and a scaling transform (Section 6.1) that completes the input ADP. It takes as input the analog device and dynamical system specifications (ADS and DSS), an input ADP, quality measure parameters (AQM and DQM), and an empirical model database. LScale derives a universal scaling constraint problem (USCP) from the program inputs and solves the USCP to acquire the mode assignments and scaling transform for the ADP. The completed ADP delivers two assurances:

- **Recoverable**: The original dynamics of the input DSS can be recovered from any ADP port by multiplying the time and magnitude of the observed signal by statically derived constants.
- **Physically Sound**: The ADP respects the operating range and frequency restrictions imposed by the ADS, meets the quality requirements imposed by the AQM and DQM, and compensates for the manufacturing variations described in the empirical model database.

6.1 Scaling Transform

The scaling transform is comprised of a collection of scaling factor assignments for the ports and data fields in the ADP (**scale v = x** or **sv(v)**) and a time constant describing the speed of the simulation (**timeconstant x** or **tc**). The scaling transform is applied by multiplying each data field **d** in the ADP by its associated scaling factor **sv(d)**. The original dynamics of each port **p** is recovered at runtime by multiplying wall-clock time by the time constant **tc** and dividing the signal amplitude by its associated scaling factor **sv(p)**. The original dynamical system is *recoverable* from the scaled ADP if the following property holds:
Recoverability Property: For each output port **op** in the ADS, the implemented *physical expression* equals its basic expression times the port scaling factor **sv(op)**. Each *physical expression* specifies the signal value at the corresponding port during execution of the ADP on the physical device. Refer to Section 6.3.1 for information on how to derive the physical expression of a port.

Example: Figure 19 presents an excerpt of a complete ADP. The transform is applied to the configured multiplier by setting **c** to **0.22*4.2**. The original dynamics of the signal at port **z** is recovered by dividing the magnitude of **z** by **0.1999** and multiplying wall-clock time by **40000**. This scaling transform ensures the physical expression of port **z** equals its basic expression (**0.22*x**) times its scaling factor (**0.1999**). This can be verified by first deriving output port **z**'s physical expression from the block configuration, block specification (Section 3.2.1), the associated empirical model (Section 4) then factoring out **0.1999** from the physical expression:

0.755781(4.2*0.22)(0.063*x) = 0.1999(0.22 x)

This property ensures the original dynamics of **z** (**0.22*x**) can be recovered by dividing the scaled signal at **z** by **0.1999**.

variable	type	scope	descriptor
mode	integer	block	`mode(block)`
property	real	port/data	$\texttt{name}_\texttt{p}\texttt{(port\|data)}$
scale factor	real	port/data	`sv(port|data)`
time constant	real	global	`tc`
injection	real	expr in/out	`iv(arg)`

Figure 20. Summary of USCP variables created from ADP.

property	from	description
$\text{opRange}_\text{p}(\text{v})$[1]	ADS	operating range restriction
$\text{maxFreq}_\text{p}(\text{v})$	ADS	maximum frequency restriction
$\text{digError}_\text{p}(\text{v})$	ADS	quantization error
$\text{coeff}_\text{p}(\text{v})$	ADS	constant coefficient
$\text{noise}_\text{p}(\text{v})$	model-db	noise
$\text{gain}_\text{p}(\text{v})$	model-db	measured gain

Figure 21. Summary of USCP properties for the port/data field v. Implemented as two properties[1].

6.2 The USCP

The USCP is a constraint problem whose solution produces a set of mode assignments and a scaling transform with the properties listed in Section 6. The USCP is comprised of a mixture of geometric programming and SMT constraints over positive integer-valued and positive real-valued variables. The USCP supports all geometric programming constraints and a restricted subset of SMT constraints.

Variables: Figure 20 lists the USCP variables. Each type of USCP variable is either globally defined or created for each block, port, or data field instance in the ADP (*scope* column). The mode variables store the selected modes. The time constant and scale factor variables store the time constant and scale factors from the ADP scaling transform. The property variables capture the mode-dependent behavior of the device and resolve to constant values when the mode variables are instantiated.

Injection Variables: LScale modifies the ADP expression data field assignments to more flexibly scale the circuit. For each ADP statement that sets data field **d** to expression **F** ($\texttt{set d vars [a}_0\texttt{...a}_n\texttt{] = F}$), LScale introduces injection variables $\texttt{iv(d)}$, $\texttt{iv(a}_0\texttt{)}$...$\texttt{iv(a}_n\texttt{)}$ which change the stored expression to $\texttt{iv(d)F[a}_i \Rightarrow \texttt{iv(a}_i\texttt{)a}_i\texttt{]}$.

Properties: Table 21 lists the USCP properties. LScale directly retrieves the **opRange**, **maxFreq**, and **coeff** properties from the ADS and the **gain** and **noise** properties from the empirical model database. LScale computes the maximum quantization error (**digError**) properties from the **quantize** statements in the ADS. For example, the maximum quantization error for an eight bit linearly encoded digital value between [-1,1] is **(1−(−1))/256 = 0.0078125**.

Intervals: LScale automatically derives the interval bounds of each signal in the ADP by propagating the intervals from the DSS through the circuit. LScale uses the derived intervals to generate operating range and quality constraints.

6.3 USCP Generation

LScale generates the USCP by producing the following constraints for each block instance and port instance in the input ADP. Each constraint is paired with an example that references the multiplier block specified in Section 3.2.1:

Viable Mode Selection: Each mode variable may only be assigned to the subset of modes that implement the selected basic expressions in the configured block instance. The value of a particular mode is the array index of that mode in the block specification.

Example: The mode variable for a multiplier instance **m** that implements the basic expression **c*x** must be set to either **(x,m,m)**, **(x,m,h)**, **(x,h,m)** or **(x,h,h)**:

$\texttt{[mode(m)} \in \texttt{[5..8]]}$

Mode-Property Relationship: For each property, the USCP must encode the relationship between the property value and the mode of the block instance as a set of implication constraints.

Example: If the mode of multiplier instance **m** is set to **(x,m,h)**, the property $\texttt{coeff}_\texttt{p}\texttt{(z)}$ is 6:

$\texttt{[mode(m) = 6} \rightarrow \texttt{coeff}_\texttt{p}\texttt{(z) = 10]}$

Operating Range Limitations: For each port, the dynamic range of the scaled signal must be contained by the operating range of that port.

Example: For port instance **z** with interval bound [-3.3,3.3], the dynamic range of the scaled signal must fall within the port's operating range:

$\texttt{[sv(z) [-3.3,3.3]} \subseteq \texttt{opRange}_\texttt{p}\texttt{(z)]}$

Analog Quality Restrictions: For each analog port, the ratio of the noise at that port to the dynamic range (if time-varying) or magnitude (if constant) of the scaled signal must be smaller than the AQM.

Example: For port instance **z** with interval bound [-3.3,3.3], the noise-to-dynamic range ratio is less than the AQM:

$\texttt{[sv(c)}^{-1}\ \texttt{6.6}^{-1}\ \texttt{noise}_\texttt{p}\texttt{(z)} \leq \texttt{AQM]}$

Digital Quality Restrictions: For each digital port, the ratio of the quantization error to the dynamic range (if time-varying) or magnitude (if constant) of the scaled signal must be smaller than the DQM.

Example: For constant data field instance **c** that provides value 0.22, the error-to-magnitude ratio is less than the DQM:

$\texttt{[sv(c)}^{-1}\texttt{0.22}^{-1}\texttt{digError}_\texttt{p}\texttt{(c)} \leq \texttt{DQM]}$

Frequency Limitations: For each port, the time constant of the ADP must be lower than the maximum frequency of that port.

Example: The time constant cannot exceed the defined maximum frequency for port instance **z**:

$\texttt{[[tc} \leq \texttt{maxFreq}_\texttt{p}\texttt{(z)]]}$

Connectivity: The scaling factors for each pair of connected ports must be equal. This constraint automatically ensures scaled analog currents are added properly.

Example: If a port instance **z** is connected to a port instance **x**, then $\texttt{[sv(z) = sv(x)]}$.

phys expr	new cstrs	rewritten expr
q(v) sv(v) v		q(v) sv(v) v
u e + u' e'	u = u'	u (e + e')
(u e)(u' e')		u u' (e e')
integ_{hw}(ue,u'e')	u tc hw^{-1} = u'	u' integ_{tc}(e,e')
call(v,[u_0e_0,..])	iv(a_i)u_i = 1	iv(v)call(v,[e_0,..])
sv(v)v = q(v)(ue)	sv(v) =q(v) u	v = e

Figure 22. Recoverability constraint generation rules. The red terms are USCP expressions. **q(v) = coeff_p(v)gain_p(v)**. **hw** is baseline integrator speed.

6.3.1 Recoverability Constraint Generation: LScale generates recoverability constraints which ensure the original DSS dynamics can be recovered from the completed ADP. These constraints ensure that each physical expression is equivalent to the associated basic expression multiplied by a USCP expression. LScale derives a physical expression from the basic expression of each ADP output port instance **expr op = E** using the following transformation:

sv(v) op =
  coeff_p(op)gain_p(op) E[v⇒coeff_p(v)gain_p(v)sv(v)v]

This transformation introduces the relevant mode-dependent coefficients, empirically derived gains, and scaling factors into the basic expression. LScale then iteratively applies the rewrite rules defined in Figure 22 to each derived physical expression until all the USCP variables have been eliminated. LScale adds any constraints generated by the rewrite rules to the USCP. The rewrite rules perform the following operations. The rules presented in rows 1-4 and 6 were adapted from prior work [1]:

Factoring (rows 1-4): LScale factors out a USCP expression (red) from the physical expression, adding constraints over USCP expressions as necessary.
Time Scaling (row 4): LScale ensures all integration operations are scaled by the same relative time constant (**tc** $\mathbf{hw}^{-1}$). This changes the speed of the simulation from the baseline speed (**hw**) to the scaled speed (**tc**). The ADS specifies the baseline integration speed (**hw**) of the analog device.
Expression Data Field Mutation (row 5): For a **call** operation that invokes the expression stored in data field **v** with arguments $\mathbf{e_0}..\mathbf{e_n}$, LScale ensures the data field's injection variables cancel out the arguments' USCP expressions. The injection variable **iv(v)** scales the returned value.
USCP Variable Elimination (row 6): LScale eliminates USCP variables from the rewritten relation.

6.3.2 Quality Guarantees: LScale does not provide any guarantees on the end-to-end result of the ADP or guarantee a minimum signal-to-noise ratio for any of the time-varying digital and analog signals. While the introduced AQM and DQM constraints affect the quality of the end-to-end result, they confer no guarantees. We note that it is highly nontrivial to provide a static error bound for a nonlinear dynamical system given some perturbation (e.g. noise).

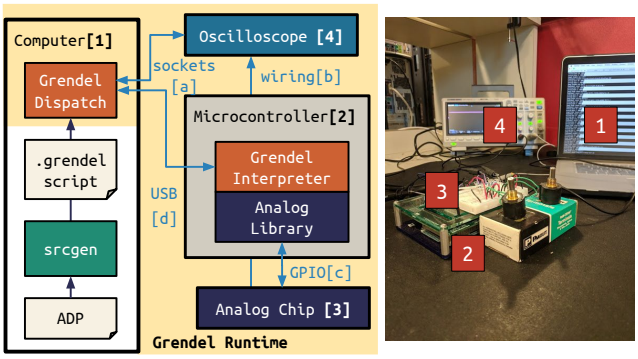


Figure 23. Grendel runtime workflow and lab setup

6.4 Completing the ADP

LScale solves the USCP to complete the input ADP.
Mode Selection: LScale first solves the USCP with an SMT solver which **nondeterministically** produces a set of mode assignments. In practice, LScale is always able to quickly find a satisfying set of mode assignments or obtain a proof that the USCP is infeasible. In the latter case, the ADP is unscalable and LScale fails to complete these ADPs.
Scale Transform Generation: LScale first applies the computed mode assignments to the USCP. This procedure resolves all USCP property variables to constants and eliminates all SMT constraints. LScale then uses a convex solver to quickly find the best scaling transform with respect to an optimization function. LScale's optimization function maximizes the speed (time constant **tc**) of the scaled ADP. LScale also applies the set of injected variable assignments to the expression data fields in the ADP.

7 Runtime and Implementation

Figure 23 presents the workflow for configuring the HCDCv2 to execute an input ADP. We first use the Grendel script generator **srcgen** to translate the ADP into a Grendel script. **srcgen** applies the scaling transform to the ADP, then translates the ADP statements to Grendel commands. The Grendel runtime then executes the produced Grendel script on the HCDCv2 analog device. After execution, we invert the ADP scaling transform at each collected waveform to recover the original DSS dynamics.
Grendel Runtime: A dispatcher and interpreter comprise the Grendel runtime. The dispatcher routes Grendel script commands to the appropriate devices. It sends HCDCv2 configuration commands the microcontroller, which runs a bare-metal interpreter that applies commands to the connected HCDCv2. The dispatcher sends measurement commands to the oscilloscope which records the desired external signals (**extern** ports) and returns the collected waveforms.
Implementation: The Legno compiler uses the **PULP** ILP solver, the **Z3** SMT solver, the **gpkit** geometric program solver with the **cvxopt** backend, and the **networkx** graph processing library for compilation [6, 9, 16, 25, 44].

8 Results

We evaluate Legno on twelve benchmarks, six of which were previously hand-implemented by our collaborators and six of which are novel or from prior work [1, 2]. Unless otherwise stated, each benchmark executes for 20 simulation units. We present the number of differential equations, number of functions, and system type (linear (**lin**) or nonlinear (**nl**)) in brackets after each benchmark:

cosc[2,1,**lin**]: The dampened spring oscillator discussed in Section 3. The position of the oscillator is measured.

cos[2,1,**lin**]: The oscillator that implements the cosine function. The oscillator amplitude is measured.

vander[2,1,**nl**]: A stiff vanderpol oscillator that executes for 50 simulation units. The oscillator amplitude is measured.

forced[4,1,**nl**]: A forced vanderpol oscillator with an internally generated input for 50 simulation units. The oscillator amplitude is measured.

pid[4,4,**lin**]: A robotics control system that adjusts the acceleration of a body to meet a target velocity in the presence of a large, oscillating disturbance for 200 simulation units. The body's velocity is measured.

pend[2,2,**nl**]: A pendulum simulation without the small-angle approximation. The mass's position is measured.

spring[4,3,**nl**]: A physics simulation of two masses connected by a spring. The first mass's position is measured.

heat[4,1,**lin**]: A one dimensional, grid-based model of the heat equation PDE that models the movement of heat through a grid. The heat at the second point in the grid is measured.

kalman[2,2,**nl**]: A continuous-time Kalman filter that tracks the average of a noisy signal for 200 simulation units [24]. The tracked average of the signal is measured.

smmrxn[1,3,**nl**]: The Michaelis-Menten chemical reaction [31]. The concentration of enzyme-substrate complex is measured.

gentoggle[4,5,**nl**]: A genetic toggle switch [13]. The amount of Tetracycline repressor protein is measured.

bont[5,1,**nl**]: A model of the botulism neurotoxin pathway [23]. The amount of translocated neurotoxin is measured.

8.1 Experimental Setup

For each dynamical system benchmark, we compile and execute a collection of analog device programs and present results from the best performing program. We configure LGraph to generate 3 ADPs and LScale to generate 3 completions for each ADP. We use the lowest possible AQM and DQM for each benchmark.

Hardware Platform: We evaluate the compiled benchmarks on the Sendyne S100Asy RevU development board, which interfaces with the HCDCv2 [15, 18, 43]. The HCDCv2 is a current-mode programmable analog device with a baseline integration speed of 126000 Hz. It has 6 types of compute blocks (**mul**, **adc**, **dac**, **int**, **cout**, and **lut**), 5 types of route blocks, and 1 type of copy block (**fan**). The compute blocks provide multiplication (**mul**) and integration (**int**) operations and support for creating (**dac**), digitizing (**adc**), and externally accessing (**cout**) analog signals on the chip. Only signals between -2 and 2 μA can be externally observed. The HCDCv2 implements user-defined functions using a one-input/one-output programmable lookup table (**lut**) and a free-running analog-to-digital converter (**adc**) and digital-to-analog converter (**dac**) that continuously converts analog and digital signals. Each block on the device has between 1-16 modes that implement between 1-16 basic expressions. There are between 16-128 hierarchically organized instances of each block.

bmark	runtime	power	energy	% rmse
cos	1.59 ms	199.50 μW	0.32 μJ	2.13
cosc	1.34 ms	395.75 μW	0.53 μJ	2.32
pend	0.50 ms	554.82 μW	0.28 μJ	2.11
spring	1.50 ms	913.13 μW	1.37 μJ	4.62
vander	1.25 ms	722.78 μW	0.90 μJ	2.39
pid	6.58 ms	861.03 μW	5.67 μJ	4.27
forced	3.97 ms	849.03 μW	3.37 μJ	5.77
kalconst	3.32 ms	864.28 μW	2.87 μJ	2.29
gentoggle	0.50 ms	804.16 μW	0.40 μJ	3.42
smmrxn	0.52 ms	526.80 μW	0.28 μJ	3.31
bont	0.50 ms	823.25 μW	0.41 μJ	4.74
heat	9.52 ms	556.03 μW	5.30 μJ	0.81

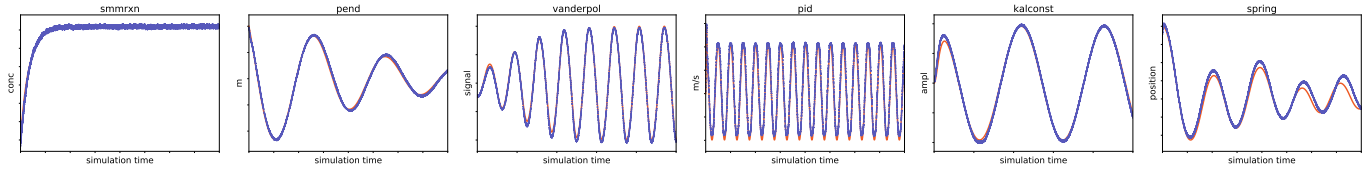
Table 1. Performance, energy, and quality for benchmarks executing on HDACv2 platform

Signal Acquisition and Analysis: We collect waveforms for each benchmark using a Sigilent X1020E oscilloscope. The amplitude of the analog waveform is measured in mV and the time is measured in wall-clock seconds. We invert the applied scaling transform to recover the original signal from each measured waveform. The recovered signal is compared to a reference simulation of the dynamical system computed by a standard digital differential equation solver which digitally simulates the dynamical system with high precision. As appropriate, we shift the measured signal in the time domain and apply minor changes to the time constant (scale by 0.98-1.02x) to account for otherwise uncharacterized deviations in hardware behavior.

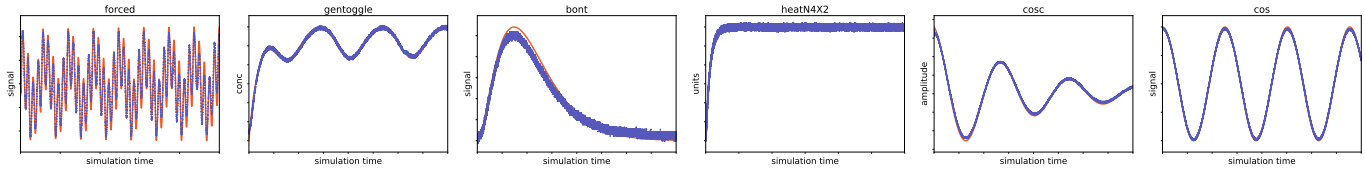
Energy Consumption: We use an empirically-derived energy model provided by our collaborators to estimate the energy consumption of the device [14]. We use a model-based approach, as it is difficult the isolate the power draw of the analog chip because it is embedded on a larger development board with other supporting circuitry. The maximum power consumption of the device is 1.2 mW.

8.2 Analysis

Performance: Table 1 presents the performance characteristics of the dynamical system simulations when executed on the HCDCv2 board. The power consumption ranges from 199.5-913.13 μW. We observe variations in power consumption because only the enabled components draw power. The simulations take between 0.50 ms-9.52 ms to execute and consume between 0.28-5.67 uJ of energy. The root-mean-squared error relative to the dynamic range of the signal is between 0.81-5.77%.

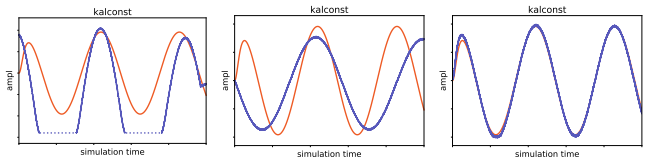


(a) smmrxn (b) pend (c) vander (d) pid (e) kalman (f) spring



(g) forced (h) gentoggle (i) bont (j) heat (k) cosc (l) sin

Figure 24. Benchmark executions on HDACv2 analog board. Red lines are reference signals. Blue lines are measured signals.



(a) single mode (b) no empirical models (c) baseline

Figure 25. Effect of modes/empirical models on accuracy

Quality: Figure 24 presents a comparison of the analog signals measured from the HCDCv2 (blue) with the reference simulation, which was executed on a digital computer with high precision (red). In all cases the analog signal closely tracks the reference simulation.

Physical Systems: Dynamical systems model physical phenomena (e.g., physics and biology simulations) that are inherently approximate: the constants are often derived from empirical measurements and, in many cases, the dynamics are approximations of physical phenomena. With these systems state variable trajectories are typically inspected visually.

Control Systems: Control systems are typically designed with some high-level objective in mind. For the **pid** program, the objective is to attenuate any perturbations. For the (**kalconst**) program, the goal is to track an input signal. Both the analog and reference implementations of these computations meet these objectives.

Effect of Modes: To explore the effect of mode selection on scaling, we artificially limit the available modes for each block to a single mode that restricts the signal operating ranges to [-2,2], increase the DQM and AQM as necessary to enable the scaling phase to succeed, and recompile the benchmark. Figure 25a presents the **kalconst** benchmark executing with the resulting limited operating range and new DQM and AQM of 0.030 and 2.22. We note the dynamics deviate substantially from the reference dynamics, highlighting the fact that the compressed operating ranges do not support accurate execution.

Effect of Manufacturing Deviations: To explore the effect of manufacturing deviations, we use an empirical model database that records no deviations (all the gains are set to one) and recompile the benchmarks. Figure 25b presents the **kalconst** benchmark executing without manufacturing deviation compensation. We note that the dynamics deviate significantly from the reference simulation.

8.3 Analysis of Analog Device Configurations

Table 2 presents the compilation outcomes for the Legno compiler. Columns 1-18 present ADP statistics and columns 19-20 of present the compilation times.

LGraph: Columns 2-11 of Table 2 present the block and connection breakdown for each benchmark. For each benchmark and block type, the number of extraneous blocks relative to the baseline number of blocks is recorded as a second number following a slash. For the **cos**, **cosc**, **pend**, **spring**, and **vanderpol** benchmarks the baseline is computed from hand-implemented configurations written by the hardware designers [14]. For the remaining benchmarks, the baseline is the minimum number of blocks of each type required to implement each benchmark.

The analog device configurations produced by LGraph have between 5-24 blocks and 5-28 connections. LGraph uses between 1 and 6 current copiers (**fan**) and between 1-5 routing blocks (**xbar**) per configuration. LGraph uses every type of compute, copy, and routing block available on the device. LGraph takes between 2.52-5.77 seconds to generate ADPs. For all benchmarks, all extraneous blocks are the result of adding unity coefficients to the DSS (for example, writing **x** as **1.0x**) to introduce degrees of freedom in the scaling process.

For all benchmarks, LGraph used the minimum number of route and compute blocks. One route block is always required to observe the signal because the device output is not directly connectable to any of the compute blocks. The **bont** benchmark uses four additional route blocks because

	LGraph									LScale							compile time (s)	
bench marks	blocks	int	mul	fan	adc	dac	lut	xbar	connections	DQM	AQM	scale factors time	scale factors total	scale factors uniq	injected vars total	injected vars uniq	lgraph	lscale
cos	5	2	0	1	0	0	0	1	5	0.020	0.128	0.10	20	4	0	0	2.52	0.18
cosc	9	2	3	2	0	0	0	1	10	0.020	0.151	0.18	36	12	0	0	3.51	0.79
pend	13	2	4/+2	2	1	1	1	1	14	0.030	0.084	0.32	46	17	2	2	3.59	0.43
spring	24	4	7/+1	5	2	2	2	1	28	0.030	0.123	0.10	88	15	4	2	5.12	0.98
vander	14	2	6/+3	3	0	1	0	1	17	0.010	0.100	0.32	54	17	0	0	4.14	1.33
pid	18	4	7/+3	4	0	1	0	1	21	0.020	0.143	0.24	74	22	0	0	4.81	1.55
forced	17	4	6/+1	4	0	1	0	1	21	0.010	0.169	0.10	70	18	0	0	4.81	1.86
kalconst	18	4	7/+2	4	0	1	0	1	21	0.030	0.231	0.32	74	22	4	4	5.43	1.50
gentoggle	21	4	5/+5	3	2	3	2	1	23	0.030	0.231	0.32	74	22	4	4	4.51	0.81
smmrxn	11	1	4/+1	2	0	2	0	1	13	0.030	0.092	0.32	38	11	0	0	3.99	0.38
bont	23	5	8/+1	4	0	0	0	5	25	0.010	0.084	0.32	90	18	0	0	5.70	1.89
heat	14	4	1	6	0	1	0	1	24	0.030	0.123	0.10	58	6	0	0	5.77	0.73

Table 2. Compilation outcomes for LGraph and LScale passes. All benchmarks use one **cout** block.

it requires more integrators than are available on one tile and therefore had to partition the circuit across two tiles. The minimum number of route blocks to work with this partitioned circuit is five.

LScale: Columns 12-18 of Table 2 present the scaling transform statistics. The scaling transformations produced by LScale have between 4 and 22 unique signal scaling factors. For the **pend**, **spring**, and **gentoggle** benchmarks, LScale injects 2-4 constants into the data fields that store expressions. The time scaling factors are 0.10x-0.32x the baseline speed of the analog device (126000 Hz). The slowest simulation (0.10x baseline) executes one unit of simulation time in $7.94 \cdot 10^{-5}$ seconds of wall-clock time. The lowest AQM ranges from 0.084 to 0.247 and the DQM ranges from 0.01 to 0.03. We note that higher error does not necessarily translate to significantly worse results, as benchmarks have varying levels of robustness. LScale requires between 0.18-1.89 seconds to scale the analog device program.

9 Related Work

Historically analog circuits were manually crafted to perform dynamical system simulations [10, 30, 33, 42]. These circuits contained classical circuit components without the sophistication of contemporary analog accelerators.

Pure and mixed-mode analog accelerators have been developed for accelerating a broad range of applications, including neural networks, SAT solvers, and neuromorphic computations [4, 11, 19, 26, 28, 32, 37]. One prominent line of work focuses on analog accelerators that target dynamical systems [7, 8, 15, 18, 34, 41, 43, 45, 46]. These accelerators cover many points in the hardware design space. Here we focus on current-mode analog devices with simple, flexible blocks that execute general purpose dynamical systems [15, 18, 43].

Prior work has been done on synthesizing analog circuits at the transistor level to assist hardware designers in crafting circuits [3, 17, 29]. Researchers have also developed techniques for programming analog accelerators that implement neural networks to approximate digital subcomputations [12, 40].

Legno uses interval analysis to bound the analog signals for the parameter scaling process. Interval analysis has a long history in fields such as electrical engineering, control theory, and robotics [20, 22]. Legno automatically performs parameter scaling so the resulting analog configuration operates within the physical constraints of the hardware. Parameter scaling is traditionally manually applied to numerical computations to improve the numerical stability [39].

10 Conclusion

Reconfigurable analog devices are a powerful new computing substrate well suited for executing dynamical systems. We present Legno, the first compiler, to our knowledge, for physical devices of this class. Legno is able to construct circuits from specialized and parametric blocks while accounting for analog noise, frequency and operating range constraints, quantization error, and manufacturing variations within the device. We demonstrate that Legno is able to automatically configure the HCDCv2 analog device to obtain acceptably accurate results with low energy consumption. Legno therefore takes a key step towards making this promising new class of devices accessible to a broad range of engineers and systems designers.

Acknowledgements

We would like to thank Yannis Tsividis and Sendyne for granting us access to the analog hardware and for supporting this research. We greatly appreciated their input and expertise during the development of the Legno compiler. The authors are supported by DARPA HACCS HR001118C0059 and DARPA TC FA8650-15-C-7564.

References

[1] Sara Achour and Martin Rinard. Time dilation and contraction for programmable analog devices with jaunt. In *ACM SIGPLAN Notices*, volume 53, pages 229–242. ACM, 2018.

[2] Sara Achour, Rahul Sarpeshkar, and Martin C Rinard. Configuration synthesis for programmable analog devices with arco. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–193. ACM, 2016.

[3] Kurt Antreich, J Eckmueller, Helmut Graeb, Michael Pronath, E Schenkel, R Schwencker, and S Zizala. Wicked: Analog circuit synthesis incorporating mismatch. In *Custom Integrated Circuits Conference, 2000. CICC. Proceedings of the IEEE 2000*, pages 511–514. IEEE, 2000.

[4] Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Shobhit Choudhary, Anand R Chandrasekaran, Jean-Marie Bussat, Rodrigo Alvarez-Icaza, John V Arthur, Paul Merolla, Kwabena Boahen, et al. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716, 2014.

[5] B. E. Boser, E. Sackinger, J. Bromley, Y. Le Cun, and L. D. Jackel. An analog neural network processor with programmable topology. *IEEE Journal of Solid-State Circuits*, 26(12):2017–2025, Dec 1991.

[6] Edward Burnell and Warren Hoburg. Gpkit software for geometric programming. https://github.com/convexengineering/gpkit, 2017. Version 0.6.0.

[7] G.E.R. Cowan, R.C. Melville, and Y. Tsividis. A VLSI analog computer/digital computer accelerator. *Solid-State Circuits, IEEE Journal of*, 41(1):42–53, Jan 2006.

[8] Ramiz Daniel, Sung Sik Woo, Lorenzo Turicchia, and Rahul Sarpeshkar. Analog transistor models of bacterial genetic circuits. In *Biomedical Circuits and Systems Conference (BioCAS), 2011 IEEE*, pages 333–336. IEEE, 2011.

[9] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[10] J.L. Douce and H. Wilson. The automatic synthesis of control systems with constraints. *Mathematics and Computers in Simulation*, 7(1):18 – 22, 1965.

[11] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. MICRO, 2012.

[12] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–460. IEEE Computer Society, 2012.

[13] Timothy S Gardner, Charles R Cantor, and James J Collins. Construction of a genetic toggle switch in escherichia coli. *Nature*, 403(6767):339–342, 2000.

[14] Ning Guo. *Investigation of Energy-Efficient Hybrid Analog/Digital Approximate Computation in Continuous Time*. PhD thesis, Columbia University, 2017.

[15] 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.

[16] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[17] Ramesh Harjani, L Richard Carley, et al. Oasys: A framework for analog circuit synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 8(12):1247–1266, 1989.

[18] Yipeng Huang, Ning Guo, Mingoo Seok, Yannis Tsividis, 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.

[19] Yipeng Huang, Ning Guo, Mingoo Seok, Yannis Tsividis, and Simha Sethumadhavan. Analog computing in a modern context: A linear algebra accelerator case study. *IEEE Micro*, 37(3):30–38, 2017.

[20] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis: With Examples in Parameter and State Estimation, Robust Control and Robotics*. Springer London, 2012.

[21] A. Joubert, B. Belhadj, O. Temam, and R. Héliot. Hardware spiking neurons design: Analog or digital? In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–5, June 2012.

[22] L.V. Kolev. *Interval Methods for Circuit Analysis*. Advanced series on circuits and systems. World Scientific, 1993.

[23] Frank J Lebeda, Michael Adler, Keith Erickson, and Yaroslav Chushak. Onset dynamics of type a botulinum neurotoxin-induced paralysis. *Journal of pharmacokinetics and pharmacodynamics*, 35(3):251, 2008.

[24] Frank L Lewis, Lihua Xie, and Dan Popa. *Optimal and robust estimation: with an introduction to stochastic control theory*. CRC press, 2017.

[25] Stuart Mitchell, Michael Gj O' Sullivan, and Iain Dunning. Pulp : A linear programming toolkit for python. 2011.

[26] Botond Molnár, Ferenc Molnár, Melinda Varga, Zoltán Toroczkai, and Mária Ercsey-Ravasz. A continuous-time maxsat solver with high analog performance. *Nature communications*, 9(1):4864, 2018.

[27] A. J. Montalvo, R. S. Gyurcsik, and J. J. Paulos. An analog vlsi neural network with on-chip perturbation learning. *IEEE Journal of Solid-State Circuits*, 32(4):535–543, April 1997.

[28] Alexander Neckar, Sam Fok, Ben V Benjamin, Terrence C Stewart, Nick N Oza, Aaron R Voelker, Chris Eliasmith, Rajit Manohar, and Kwabena Boahen. Braindrop: A mixed-signal neuromorphic architecture with a dynamical systems-based programming model. *Proceedings of the IEEE*, 107(1):144–164, 2018.

[29] E.S. Ochotta, R.A. Rutenbar, and L.R. Carley. Synthesis of high-performance analog circuits in ASTRX/OBLX. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 15(3):273–294, Mar 1996.

[30] Yakup Paker and Stephen H. Unger. {ADAC} — a programmed direct analog computer. *Mathematics and Computers in Simulation*, 9(1):16 – 23, 1967.

[31] Denise R. Ferrier PhD. *Biochemistry (Lippincott Illustrated Reviews Series)*. LWW, 2013.

[32] Sylvain Saighi, Yannick Bornat, Jean Tomas, Gwendal Le Masson, and Sylvie Renaud. A library of analog operators based on the Hodgkin-Huxley formalism for the design of tunable, real-time, silicon neurons. *Biomedical Circuits and Systems, IEEE Transactions on*, 5(1):3–19, 2011.

[33] Sams. Arrangement and scaling of equations. *Mathematics and Computers in Simulation*, 6(3):179 – 182, 1964.

[34] Rahul Sarpeshkar. *Ultra Low Power Bioelectronics: Fundamentals, Biomedical Applications, and Bio-Inspired Systems*. Cambridge University Press, 2010.

[35] Srinagesh Satyanarayana, Yannis Tsividis, and Hans Peter Graf. A reconfigurable analog vlsi neural network chip. In *Proceedings of the 2Nd International Conference on Neural Information Processing Systems*, NIPS'89, pages 758–768, 1989.

[36] J. Schemmel, J. Fieres, and K. Meier. Wafer-scale integration of analog neural networks. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 431–438, June 2008.

[37] Christian Schneider and Howard Card. Analog CMOS synaptic learning circuits adapted from invertebrate biology. *Circuits and Systems, IEEE Transactions on*, 38(12):1430–1438, 1991.

[38] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[39] Robert D Skeel. Scaling for numerical stability in gaussian elimination. *Journal of the ACM (JACM)*, 26(3):494–526, 1979.

[40] Renée St Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. General-purpose code acceleration with limited-precision analog computation. *ACM SIGARCH Computer Architecture News*, 42(3):505–516, 2014.

[41] Jonathan J. Y. Teo, Sung Sik Woo, and Rahul Sarpeshkar. Synthetic biology: A unifying view and review using analog circuits. *IEEE Trans. Biomed. Circuits and Systems*, 9(4):453–474, 2015.

[42] Rajko Tomovic. Proceedings of the international association for analog computation method of iteration and analog computation. *Mathematics and Computers in Simulation*, 1(2):60 – 63, 1958.

[43] Yannis Tsividis. Not your father's analog computer. *IEEE Spectrum*, 55(2):38–43, 2018.

[44] Lieven Vandenberghe. The cvxopt linear and quadratic cone program solvers. *Online: http://cvxopt. org/documentation/coneprog. pdf*, 2010.

[45] Sung Sik Woo, Jaewook Kim, and Rahul Sarpeshkar. A cytomorphic chip for quantitative modeling of fundamental bio-molecular circuits. *IEEE Trans. Biomed. Circuits and Systems*, 9(4):527–542, 2015.

[46] Sung Sik Woo, Jaewook Kim, and Rahul Sarpeshkar. A digitally programmable cytomorphic chip for simulation of arbitrary biochemical reaction networks. *IEEE transactions on biomedical circuits and systems*, 12(2):360–378, 2018.

Algorithm: Compilation of dynamical system to analog device
Compilation: Python 3.X
Dataset: Oscilloscope data collected from analog device
Runtime environment: Linux, Mac OSX
Hardware: HCDCv2 Device / Sendyne S100Asy RevU Development Board
Runtime State: `grendel` firmware
Execution: Python 3.X
Metrics: compile time, energy consumption, execution time, qualitative accuracy, resource utilization and program complexity metrics
Output: execution time, energy consumption, generated program complexity.
Experiments: semi-automated
Space required: 7.3 GB
Setup time: 10 minutes
Evaluation time: 3 hours
Publicly available: yes, GPL and CC licenses
DOI: `https://doi.org/10.5281/zenodo.3606664`

Figure 26. artifact summary.

A Artifact Appendix

Abstract: The Legno compiler is a compilation toolchain for the HCDCv2 analog device. This toolchain is able to compile dynamical systems to execution scripts that can be dispatched to the analog hardware. The Legno compiler is the first compiler to target a real-world differential equation solving analog device. This guide outlines how to download and install the publicly available Legno toolchain.
Description: Using these instructions, the user should be able to perform the actions listed below. This manually specially formats `commands`, `directories`, and `files`:

1. Compile the benchmark applications using the Legno toolchain. The compilation results are used to generate the results for the prose in Section 8 and Tables 2, 1.
2. Analyze the oscilloscope data to generate the results in Table 1 and Figures 24,25.
3. Add a new program and compile it using the Legno toolchain.

Hardware dependencies: This compiler generates configurations for the HCDCv2 analog device. This device is not yet commercially available. The full dataset of oscilloscope waveforms and a video demonstrating the operation of the device are included in the data artifact.
Software dependencies: Evaluators need **docker** installed to use the containerized version of the compiler. To run the compiler natively, evaluators can follow the instructions outlined in the user manual included in the github repository.
Datasets: Two datasets that contain all the collected oscilloscope waveforms are included in the artifact. These datasets contain the produced analog device programs, waveforms and visualizations for all the benchmarks.

A.1 Downloading the Toolchain

To start, clone the Legno git repository containing the compiler. This project contains the documentation, the compiler benchmarks, and the compiler source code. Next, navigate to the `docker/` directory to begin building the docker container:

```
git clone \
  https://github.com/sendyne/legno-compiler.git
cd legno-compiler/docker/
```

The **docker** directory should contain scripts for building the docker image (`Dockerfile`, `build_image.sh`, `run_image.sh`), table and plot generation scripts for generating the paper results (`generate_*.sh`, a manual for interpreting results (`evaluation_manual.pdf`), a script for compiling the benchmarks (`run_all.sh`), and a script for converting `.tbl` files to latex code (`latex_gen`).
Next, visit the **DOI** link from the Figure 26 to download the required data. Download the `oscilloscope_data.zip`, `oscilloscope_data_standard.zip`, and `state.db` files to the `docker/` directory. This DOI entry contains the necessary oscilloscope datasets (`oscilloscope_data*.zip`), the empirical model database derived for our device (`state.db`), demo videos demonstrating the analog hardware executing the dampened oscillator (`demo*.mp4`), and an archives of oscilloscope data visualizations (`quality_graphs.zip`)

A.2 Installation

To build the **docker** image, execute the `./build_image.sh` command (5 minutes). This command creates a docker image named `legno-container`. There may be some errors during the model inference procedure – these are safe to ignore. Once this completes successfully, you can create and login to a docker container using the `./run_image.sh` command. This command creates and mounts two externally accessible shared directories `outputs` and `PAPER` in the image and log you into the docker container. The project directory can be found in `/root/legno_compiler`. The shared folders correspond to the automatically created `outputs` and `PAPER` directories in the **docker** directory.

- `output` **directory**: This directory contains all the compilation outputs. The `output/logs` subdirectory contains all the standard output/error logs from execution. The `output/legno/extended` directory contains the compilation outputs for all the benchmarks.
- `PAPER` **directory**: This directory contains all the generated figures and tables for the paper. The figures and tables in this directory can be compared to the figures and tables included in the `paper_data.pdf` file.

A.3 Executing the Compiler

The commands presented below compile all the benchmarks in the paper [60 minutes, 1-2 minutes/operation]:

```
python3 run_all.py > run_all_bmarks.sh
chmod +x run_all_bmarks.sh
time ./run_all_bmarks.sh
```

The compilation outputs are written to the shared **output/legno/extended** directory, and can be accessed from the container and host machine. Each benchmark subdirectory contains unscaled and scaled ADP visualizations (**lgraph-diag** and **lscale-diag**), grendel files (**grendel**), and execution times for the different compilation steps (**times**).

A.4 Interpreting Filenames

LEGNO encodes the compilation parameters into the name of each intermediate and output Grendel file.

```
cosc_g0x0_s0_dgd4.00a15.10v2.77c96.00b80.00k
   _obsfast_t20_osc.grendel
```

The above Grendel script executes the **cosc** benchmark for 20 simulation units [**t20**] and collects data with the oscilloscope [**osc**]. This script implements the unscaled ADP with identifier **g0x0** (see **lgraph-diag**) and the mode selection and scaling transform with identifier **s0** (see **lscale-diag**). The implemented scaling transform has a maximum frequency of **80** khz [**b80.0k**], an AQM of 0.1510 (**a15.10**), and a DQM of 0.04 [**d4.00**]. The AQM for state variables is 0.0277 [**v2.77**] and the user-defined functions utilize 96% of the device lookup tables (**c96.00**). The implemented scaling transform maximizes the speed and the dynamic range of the observed signals subject to the above restrictions [**obsfast**]. The scaling transform was produced using empirical models elicited from the device (**dg**) – this is called the device *tag*. Please refer to the user manual for a complete description of tags. Some benchmarks may be missing ADPs **dg** and **de** tags – this is because LSCALE cannot scale ADPs using the empirical model database if any of the necessary models are missing.

A.5 Building Compilation Results Tables

The compilation outputs from the previous step can be used to build produce the tables and figures in Section 8[few minutes]. Execute the **./generate_tables.sh** command from the docker container to generate the following tables (**.tbl** files) to the **PAPER** directory:

File	Table / Figure
bmarks.tbl	benchmark prose, Section 8
hwblocks.tbl	HCDCv2 prose, Section 8
hwboard.tbl	HCDCv2 prose, Section 8
circuit-lgraph.tbl	columns 2-11, Table 2
circuit-lscale.tbl	columns 12-18, Table 2
compile-time.tbl	columns 19-20, Table 2
energy-runtime.tbl	columns 2-4, Table 1

A.6 Analyzing Oscilloscope Data

Execute the **./generate_quality_plots.sh** command from the **legno-compiler** directory of the docker container to analyze the oscilloscope dataset [75 minutes]. This script invokes **exp_driver.py** program, which invokes source files from the **scripts** directory and stores analysis results in the database **outputs/experiment.db**. This database can be regenerated by using the **scan** and **analyze** **exp_driver.py** commands (see the **generate_quality_plot.sh** script). Note that the analysis script will backup the compilation outputs generated from the previous step to the directory **outputs/local-results**. This procedure generates the following visualizations in the **PAPER** directory:

File	Table / Figure
-delta-max-fit-	Figure24
-kalconst--naive-max-fit-*	Figure 25b
paper-quality-energy-runtime.tbl	Table 1

The individual oscilloscope plots are written to the **outputs/legno/extended/bmark/plots** directory. After evaluating the analysis outputs, restore the compiler results with:

```
python3 setup_exp_data.py restore
```

A.6.1 Analyzing standard Oscilloscope Data (Figure 25a)

The **oscilloscope_data_standard.zip** file contains the oscilloscope data for the standard single-mode executions described in Section 8.2. To analyze this dataset, replace **oscilloscope_data.zip** with **oscilloscope_data_standard.zip** in the docker container and then execute **python3 setup_exp_data.py install**. The **outputs/legno/standard** directory should have output directories for the **spring** and **kalconst** benchmarks. Execute the following to produce quality graphs for the **standard executions**:

```
./generate_standard_quality_plots.sh
```

The **paper-kalconst-standard-delta-max-fit-best.pdf** file is the plot used in Figure 25a.

A.7 Compiling a Custom DSS

The benchmark DSS s are in the **progs/columbia** and **progs/biology** directory. To compile a custom DSS, first copy the template DSS to a new file using the command:

```
cp progs/template.py progs/myprog.py
```

Modify **dsname** to return the program name (alphabetical characters only). Modify **dsprog** function to define the desired dynamical system (see **Chapter 3** of **user_manual.pdf**). Modify the **dssim** function to return the number of simulation units to execute the simulation for. The following command executes the DSS named **myprog**:

```
time python3 legno_runner.py --config \
   configs/default_maxfit_naive.cfg \
   myprog --lgraph --ignore-missing
```

The compilation outputs can be found in the output directory **outputs/legno/extended/myprog/**.