Contents

Chapter 1

Installation

The `Legno` compiler has been tested on OSX and Linux systems. It is a command-line utility – all commands in this document should be executed using the command line.

1.1 Setting up the Compilation Environment

1.1.1 Mac OS

In order to install the dependencies for the `Legno` compiler, the `brew` package manager (https://brew.sh/). Brew is installed using the following command:

```
/usr/bin/ruby -e \
"\$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Once brew is installed, it may be accessed using the command `brew`. Next, we should install the required dependences.

```
brew install python3 git graphviz
```

You might need to execute the following command to get get `pip3` setup:

```
brew postinstall python3
```

Testing the installation:If this step executed correctly, each of these commands should return the usage information for the command:

```
pip3 --help
python3 --help
git --help
dot --help
```

If any of these commands return the following error (where program is `pip3`, `python3`, `git` or `dot`), please contact me for help:

```
-bash: <program>: command not found
```

1.1.2 Linux

First, install the required dependencies

```
sudo apt-get install -y python3-pip libatlas-base-dev git graphviz
```

1.2 Setting up the Compilation Toolchain

First, we need to download the compilation toolchain and make sure we have selected the `master` branch:

```
git clone git@github.com:sendyne/legno-compiler.git
git checkout master
```

This should create a `legno-compiler` directory; inside this directory is the legno compiler toolchain.

1.2.1 Setting up the Compiler

The compiler is a pure python program, so almost all of the dependences can be installed using the python package manager, `pip`. Execute the following command from the root directory of the `legno-compiler` project to execute all the required packages:

```
pip3 install -r packages-legno.list
```

1.3 Setting up the HCDCv2 Device

These steps are only necessary if you wish to execute the generated programs on the HCDCv2 device. To write the device firmware to the SA100ASY development board, you must install the Arduino IDE. To install the Arduino IDE, download the executable from the following link and drag it do your `Applications` directory:

```
https://www.arduino.cc/en/Main/Software
```

After installing the Arduino IDE, navigate to `Tools/Board: <Board Name>` in the menu, and select the `Boards Manager` option. After doing so, search for `Due` and install the `Arduino Sam Boards (32 bits ARM-Cortex M3` package. We installed `1.6.12` of `DueTimer`.

You also want to install the `DueTimer` package. This can be done by navigating to the `Sketch/Include Library/Manage Libraries` option in the Arduino IDE and searching for the `DueTimer` package. We installed version `1.4.7` of `DueTimer`.

1.3.1 Installing Arduino-Make

The firmware is built using a Makefile that extends a specialized set Arduino Makefiles. The Arduino makefiles are provided in the `arduino-mk` package.

MacOS Command

The following installs the arduino makefiles on OSX:

```
brew tap sudar/arduino-mk
brew install --HEAD arduino-mk
```

Linux Command

The following installs the arduino Makefiles on linux.

```
sudo apt-get install -y android-mk
```

1.3.2 Installing the Grendel Runtime Dependences

The firmware communicates with a runtime which is also written in python. Execute the following command from the root directory of the `legno-compiler` project to install all the required packages.

```
pip3 install -r packages-grendel.list
```

1.3.3 Writing Firmware to the HCDCv2 Device

After following these steps, it should be possible to flash the HCDCv2 firmware to the analog device. Navigate to the following directory:

```
legno-compiler/lab_bench/arduino/grendel_interp_V1
```

To build the firmware, type the following command:

```
make
```

This should complete without incident. Next we will upload the firmware to the device. Please ensure the device is programmed via USB, and that the USB cable is plugged into the programming port of the device. This is the port closest to the DC power port. After connecting the device to the computer, type the following command to flash the firmware to the device:

```
make upload
```

When you're done, navigate back to the `legno-compiler` directory.

```
legno-compiler/
```

1.3.4 Setting up the Sigilent 1020XE Oscilloscope

This toolchain works with the Sigilent 1020XE Oscilloscope. Any part of the manual that requires the oscilloscope be set up will be annotated an `[OSC]` tag. To use the oscilloscope, make the following connections between the oscilloscope and the board:

1. Connect the first channel to pin A0 of the analog device (positive channel of chip[0,3,2])
2. Connect the second channel to pin A1 of the analog device (negative channel of chip [0,3,2]).

3. Connect the EXT lead to pin 25/SDA1 of the Arduino. The Arduino triggers the oscilloscope to record data.

Next, we have to connect the oscilloscope to the network. First, connect the oscilloscope to an Ethernet jack (the port is in the back of the device). We also need the IP address of the device. To retrieve the IP address, press the `Utility` button on the Oscilloscope control panel. This should change the menu at the bottom of the screen. Navigate to page 2 of the menu, select the `IO` option, and then select the `IP Set` option. This should display the IP address on the oscilloscope's screen. Write it down and check you can reach the oscilloscope from your computer using the following command:

```
ping <oscilloscope-ip>
```

1.4 Configuring the Legno Toolchain

The `Legno` toolchain must first be configured before it can be used. First, copy the reference configuration:

```
cp util/config_local.py util/config.py
```

The following fields may be adjusted as needed in the config file. Note that the only field that should definitely be changed is the `OSC_IP` field:

1. `OSC_IP`: populate this field with the IP address of the oscilloscope. If you are not using an oscilloscope, you can leave this field as is. Refer to Section 1.3.4 for instructions on how to get the IP address of the oscilloscope.
2. `DEVSTATE_PATH` (default recommended): set this field to the directory where all the device data (calibration information, analog block models, etc) should be stored. We recommend leaving this field unchanged.
3. `OUTPUT_PATH` (default recommended): set this field to the directory where all the compiler outputs should be stored. We recommend leaving this field unchanged.

The remaining fields are automatically populated from these three fields. We don't recommend directly setting any of these. The `ARDUINO_FILE_DESC` field points to the USB port that is linked to the Arduino and is automatically populated by looking for the proper file descriptor in the `/dev/` directory.

1.5 Troubleshooting

The following section provides fixes for common errors that occur during installation.

1.5.1 `brew link` when installing `python3`

If you get the following error trying to install `python3` using brew:

```
Linking /usr/local/Cellar/python/3.7.5... Error: Permission denied @ dir_s_mkdir
- /usr/local/Frameworks
```

Execute the following commands to fix it:

```
sudo mkdir /usr/local/Frameworks
sudo chown -R \$(whoami) /usr/local/Frameworks/
brew link python
brew postinstall python3
```

1.5.2 `pyserial` missing when making firmware

If you get the following error when trying to compile the firmware:

```
import failed, serial not found
```

This is likely because `arduino-mk` is using a different version of python than the rest of the project. To mitigate this, install pyserial for `python2`:

```
pip install pyserial
```

1.5.3 ARDUINO_DIR not defined when making firmware

If you get the following error while trying to compile the firmware:

```
/usr/share/arduino/Arduino.mk:279: *** "ARDUINO_DIR is not defined". Stop.
```

This is likely because the Arduino IDE is not properly installed. Please make sure that the Arduino IDE is in the `Applications` folder in OSX.

Chapter 2

Legno Quickstart

The following quickstart walks through compiling a dynamical system program that implements the cosine (`cos`) function. This section

2.1 Compiling the `cos` Program

The following section describes how to compile the cosine program. We use the `legno_runner.py` script to compile the program. This script executes all the necessary compiler passes (by invoking `legno.py` multiple times) to generate `grendel` scripts, using the parameters defined in `configs/quickstart.cfg`. The cosine program is compiled by executing the following command from the `legno-compiler` directory:

```
python3 legno_runner.py --config configs/quickstart.cfg --hwenv osc \
  cos --lgraph
```

This command executes all the necessary compiler passes to generate `.grendel` scripts, using the parameters defined in `configs/quickstart.cfg`. The grendel script can then be executed on the analog device.

2.1.1 The `quickstart.cfg` File

The `quickstart.cfg` file is a `json` dictionary with the following fields:

```
{
  "subset":"extended",
  "model":"naive-max_fit",
  "n-lgraph":1,
  "n-lscale":1,
  "max-freq":80
}
```

This configuration file specifies the parameters to provide to each compilation pass in the compilation process. We describe the fields below:

- **subset**: The subset argument is passed in at each step of the compilation process. It dictates what subset of analog device features the compiler should use. The `extended` subset allows the compiler to use medium ([-2,2] uA) and high ([-20,20] uA) modes. It is the broadest set of tested features on the device.

- **model**: The model argument tells the circuit scaling pass (`lscale`, Section **??**) how to model the behavior of the analog blocks. The `naive-max_fit` model tells the circuit scaling pass to assume the block behavior adheres to the behavior described in the hardware specification (`naive`). It also tells the compiler the blocks will be calibrated using the `max_fit` calibration objective (see Section XXX below).

- **n-lgraph**: The number of unscaled circuits the graph synthesis pass should generate (`lgraph`, Section 5)

- **n-lscale**: The number of scaled circuits to produce per unscaled circuit. This is provided to the circuit scaling pass (`lscale`, Section **??**).

- **max-freq**: The maximum allowable speed of the simulation. This is provided to the circuit scaling pass (`lscale`, Section **??**).

2.1.2 Inspecting the Compilation Outputs

The `legno_runner.py` script produces unscaled circuits, scaled circuits and grendel scripts. All compilation and execution outputs for the cosine program is stored in the following directory:

```
legno-compiler/outputs/legno/extended/cos/
```

We call this the *program directory*. The following sub-directories in the program directory contain compilation outputs:

- `lgraph-adp` and `lgraph-diag`: These directories contain the unscaled analog device programs and associated diagrams. The diagrams are visual representations of the programs that are useful for debugging.

- `lscale-adp` and `lscale-diag`: These directories contain the scaled analog device programs and associated diagrams.

- `grendel`: This directory contains the compiled `.grendel` scripts generated by the `srcgen` pass of the `Legno` compiler.

- `times`: this directory contains the runtime information for each pass.

We should see one `.adp` file in the `lscale-adp` and `lgraph-adp` directories, and one `.png`/`.dot` file in the `lgraph-diag` and `lscale-diag` directories. There should be one `.grendel` file in the `grendel` directory that should have a name similar to the one below:

```
cos_g0x0_s0_ngd3.00a12.28v1.77c97.00b80.00k_obsfast_t20_osc.grendel
```

For brevity, we will refer to this file as `<file>.grendel` for the rest of the quickstart section.

2.2 Running the Cosine Grendel Script

The grendel runtime (`grendel.py`) executes `.grendel` scripts generated by the `Legno` compiler. Before executing the script, we must calibrate any uncalibrated blocks. To do so, execute the following command from the `legno-compiler` directory:

```
python3 grendel.py calibrate --calib-obj max_fit \
  outputs/legno/extended/cos/grendel/<file>.grendel
```

The `--calib-obj` argument specifies which calibration objective to use. We use the `max_fit` calibration objective. We can run the benchmark after all the blocks have been calibrated. Execute the following command to execute the program an the analog computer:

```
python3 grendel.py run --calib-obj max_fit \
  outputs/legno/extended/cos/grendel/<file>.grendel
```

If the script executed correctly, you should see a `.json` file appear in the `out-waveforms` directory. This file is the waveform downloaded from the oscilloscope:

```
legno-compiler/outputs/legno/extended/cos/out-waveform/
```

2.2.1 Execution without the Sigilent 1020XE Oscilloscope

:

If you are not using the Sigilent 1020XE oscilloscope, include the `--no-oscilloscope` flag to `grendel.py` commands to prevent the runtime from attempting to communicate with the measurement device. If you would like to use an unsupported oscilloscope, configure the voltage and time divisions on the oscilloscope using the voltage and time ranges in the `<file>.grendel` file (see Section 7) and set it to wait for an edge trigger.

If the script executed correctly, you should see a cosine waveform on your oscilloscope. Note that you will not see a `.json` file in the `out-waveforms` directory, because the runtime cannot communicate with an unsupported oscilloscope.

2.3 Analyzing Oscilloscope Waveform Data [OSC]

The `exp_driver.py` tool is used to manage and analyze oscilloscope outputs. We execute the following command from the `legno-compiler` directory to discover all the grendel scripts and oscilloscope waveforms in the `output` directory:

```
python3 exp_driver.py scan
```

The discovered waveforms can then be analyzed with the following command. Currently, `exp_driver.py` is able to automatically compare waveforms to digital simulations:

```
python3 exp_driver.py analyze
```

The oscilloscope waveforms and visual analysis results are written to the `plots` directory in the program directory:

```
outputs/legno/extended/cos/plots
```

Chapter 3

Dynamical System Language

The `Legno` compiler (`legno.py`) enables developers to compile dynamical systems down to configurations for the analog hardware. The `Legno` compiler requires that dynamical systems be specified in the `dynamical system language`, a high level language that supports writing first-order differential equations.

3.1 Example: Cosine Program

The following dynamical system program implements the cosine function. This program can be found in `progs/quickstart/cos.py`. In order for `legno.py` to find the program, the program must be placed in the `progs/` directory.:

```
from dslang.dsprog import DSProg
from dslang.dssim import DSSim,DSInfo

def dsname():
  return "cos"

def dsinfo():
  return DSInfo(name=dsname(), \
                desc="cosine",
                meas="signal",
                units="signal")
  info.nonlinear = False
  return info

def dsprog(prob):
  params = {
    'P0': 1.0,
    'V0' :0.0
  }
```

```
    prob.decl_stvar("V","(-P)","{V0}",params)
    prob.decl_stvar("P","V","{P0}",params)
    prob.emit("P","Position")
    prob.interval("P",-1.0,1.0)
    prob.interval("V",-1.0,1.0)
    prob.check()
    return prob

  def dssim():
    exp = DSSim('t20')
    exp.set_sim_time(20)
    return exp
```

A dynamical system program must define four python functions in order to be used by the compiler:

- **dsname**: This function returns the name of the program. The user can later refer to this program by its name during compilation.
- **dsinfo**: This function returns detailed information about the program. This includes a short description of the program, the name of the signal being measured and the units of the signal being measured. Only one signal can be measured right now.
- **dsprog**: This function returns the dynamical system associated with the program. The dynamical system is defined as a collection of state variable and function declarations. The dynamical system also contains interval annotations for each state variable (this defines the bounds for each variable) and which variable is measured. The `check()` function checks to see that the system is bounded.
- **dssim**: This function returns the simulation parameters. This program is to be run for 20 simulation units.

3.1.1 Breaking Down `dsprog`

In this dynamical system, the position `P` corresponds to the amplitude of the cosine function. The V and P variables are internal variables that are used to model the dynamics of the system. We describe each line of the program below:

- `params = {..}`: This statement creates a parameter map. The parameter map associates names with values. This is used to fill in parameters when defining the dynamical system. Parameters are referred to by their name, enclosed in curly braces (for example `\{V0\}` refers to the parameter V0).
- `prob.decl_stvar("V","(-P)","{V0}",params)`: This statement defines a state variable V, whose dynamics are governed by $V' = -P$. The initial value of the state variable is the value of parameter V0. The last argument is the parameter map to use.
- `prob.decl_stvar("P","V","{P0}",params)`: This statement defines a state variable P, whose dynamics are governed by $P' = V$. The initial value of the state variable is the value of parameter P0. The last argument is the parameter map to use.

- `prob.emit("P","Position")`: This statement indicates the expression P is of interest, and should be observed. The observation is named `Position`.
- `prob.interval("P",-1.0,1.0)`: This statement defines the variable `P` as falling within the bounds [-1,1]. All state variables must be restricted to a user-defined interval in order for the program to pass the well formedness check (`check()` invocation).
- `prob.interval("V",-1.0,1.0)`: This statement defines the variable `V` as falling within the bounds[-1,1].
- `prob.check()`: This statement checks that all the variables in the dynamical system are bounded. This function must pass (not trigger an error) in order for the program to compile successfully.

3.1.2 Executing the `cos` Dynamical System with a ODE Solver

The `Legno` compiler supports digitally simulating dynamical systems for testing purposes.

```
python3 legno.py --subset extended simulate cos --reference
```

The `--reference` argument tells the compiler to perform a digital simulation and produce reference figures. To profile the runtime of the reference simulation, add the `--runtime` argument.

Note that the `--subset` argument is used to determine where to write the reference simulation – it does not impact the behavior of the simulation. The digital simulator writes the plots of the state variable trajectories to the following directory:

```
legno-compiler/outputs/legno/extended/cos/sim/ref/
```

3.2 Creating your own Dynamical System

The following section walks through how to create a dynamical system named `deg` that implements exponential decay. First, copy the dynamical system template to a new file. This file will automatically be picked up by the compiler:

```
cp progs/template.py progs/deg.py
```

Next we modify the contents of `test.py` to implement the desired dynamical system. First we change the name of the program to `decay`:

```
def dsname():
   return "decay"
```

Next, we implement the dynamical system by populating the `dsprog` function. The exponential decay function is implemented as $x' = -k \cdot x$, where k is the decay parameter. The dynamical system takes one additional parameter, $x(0)$, the initial condition of the system. We are interested in executing the dynamical system with $k = 0.1$ and $x(0) = 10$. We define the following parameter dictionary:

```
params = {'k':0.1,'x0':10.0}
```

This system has one state variable, x. We next define the dynamics of x with the statement below.

```
prog.decl_stvar("x","{k}*(-x)","{x0}",params)
```

This statement creates a new state variable named `x`, whose derivative is `k*(-x)` (`k` is a parameter) and initial condition is the parameter `x0`. Wrapping a variable in curly braces indicates it's a parameter – the compiler will look for the parameter in the parameter map that is passed in. The last argument, `params`, provides the parameter map to the compiler.

We are interested in observing this state variable; we tell the compiler we would like to forward this variable to an external pin for observation with the `emit` command. This signal will be named `OBS` in any generated plots:

```
prog.emit('x','OBS')
```

We also need to provide an interval range annotation for x. This annotation tells the compiler the range of values x may take on. It is important to set an appropriate dynamic range for each variable. If this range is smaller than the actual dynamic range of x, it may saturate. If this range is much larger than the actual dynamic range, then x might be overtaken by noise and error during execution.

We know that x starts at 10.0 and decays to 0.0. It therefore always falls within $[0, 10.0]$. We set the interval of x with the following command:

```
prog.interval('x',0,10.0)
```

The interval of OBS is automatically derived from x. The full `dsprog` function is shown below:

```
def dsprog(prog):
  params = {'k':0.1,'x0':10.0}
  prog.decl_stvar("x","-{k}*x","{x0}",params)
  prog.emit('x','OBS')
  prog.interval('x',0,10.0)
  prog.check()
  return
```

We next need to provide the simulation parameters of the system by changing the `dssim` function. We are fine with executing the simulation for 20 simulation units, so we can leave the `dssim` function as is:

```
def dssim():
  exp = DSSim('t20')
  exp.set_sim_time(20)
  return exp
```

Next, we change the program information to reflect the exponential decay system. The `decay` system is a linear system that tracks the trajectory of x, which is measured in arbitrary units.

```
def dsinfo():
  info = DSInfo(dsname(), \
                desc="exponential decay"
                meas="trajectory",
                units="units")
  info.nonlinear = False
  return info
```

We can test the dynamical system by digitally simulating with using the command below:

```
python3 legno.py --subset extended simulate decay --reference
```

The command will write plots of `x` and `OBS` to the following directory:

```
legno-compiler/outputs/legno/extended/decay/sim/ref/
```

3.3 Troubleshooting

3.3.1 `handle not in interval` when executing `check()`

If you get the following error:

```
Exception: handle not in interval: :h0
```

This indicates that the compiler could not compute intervals for all the variables in the dynamical system. Please make sure all state variables and functions (optionally, usually) are bounded using `interval` commands.

3.3.2 `Legno` cannot find my dynamical system program!

Make sure that (1) the program is a `.py` file that resides somewhere in the `legno-compiler/prog` directory (2) the `dssim`, `dsinfo`, `dsname` and `dsprog` functions are defined in the file.

Chapter 4

Compiler Overview

The compilation toolchain contains five major components:

- **Compiler** (`legno.py` and `legno_runner.py`): The compiler. It reads dynamical system programs (in the `prog` directory) and generates grendel scripts. These scripts can be executed by the runtime.
- **Runtime** (`grendel.py`): The runtime. It executes grendel scripts on the analog device by dispatching commands to the microcontroller.
- **Firmware** (`lab_bench` directory): The firmware. This must be written to the microcontroller in order for the microcontroller to understand the grendel commands.
- **Experiment Driver** (`exp_driver.py`): This tool analyzes the outputs produced by the analog chip. It computes the energy, power, quality (with respect to a reference simulation) and runtime of each program.
- **Model Builder** (`model_builder.py`): This tool builds delta models from empirical data collected from analog blocks using grendel commands. These models are used by the `Legno` compiler during the `LScale` and `SrcGen` pass to compensate for behavioral deviations in the device.

4.1 `Legno` Compiler

The `Legno` compiler (`legno.py`) enables developers to compile dynamical systems down to configurations for the analog hardware. The `Legno` compiler requires that dynamical systems be specified in the `dynamical system language`, a high level language that supports writing first-order differential equations. The `Legno` compiler also accepts a specification of the target analog device, described using the Analog Device API. The `Legno` compiler generates an *analog device program* (`adp`) which implements the target dynamical system on the specified analog device. The compiler is broken up into three passes:

- `LGraph`: This pass synthesizes an unscaled circuit that implements the specified dynamical system.

- `LScale`: This pass synthesizes one or more scaled circuits for each unscaled circuit generated by the `LGraph` pass.

- `SrcGen`: This pass generates a `.grendel` script for each scaled circuit generated by the `LScale` path. The `.grendel` script configures the analog device and optionally the oscilloscope, and executes the simulation described in the dynamical system program.

The `Legno` compiler accepts a `--subset` argument which indicates what subset of device features to use to compile the program. There are three subsets supported by the compiler:

- `standard` subset: This limits the accepted modes of each block to the medium mode (-2 uA to 2 uA).

- `extended` subset: This limits the accepted modes of each block to the high (-20 uA to 20 uA) and medium (-2 uA to 2 uA) modes.

- `unrestricted` subset: This allows all modes (low, medium and high) to be used. This subset involves modes that have not been thoroughly tested by us.

The `Legno` compiler writes all compilation outputs for some program `<prog>` compiled with the feature subset `<subset>` to the following directory:

`outputs/legno/<subset>/<program>/`

This directory has eight subdirectories:

- `lgraph-adp` and `lgraph-diag`: These directories contain the unscaled analog device programs and associated diagrams. The diagrams are visual representations of the programs that are useful for debugging. These files are produced by the `lgraph` pass of the `Legno` compiler (Section 5).

- `lscale-adp` and `lscale-diag`: These directories contain the scaled analog device programs and associated diagrams. The diagrams are visual representations of the programs that are useful for debugging. These files are produced by the `lscale` pass of the `Legno` compiler (Section **??**), and are derived from files in the `lgraph-adp` directory.

- `grendel`: This directory contains the compiled `.grendel` scripts generated by the `SrcGen` pass of the `Legno` compiler, and are derived from scaled circuits produced by the `lscale` pass.

- `times`: this directory contains the runtime information for each compilation pass.

- `sim`: this directory contains digital simulation results for the program. These results are generated by the `simulate` operation of the `Legno` compiler (see Section 3)

- `out-waveform` (oscilloscope): this directory contains the output waveforms collected by the oscilloscope during execution. This directory is populated by the `Grendel` runtime.

- `plots` (oscilloscope): this directory contains visualizations of the output waveforms collected by the oscilloscope. These plots are generated by the experiment driver (`exp_driver.py`).

4.2 Runtime System

The `Grendel` runtime dispatches `.grendel` scripts to the analog device. The `Grendel` scripting language supports configuring the analog device, running experiments and setting up the Sigilent 1020XE oscilloscope. It collects and writes the oscilloscope data to the following directory (given a ¡program¿ compiled with the ¡subset¿ feature set)

```
outputs/legno/<subset>/<program>/out-waveform
```

The grendel runtime communicates with a microcontroller running the `Grendel` firmware. This firmware is found in the following directory:

```
legno-compiler/lab_bench/arduino/grendel_interp_V1
```

4.2.1 Calibration and Profiling

The `Grendel` runtime system is able to calibrate and profile blocks that appear in a `Grendel` script. The system caches combinations of calibration codes and profiling information in the *state database* (`state.db`), which resides at the following location:

```
legno-compiler/device-state/state.db
```

Calibration: The `Grendel` runtime is able to calibrate blocks to minimize some objective function. The code values selected by the calibration routine are cached in the state database. The runtime and firmware currently supports two objective functions:

- `min_error`: This objective function minimizes the error of the output across a sequence of test inputs.
- `max_fit`: This objective function minimizes the delta model error (maximizes the fit of the delta model) across a sequence of test inputs.

Profiling: The `Grendel` runtime is able to profile calibrated blocks. The profiling operation tests the block with 100-200 inputs and records the bias (error) and noise of the measured output. This profiling information is used to build a model that describes the behavior of the block.

4.3 Experiment Driver

The experiment driver (`exp_driver.py`) performs batch execution and analysis on the analog computer. It requires the Sigilent oscilloscope be used. The experiment driver has three functions:

- **Mass Execution**: The run routine executes any outstanding `grendel` scripts (grendel scripts that don't have outputs) on the analog device.
- **Analysis**: The analysis routine applies the transform computed by the compiler to the measured waveform to recover the original dynamics. It then compares the recovered waveform with the expected dynamics (computed via digital simulation). The analyzer also computes the wall-clock time and power required to execute the simulation.
- **Visualization**: The visualize routine is able to emit a variety of statistical summaries for benchmarks. All tables and figures are written to the `PAPER` directory.

4.4 Model Inference

The model inference engine (`model_builder.py`) converts the profiling data collected by the grendel runtime into a symbolic delta model that the `Legno` compiler can use. It infers delta models from all of the profiling data from the state database (`state.db`). The delta models are written to the model database (`model.db`). The model database is stored in the following location:

```
legno-compiler/device-state/model.db
```

The model inference engine optionally emits visualizations and text descriptions of the models, which can be found in the location below:

```
legno-compiler/device-state/models
```

The models in the database are used by `LScale` and `SrcGen` to correct for deviations in block behavior at compile time.

Chapter 5

LGraph Compilation Pass

The following section describes how to run the LGraph compilation pass.

5.1 Example: Synthesizing Circuits for the Cosine Function

The Legno compiler first generates an unscaled analog device program that implements the cos benchmark. An analog device program (.adp) consists of a set of block configurations and digitally programmable connections to write to the device. This program is unscaled, meaning that the parameters have not been changed so that dynamical system operates within the constraints of the device. We generate the analog device program for the cos benchmark with the following command:

```
python3 legno.py --subset extended lgraph cos --abs-circuits 5
    --conc-circuits 5 --max-circuits 1
```

This step of the compilation process is called the LGraph compilation pass. The --abs-circuits and --conc-circuits parameters control the search space explored by LGraph compilation pass. Specifying higher numbers to these flags produces more analog device programs. The --subset flag specifies what subset of features to use. We choose the extended subset because it includes the broadest subset of tested features.

For the cos dynamical system with the extended set of features, all unscaled circuits are stored in the following directory:

```
legno-compiler/outputs/legno/extended/cos/lgraph-adp
```

The LGraph command presented above produces exactly one unscaled analog device program:

```
cos_g0x0.adp
```

Since the analog device program is not human readable, the compiler also produces a graph that describes the analog device program.

```
outputs/legno/extended/cos/lgraph-diag/cos_g0x0.png
```

5.2 LGraph Command Reference

The following section is a command reference for the `LGraph` command.

- `--help`: show documentation for the command.
- `--simulate`: ignore resource constraints. This is currently not supported.
- `--xforms`: Number of transforms to apply. A transform is a rewrite rule that may be applied to a differential equation. This is currently not supported.
- `--abs-circuits`: number of abstract circuits to generate. The first step of graph synthesis involves generating abstract circuits.
- `--conc-circuits`: number of concrete circuits to generate per abstract circuits.
- `--max-circuits`: maximum number of circuits to generate.

5.2.1 .adp Naming Convention

`LGraph` generates analog device programs with the following naming convention:

```
outputs/legno/<subset>/<prog>/lgraph-adp/{<prog>_g<abs>x<conc>.png
```

The `abs` and `conc` fields are the numerical identifier of the abstract and the concrete circuit. Together, these form a unique identifier for the circuit.

Chapter 6

LScale Compilation Pass

This section describes how to use the lscale compilation pass.

6.1 Example: Cosine Function

Next, we direct the Legno compiler to scale each unscaled analog device program in the abs-circ directory:

```
python3 legno.py --subset extended lscale cos --search \
   --model naive-max_fit --scale-circuits 1
```

This step of the compilation process is called the LScale compilation pass. The --subset flag indicates what subset of device features to use. The --model argument indicates models should be used when scaling the system. Legno supports four different modeling schemes, some of which involve delta models. A delta model is an empirically derived model that describes the physical behavior of the hardware. Please refer to Section **??** for information on how to use delta models with LScale:

- delta-max_fit: Scale the system using delta models derived from block behavior, when the block is calibrated with the max_fit strategy. The max_fit calibration strategy maximizes the chances a delta model can be fit to account for deviations in block behavior.
- delta-min_error: Scale the system using delta models derived from block behavior, when the block is calibrated with the min_error strategy. The min_error calibration strategy minimizes the error of each block.
- naive-max_fit and naive-min_error: Scale the system without delta models. With these schemes, the block behavior is as described in the specification. These is no behavioral difference in these models; there are two models for implementation reasons.

We do not have any delta models for the chip yet, so we will use the naive-max_fit configuration. The naive_maxfit model assumes each block delivers its expected behavior with no deviations. The --scale-circuits parameter determines how many scaled programs to produce

from each unscaled program. Finally the `--search` parameter tells `Legno` to search for the scaling transform that produces the best signal-to-noise ratio.

For the `cos` dynamical system with the `extended` set of features, the resulting scaled programs are written to the following directory:

```
legno-compiler/outputs/legno/extended/cos/lscale-adp/
```

The `LScale` execution presented above produces exactly one scaled analog device program:

```
cos_g0x0_s0_ngd3.00a3.77v1.77c97.00_obsfast.adp
```

Since the analog device program is not human readable, the compiler also produces graphs that visually depict the circuit each program implements. These graphs are stored in the following directory:

```
legno-compiler/outputs/legno/extended/cos/lscale-adp/
```

Chapter 7

Generating and Executing Grendel Scripts

7.1 Example: Cosine Program

7.1.1 Anatomy of a Grendel Script

We present the `.grendel` script below generated for the cosine program below:

```
micro_reset
micro_use_osc
osc_set_volt_range 0 0.102000 1.310000
osc_set_sim_time 2.063e-03
osc_set_volt_range 1 0.102000 1.310000
osc_set_sim_time 2.063e-03
micro_set_sim_time 1.587e-03
micro_use_chip
use_integ 0 3 0 sgn + val 0.0 rng h m debug
use_integ 0 3 1 sgn + val 0.9765625 rng h m debug
use_fanout 0 3 0 0 sgn + - + rng m two
mkconn fanout 0 3 0 0 port 0 tile_output 0 3 0 0
mkconn tile_output 0 3 0 0 chip_output 0 3 2
mkconn fanout 0 3 0 0 port 1 integ 0 3 0
mkconn integ 0 3 1 fanout 0 3 0 0
mkconn integ 0 3 0 integ 0 3 1
osc_setup_trigger
micro_run
osc_get_values differential 0 1 Position outputs/legno/extended/cos/out-waveform/<file>.json
get_integ_status 0 3 0
get_integ_status 0 3 1
micro_get_status
disable integ 0 3 0
```

```
disable integ 0 3 1
disable fanout 0 3 0 0
rmconn fanout 0 3 0 0 port 0 tile_output 0 3 0 0
rmconn tile_output 0 3 0 0 chip_output 0 3 2
rmconn fanout 0 3 0 0 port 1 integ 0 3 0
rmconn integ 0 3 1 fanout 0 3 0 0
rmconn integ 0 3 0 integ 0 3 1
```

Next we walk through the parts of the grendel script step-by-step:

```
micro_reset
micro_use_osc
```

the `micro_reset` resets the state of the micro-controller (not the analog device). The `micro_use_osc` command tells the microcontroller to emit a trigger signal on the `SDA1` pin right before it begins executing the program:

```
osc_set_volt_range 0 0.102000 1.310000
osc_set_sim_time 2.063e-03
osc_set_volt_range 1 0.102000 1.310000
osc_set_sim_time 2.063e-03
```

the `osc_set_sim_time` command sends the amount of time the oscilloscope should record for. This is automatically set by the compiler to be slightly more than the scaled simulation time. The `osc_set_volt_range` is the voltage range the oscilloscope should monitor. These quantities are converted to voltage and time divisions by the oscilloscope driver.

```
micro_set_sim_time 1.587e-03
micro_use_chip
```

the `micro_set_sim_time` command tells the microcontroller how long to wait for after triggering the simulation. The `micro_use_chip` command tells the microcontroller that the analog chip will be used by the program.

```
use_integ 0 3 0 sgn + val 0.0 rng h m debug
use_integ 0 3 1 sgn + val 0.9765625 rng h m debug
use_fanout 0 3 0 0 sgn + - + rng m two
```

the `use_integ` and `use_fanout` blocks configure the blocks in the circuit. The first `use_integ` command sets the input of the integrator to high (`h`) and the output of the integrator to medium (`m`). The `debug` flag tells the integrator to watch out for overflows. The output is noninverted (`sgn +`) and the initial condition is 0.0. The fanout is set to be in medium mode (`m`). The second current copy is negated (`+ - +`).

Use Commands and Calibration/Profiling: The calibration and profiling routines find all the use commands in the script and generate calibration or profiling commands for each use command. Both calibration and profiling are performed for each block configuration.

```
mkconn fanout 0 3 0 0 port 0 tile_output 0 3 0 0
mkconn tile_output 0 3 0 0 chip_output 0 3 2
mkconn fanout 0 3 0 0 port 1 integ 0 3 0
mkconn integ 0 3 1 fanout 0 3 0 0
mkconn integ 0 3 0 integ 0 3 1
```

the `mkconn` command programs the required connections for integrating the circuit.

```
osc_setup_trigger
micro_run
osc_get_values differential 0 1 Position outputs/legno/extended/cos/out-waveform/<file>.json
get_integ_status 0 3 0
get_integ_status 0 3 1
micro_get_status
```

the `osc_setup_trigger` command sets up the trigger on the oscilloscope. The `micro_run` command runs the program. The `osc_get_values` command retrieves the data from the oscilloscope. The waveform is computed by taking the differential signal between channel 0 and 1, and is named `Position`. For brevity we omit the complete path to the `.json` file. Finally, the script retrieves the overflow status of the integrators and the status of the microcontroller.

The Output Waveform: All of the post-processing and analysis routines process the output waveform produced by the `osc_get_values` command. The existence of this output waveform is used to determine if the script has been executed.

```
disable integ 0 3 0
disable integ 0 3 1
disable fanout 0 3 0 0
rmconn fanout 0 3 0 0 port 0 tile_output 0 3 0 0
rmconn tile_output 0 3 0 0 chip_output 0 3 2
rmconn fanout 0 3 0 0 port 1 integ 0 3 0
rmconn integ 0 3 1 fanout 0 3 0 0
rmconn integ 0 3 0 integ 0 3 1
```

The `disable` and `rmconn` teardown the program circuit by removing programmed connections and disabling used blocks.

7.1.2 Generating Grendel Scripts with `Legno`

For each scaled analog device program, legno generates a low-level `Grendel` script that executes the experiment on the analog hardware.

```
python3 legno.py --subset extended srcgen cos --hwenv osc --trials 1
```

This command generates a `Grendel` script for each scaled analog device program. The `--hwenv` parameter specifies the surrounding hardware environment (e.g. oscilloscope). Since we're using the Sigilent 1020XE oscilloscope, we use `--hwenv osc`, which contains the channel configuration described in section **??**.

no oscilloscope: If you're not using a supported measurement device and want to discard the voltage and time information, use `--hwenv noosc` – this will still generate a trigger signal, but will not generate the commands to setup the measurement device. Note these commands can always be ignored during execution using the `--no-oscilloscope` flag.

For the `cos` dynamical system with the `extended` set of features, all the `Grendel` scripts are stored in the following directory:

```
outputs/legno/extended/cos/grendel
```

Since the `cos` benchmark only has one scaled circuit, only one `Grendel` script is produced:

```
cos_g0x0_s0_ngd3.00a3.77v1.77c97.00_obsfast_t20_osc.grendel
```

We will refer to this script as `<file>.grendel` for the remainder of this example.

7.1.3 Calibrating Blocks in the Grendel Script

The grendel runtime (`grendel.py`) executes `.grendel` scripts generated by the `Legno` compiler. Before executing the script, we must calibrate any uncalibrated blocks. The `--calib-obj` argument specifies which calibration objective to use (`min_error` or `max_fit`).

```
python3 grendel.py calibrate --calib-obj max_fit \
  outputs/legno/extended/cos/grendel/<file>.grendel
```

The calibration information is stored in `device-state/state.db`, and reused for subsequent executions. Any blocks that already have calibration information in `state.db` are automatically skipped. If you wish to recalibrate these blocks, use the `--recompute` flag.
No Oscilloscope: If you are not using the Sigilent 1020XE oscilloscope, include the `--no-oscilloscope` flag to prevent the runtime from attempting to communicate with the measurement device.

7.1.4 Executing the Grendel Script

We can run the benchmark after all the block have been calibrated. Executing the following command to executes the program on the analog device:

```
python3 grendel.py run --calib-obj max_fit \
outputs/legno/extended/cos/grendel/<file>.grendel
```

This should write the configuration to the device, and execute the cosine function for 20 simulation units.
No Oscilloscope: If you are not using the Sigilent 1020XE oscilloscope, include the `--no-oscilloscope` flag to prevent the runtime from attempting to communicate with the measurement device.

Bibliography