Agile Hardware Development and Instrumentation With PyRTL

Deeksha Dangwal, Georgios Tzimpragos, and Timothy Sherwood
University of California, Santa Barbara

***Abstract*—Domain-specific architectures have emerged as a promising solution to meet growing technology demands but with this comes an urgent need to improve hardware methodologies which often have long design cycles, rely on closed source and expensive tools, and have high nonrecurring engineering costs. In this article, we describe how our work developing PyRTL, an open source Python-based Hardware Development Toolkit, has proven to be a powerful agile hardware development and analysis tool with the features to improve current methodologies. We describe how this toolkit-driven approach encourages hardware reuse using modern object-oriented programming features and present an examination of its custom intermediate representation for hardware debugging, analysis, and instrumentation. This approach has proven useful in supporting fast design iteration in a variety of domains including cryptography and machine learning.**

The recent growth in specialized hardware and accelerators as a means to improve performance and energy efficiency has created a development flow mismatch between the rapid cycles expected from software and the more top-down waterfall style of development dominating traditional hardware design. If we are to support agile computational development in this mixed hardware/software world, we will need the ability to prototype, test, and characterize new designs in a timely manner. Of special note is reconfigurable hardware, in the form of either FPGAs or coarse grain reconfigurable arrays, which further open the door to agile development by allowing early roll-out designs to be improved and updated incrementally. However, we observe that most modern agile development models rely heavily on

Digital Object Identifier 10.1109/MM.2020.2997704
Date of publication 26 May 2020; date of current version 30 June 2020.

a rich ecosystem of tools, libraries, and services that can be customized and deeply integrated in the process. That ecosystem, when integrated well, enables continuous improvement in the *entire process* of development. Critically, for both hardware and software, this includes not only design, but also test, analysis, evaluation, and how it integrates with existing development tools. Rather than invent new languages and practices that mirror those in agile development, an important goal of PyRTL is to act as a *bridge* between these traditionally distinct worlds.

PyRTL,[1] a Python Hardware Design Toolkit, provides a pathway to concisely and precisely design hardware structures. PyRTL's overarching goals are simplicity, usability, clarity, and extensibility. With this in mind, the toolkit is developed around a small and well-defined internal core structure which intentionally restricts users to a set of reasonable digital design practices that always lead to a synthesizable design. While we have extensively used it over the past six years in the development of numerous processors and accelerators, PyRTL does not claim to solve every hardware development problem out of the box. Rather the core of PyRTL provides a minimal set of hardware primitives, expressed as a Python class, which can then be extended with other classes and libraries as appropriate. PyRTL enables the use of higher order functions, lambdas, list comprehension (see Figure 1), recursion (see Figure 2), and is supported by Python's rich ecosystem of libraries. For example, we have previously integrated our various PyRTL designs generated with Python packages such as TensorFlow, PyTorch, Scikit-Learn, Numpy, Hypothesis testing, Z3 SMT solver, and more. This, in turn, allows for a property of a hardware design to be checked through with a solver, an optimization to be performed by exploiting existing libraries, or a set of parameters to be shared between a software model and hardware design easily and without the need for new stand-alone tools or even special CAD skills. As we will discuss later, we have supported some of the more common patterns, such as a "hardware transformation" and complimentary libraries to PyRTL to even further lower the barrier to experimentation and tooling.

The core philosophy of PyRTL is elaboration-through-execution; i.e., to generate hardware as

```
# AES SubBytes
def sub_bytes(in_vector):
    subbed = [sbox[byte] for byte in
     ↪  partition_wire(in_vector, 8)]
    return pyrtl.concat(*subbed)

# AES InverseMixColumns
def inv_mix_columns(in_vector):
    igm_divisor = [14, 11, 13, 9]
    def inv_mix_single(index):
        mult_items =
         ↪  [inv_galois_mult(a[mod_add(index, loc,
         ↪  4)], mult_table) for loc, mult_table in
         ↪  enumerate(igm_divisor)]
        return mult_items[0] ^ mult_items[1] ^
         ↪  mult_itmes[2] ^ mult_items[3]

    inverted = [inv_mix_single(index) for index in
     ↪  range(15)]
    return pyrtl.concat(*inverted)
```

Figure 1. Use of hardware comprehension for concise code to implement the data scrambling operation, SubBytes and data unscrambling operation, InverseMixColumns, in AES.

PyRTL code is being executed, an approach it shares with the hardware construction languages Lava[2] and Chisel.[3] PyRTL does *not* infer hardware from a restricted set of Python. Instead it maintains an internal map of core components and, as the PyRTL code executes, it grows that map in the way specified by the software. The python code is in some sense a single "generator" for the hardware, and software abstractions can be used to build arbitrarily large and complex complete systems. Small generators (such a FIFO queue generator function) can then be called by larger generators (such as a Pipeline generator class). Because all of these generators ultimately call PyRTL core elements under the hood (visible to the designer), the hardware designed is always explainable and synthesizable.

PyRTL comes with "batteries included" and students/designers can go from writing tests to designing hardware blocks to visualizing waveforms without ever having to leave the Python shell. A single "pip install" is all that is needed to get going. PyRTL includes classes to support design, simulation, debugging, visualization, synthesis, testing, tracing, and instrumentation and the goal is to allow for the quick design/test cycles all in Python. Of course there are many other hardware design tools one might wish to bring into the mix and so all hardware described in PyRTL, by nature of building around a small and well-defined core, can be exported to either synthesizable Verilog or FIRRTL – Chisel's[3] intermediate representation (IR). Testbenches can be exported to Verilog and simulation results can be visualized with almost any standard waveform viewing tool.

In this article, our goal is to describe our experiences in using PyRTL in various research projects and highlight the features that make it especially suitable for agile development. We first explain a typical PyRTL workflow and then go through the core and IR that make that possible. We present a number of interesting examples where PyRTL allows us to tightly integrate the hardware design process with software libraries and evaluate resulting designs with detailed analyses.

Of course, PyRTL draws on a wealth of existing work in modern hardware design DSLs. Chisel[3] presents elaboration-through-execution in Scala and is a popular and powerful hardware construction language used in many RISC-V projects. MyHDL[4] utilizes Python's decorators and generators while maintaining syntax close to Verilog. PyMTL[5] and Mamba[6] enable generation, verification, and simulation of hardware at multiple levels. Other similar projects built on top of pure functional languages are CλaSH[7] and Lava.[2]

With the rise of domain specific architectures, there is more demand than ever for domain experts to play a role in making informed algorithm/hardware tradeoffs. In order to enable hardware–software co-design, PyRTL presents algorithm developers with flexibility, high-level abstractions, and the support of well-loved software libraries while still maintaining low-level control of details of hardware design decisions

PYRTL FOR AGILE HARDWARE DEVELOPMENT

With the rise of domain specific architectures, there is more demand than ever for domain experts to play a role in making informed algorithm/hardware tradeoffs. In order to enable hardware–software co-design, PyRTL presents algorithm developers with flexibility, high-level abstractions, and the support of well-loved software libraries while still maintaining low-level control of details of hardware design decisions. In this section, we detail how we have mixed-in code from popular python libraries, such as PyTorch, Scikit-Learn, and Hypothesis, in our accelerator hardware design flows.

PyRTL Workflow

To understand how this works, it is helpful to understand what it even means to design hardware in PyRTL. Getting started really is as simple as typing `pip install pyrtl` in your terminal and the python shell will suffice to go from design to debugging, simulation, and synthesis. No other libraries are required, although it does have built in connections to other hardware frameworks. For example, PyRTL can provide area and timing estimates, either through internal estimators or by making calls out to yosys.[8] Consider the toy example of building an adder in PyRTL shown in Figure 2. Once the functions for the full adder and ripple-carry adder are defined, they can be "wired" using PyRTL's wire types (see the "PyRTL'S Core and Internals" section) and simulated with the waveforms displayed on the terminal. Because definition and simulation all happen in the same execution, it is easy for standard agile testing approaches to apply. After initial test-driven development the designs can be exported automatically to Verilog for use with traditional hardware workflows. While PyRTL's core and primitives (see the "PyRTL'S Core and Internals" section) can be used reuse hardware beyond simply "modules," PyRTL's real strength is being able to build hardware *alongside* existing software libraries.

Developing and Verifying With Scikit-Learn

In the case of Race Trees,[9] the authors use PyRTL to develop a template-based methodology, integrated with Scikit-Learn, that allows for the design and evaluation of scalable temporal accelerators for ensembles of decision trees with little effort. The training of the model on inputs and design of the accelerators happen in the same Python code base. More specifically, once the training process completes, the tool uses a set of basic building blocks (e.g., parametric shift register and buffer, INHIBIT gate, etc.) and glue logic to automatically generate synthesizable RTL code out of a simple graph-based IR,

```
import pyrtl
import random

def fa(x, y, cin):
'''A full adder'''
    sum = x ^ y ^ cin
    cout = x&y | y&cin | x&cin
    return sum, cout

def adder(a, b, cin):
'''A ripple-carry adder with carry in and out'''
    a, b = match_bitwidth(a, b)
    n = len(a)
    sum = {}
    for i in range(n):
        sum[i], cout = fa(a[i], b[i], cin)
        cin = cout
    full_sum = concat_list([sum[i] for i in
      ↪  range(n)])
    return full_sum, cout

# define pyrtl inputs and outputs
a, b, cin = pyrtl.input_list('a/5 b/5 cin/1')
full_sum, cout = pyrtl.output_list('full_sum/5
 ↪  cout/1')
sum_t, cout_t = adder(a, b, cin)
full_sum <<= sum_t
cout <<= cout_t

# simulate the design
sim = Simulation()
for cycle in range(10):
    sim.step ({
        'a': random.randrange(32)) ,
        'b': random.randrange(2**len(b))) ,
        'cin': random.randrange(2)
    })
    answer = sum([sim.inspect(i) for i in 'a b
      ↪  cin'.split()])
    assert pyrtl.truncate(answer, 5) ==
      ↪  sim.inspect('full_sum')

sim_trace.render_trace()
```

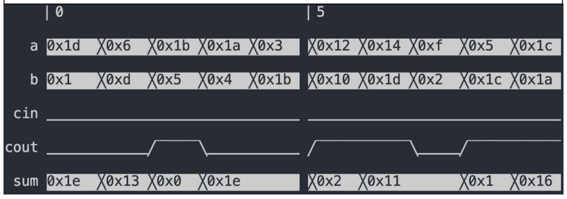

Figure 2. This example shows the entirely of code required to implement and test a ripple carry adder. Just as an example, a Python dictionary keeps track of the wires carrying the sum bits as we iterate through. The design is simulated on random values over full span of inputs and a standard python assertion is added to check for errors in the sum output.

used by Scikit-Learn to store the trained model. During functional verification, when the PyRTL code for the given model is ready, the predict function of the Scikit-Learn library can be used as the golden reference to perform cross-checking. To get the value of any of the hardware design's variables in the last simulation cycle PyRTL's inspect function can be used. Moreover, given that Scikit-Learn does not provide/expect signals in the temporal domain, while the developed hardware does, the authors use Pythons generators to properly simulate input stimuli. Existing ML evaluation and training approaches can be directly employed in the evaluation of hardware under test.

Prototyping ML Accelerators With PyTorch

In addition to providing useful input/output evaluation of designs, PyRTL allows tools to be integrated into the design process itself, for example in the search for good parameterizations of hardware generation for ML accelerators. One embodiment of this idea is PyRTLMatrix[10]: a general purpose hardware design pattern for instantiating and composing common neural network primitives. The PyRTLMatrix class contains hardware implementations of matrix operations that are at the heart of ML applications. An instance of PyRTLMatrix represents a wire bus that directs data through various logical computations representing matrix operations. Consider Figure 3(a), which shows the forward function (inference step) in both PyTorch and PyRTLMatrix. Using the PyRTLMatrix class, we are able to design hardware that looks as simple as high-level software. But, this is not without tradeoffs. As a direct result of using the PyRTLMatrix class, designers are distanced from the hardware implementation which can lead to a loss of low-level understanding. Low-level effects such as locally inferring *bitwidth* of operations repeatedly can lead to a suboptimal designs because of the cumulative growth the inferred bitwidth of wires. Using block matrix multiplications, we end up with a tree-like hardware structure and due to the cascading of a series of element-wise vector adders, there is an accumulation of wires which leads to suboptimal design. To combat this and ensure a consistent bitwidth throughout, the students developing that code were able to wrap functions with a custom Python decorator pattern as shown in Figure 3(b).

Agile Testing With Hypothesis

Because the simulation of hardware can happen right in the same execution as the elaboration, like other elaboration through execution techniques, we can leverage agile testing approaches. A specific advantage of PyRTL's direct implementation (rather than relying on

PyTorch Forward Function

```
def forward(self, x):
  out = x.view(BATCH_SIZE, -1)
  for i in range(len(self.weights)):
    out = self.weights[i](out)
    out = self.relu(out, threshold)
  return out
```

PyRTL Forward Function

```
def forward(self, x):
  out = x
  for i in range(len(self.weights)):
    out = self.weights[i].toMatrix()@out
    out = out + self.bias[i].toMatrix()
    out = self.relu(out, int(threshold))
  return matrix.argmax(out)
```

(a)

```
@property
def bits(self):
  ''' Gets the number of bits
  :return: the number of bits.'''
  return self._bits

@bits.setter
def bits(self, bits):
  ''' Sets the number of bits.
  :param int bits: The number of bits.'''
  self._bits = bits
  for i in range(self.rows):
    for j in range(self.columns):
      self[i, j].bitwidth = self._bits
```

(b)

Figure 3. (a) Forward function implemented within PyTorch and PyRTL. Using the PyRTLMatrix class, the PyRTL code becomes simpler despite integrating hardware designs. Through the application of software design patterns, hardware design languages become easier to understand.
(b) Decorator pattern is used to ensure a consistent bitwidth among all items in the PyRTLMatrix.

```
f = fifo(4, in_valid, in_ready, in_data, out_valid,
         out_ready, out_data)

@hypothesis.given(pyrtl.fulltrace('in_valid in_data
 ↪ out_ready'))
def test_fifo(siminput_list):
    sim = pyrtl.Simulation()
    ref = collections.deque()
    for siminput in siminput_list:
        sim.step(siminput._asdict())
        if sim.inspect('in_ready') and
         ↪ sim.inspect('in_valid'):
            # something into the fifo
            ref.append(sim.inspect('in_data'))
        if sim.inspect('out_valid') and
         ↪ sim.inspect('out_ready'):
            # check same thing coming out
            data = ref.popleft()
            assert data == sim.inspect('out_data')

test_fifo()
```

Figure 4. Example of testing in PyRTL. The function test_fifo is a simple unit test of a style familiar to agile software developers, but in this case it is testing a hardware implementation of a FIFO against the python deque queue. The standard Python package Hypothesis is used to automatically generate and then minimize stateful failing inputs.

higher level operations like decorators for core operation) is that such integration is surprisingly straightforward. As pointed out by others,[5] Hypothesis is a python library for property-based testing which generates data matching a provided specification of a property to test whether a guarantee holds. Importantly, the package seeks to *automatically simplify* failure cases to the smallest possible failing input—a feature that is incredibly useful in hardware unit testing. While the earlier ripple-carry example showed direct testing of combinational logic, any nontrivial hardware has state to manage. In Figure 4, you can see the entirety of the code required to perform testing on a hardware FIFO implementation. Omitted is the function fifo, which defines the hardware implementation of a fifo. The base function test_fifo takes a list of input signals and simulates them, raising an assert when an error is found. Hypothesis is simply instructed to find inputs of the requested type that cause the assert to fail. The resulting function test_fifo does exactly that. With a couple extra lines of the code, not shown, one can easily extract the minimum resulting failure. This approach actually found a subtle bug in the original FIFO implementation that only happened when the queue was in its full state and a certain combination of ready and valid arrive. While many traces are generated as part of the testing, the first trace generated by hypothesis that caused a failing assertion was over 40 cycles long. However, before returning this trace it was able to reduce the example to a truly minimum exciting input that immediately filled up the queue and made the failing behavior trivial to identify. Failing traces can be added to a regression so that future changes do not change the behavior of these identified corner cases. While more holistic testing will always be required for hardware, the test-driven style common in agile certainly makes sense for even complex components.

PYRTL'S CORE AND INTERNALS

At the heart of PyRTL lies an IR that provides a complete set of operations and structures for the description and manipulation of hardware. PyRTL's IR supports a reasonable set of hardware design practices but is not intended to allow full arbitrary hardware to be specified. For example,

Table 1. Synthesis results comparing area, delay, gate count, and LOC of PyRTL and Verilog designs.

Microbenchmark	PyRTL				Verilog			
	Area	Delay	Gate count	LOC	Area	Delay	Gate count	LOC
Wallace tree multiply	223.52	891.61	70	31	219.29	1129.04	70	115
Matrix multiply	4176.3	1647.66	1298	19	4176.3	1647.66	1298	33
Gray counter	101.38	348.98	30	8	48.36	276.45	15	27
Mealy machine	65.33	363.31	25	37	47.46	266.8	20	110
AES128 sub_bytes	30652.17	788.25	11114	31	76380.47	905.76	27916	607
AES128 mix_columns	2471.63	339.78	527	19	2213.68	339.78	472	42

recurrent hardware (where logic loops back on itself without traversing a state element) and semianalog designs (for example Z-state or wired-or logic) are not allowed. While these structures correspond to valid hardware, most modern high performance design practices avoid them. By taking away some control in this way, we can replace it with an intermediate structure that includes a complete and understandable tool chain. Knowledge of PyRTL's IR is not strictly necessary for the development of hardware design, but due to its small size it is very handy for understanding hardware design basics for beginners and for performing instrumentation and the translation of PyRTL designs into other hardware description languages for more advanced users.

Blocks—Within PyRTL, sets of operations are grouped in Blocks, each containing well-defined inputs and outputs. Blocks represent the full bipartite graph between logic elements (the primitive operations) and WireVectors (which connect these elements together). Blocks allow the design of multiple circuits at once and enable grouping of related hardware components together.

WireVectors—PyRTL supports five different wire types: WireVector, Input, Output, Const, and Register. Input and Output are special wires that represent dynamic inputs and outputs in the circuit, Const wires represent fixed values in the circuit, and Registers store the value from its source for the next cycle.

Memories—As memories are one of the most critical elements of hardware designs and in PyRTL they have their own construct to represent them. The memories are declared with a size and bitwidth but are exposed to the user as a collection of ports. Each array access or array assignment corresponds to a specific read or write port. PyRTL defaults to allowing 2-read 1-write memories unless such memories are specifically declared to have a greater number.

One of the strongest aspects of PyRTL is the aggressive and useful checks the system provides in the hardware design process. Python is often mischaracterized as "weakly typed" when in fact it is "strongly typed" but just dynamically typed. This means at execution time every type in Python is well defined. Because the elaboration of the design happens at run time (i.e., when the generator runs) there is never any question about the types used in generating hardware. This allows us to build a robust set of elaboration-time checks that identify errors such as mismatched bitwidths or unexpected conversions. However, some properties, such as that all wires in the system must be driven by a source, have to be checked postelaboration. Properties such as this are checked by the sanity_check method when the circuit is completed to ensure the working model of hardware is valid at all times. This further ensures that user-defined transforms never violate the IR semantics and never create invalid hardware states. For a detailed explanation of PyRTL's core and operations, please refer to our FPL2017 paper.[1]

While PyRTL is useful for rapid prototyping and design tradeoff evaluation, some confidence that resulting designs are not terribly inefficient is important. To that end, we present results from several microbenchmarks as compared on area, delay, and gate counts for designs in both PyRTL and Verilog in Table 1. For reference we also present the number of lines of code (LOC) used.

```
def glift_tracking():
    glift_ws, glift_mems = {}, {}
    for wire in tuple(block.wirevector_set):
        # ... (preprocessing of wires)
    _build_net_glift(glift_ws, glift_mems)
    for mem in glift_mems.values():
        mem.ga.next <<= or_all_bits(mem.gwa)
    return glift_ws

@transform.all_nets
def _build_net_glift(net, g_w, g_mems):
    or_all = pyrtl.or_all_bits
    if net.op in "+-*cs><=w~^":
        g_w[net.dests[0]] <<= \
            or_all(tuple(g_w[a] for a in net.args))
    elif net.op == 'r':
        g_w[net.dests[0]].next <<= g_w[net.args[0]]
    elif net.op in '&n':
        a0, a1 = net.args
        both_g = g_w[a0] & g_w[a1]
        a0_g = g_w[a0] & or_all(a1)
        a1_g = g_w[a1] & or_all(a0)
        g_w[net.dests[0]] <<= both_g | a0_g | a1_g
    elif net.op == '|':
        a0, a1 = net.args
        both_g = g_w[a0] & g_w[a1]
        a0_g = g_w[a0] & (~ pyrtl.and_all_bits(a1))
        a1_g = g_w[a1] & (~ pyrtl.and_all_bits(a0))
        g_w[net.dests[0]] <<= both_g | a0_g | a1_g
    # ... (Mux and memories processing)
```

Figure 5. Implementation of GLIFT in PyRTL's instrumentation framework.

We attempt to maintain an apples-to-apples comparison between the hardware designs by using yosys to synthesize both sets of designs and find that, in most cases, PyRTL designs have comparable resulting delay and gate counts to Verilog.

HARDWARE INSTRUMENTATION AND ANALYSIS

The move to more integrated hardware acceleration logic, both programmable and of fixed functionality, makes the task of understanding systems holistically more complicated. While binary instrumentation frameworks such as Pin, Valgrind, and Atom allow the rapid development of tools for instrumenting software, it is more complex to build similar frameworks with functionality split between the worlds of software and application-specific hardware. Our instrumentation framework provides an interface that simplifies walking, augmenting, and modifying reconfigurable hardware designs. For example, one might wish to count the number of times a hardware event is triggered, access the frequency of particular memory address, or perform complete hardware-level information flow tracking. In any of these cases, PyRTL's instrumentation framework allows development of the tools in a few hundred LOC, yet the performance of the resulting tools allows for the instrumentation of designs of a few millions of gates in only a few seconds.

Gate-Level Information Flow Tracking

Before we describe our framework for instrumentation and transforms, we present a sample tool for gate-level information flow tracking (GLIFT).[11] Information flow analysis is pivotal to evaluate security properties of programs and processes. We can guarantee data integrity by verifying that particular values were never affected by outside signals, and ensure isolation by verifying that values never leak to the outside world. This is often accomplished by tracking the "taintedness" of values to check for the existence of undesired flows (untrusted to trusted, secure to insecure, etc.). However, information leakage of processor state may not be detectable at the ISA level. GLIFT extends information flow analyses to hardware and demonstrates how additional logic can be added to an existing design to track the information flow at the level of individual logic gates. Such analysis at design time can be extremely valuable for implementing secure processors, and other hardware security modules.

Using PyRTL's instrumentation framework, GLIFT analysis can be implemented such that, for each logic element additional "shadow" hardware is added to track its "taintedness" and compute whether the result of the wire should be considered tainted as well. Adding the additional GLIFT logic has little performance impact so long as it fits in the available LUTs. By building GLIFT as an instrument, we present the designer with freedom to decide how to access data, what data to access, whether custom inputs and outputs in the hardware circuitry are considered, whether special instructions are considered, etc. The resulting GLIFT implementation takes up only 72 LOC (Figure 5 shows an excerpt of the transform that instruments a circuit with full GLIFT monitoring). Even though LOC is an imperfect metric of the programmer effort needed to design such hardware, it still demonstrates that these instruments can be built with ease.

Instrumentation API

Raw manipulations of the internal representation are tedious and error-prone. Therefore, PyRTL's instrumentation API is the cornerstone to making the instrumentation platform accessible. These API calls collect commonly needed information and perform common modifications to the hardware block.

Data-Flow Respecting Iterator for Topological Sorting It is sometimes useful to have a well-defined data-flow preserving iteration order for circuit elements. Without such an ordering, much additional verification would be required. PyRTL's instrumentation framework provides a data flow respecting iterator that guarantees a logic operation is only returned after its predecessors (with the exception of registers). This, in turn, guarantees any modifications to predecessors is completed and therefore can be used, prior to modification of any logic operation.

While there is a growing community of tools and techniques developing in this space, we offer PyRTL as solution to improve productivity of hardware designers by providing high-level abstractions, increased opportunity for design reuse, tight integration with existing software libraries, and an instrumentation infrastructure for analysis.

Net Connections While LogicNets, which represent operations, store which wires are connected to them, the wires do not store which nets they are connected to. In order to improve performance and eliminate this confusion, we built the function net_connections. It returns a dictionary that, for each wire, notes which net is its source and which nets use the wire. With this information and the information contained in the nets themselves, an instrument is able to efficiently traverse and transform the circuit.

Wire and Logic Replacement Replacing logic and wire elements from the hardware block can vary depending on the application. Adding new logic and wires to a design is trivial, but modifying existing logic is more nuanced, e.g., finding all places where a wire is used and making sure not to iterate over new hardware when transforming existing hardware, etc. We provide two API functions — wire_transform and net_transform — to facilitate replacement. The input to both functions is a mapping of a single wire/net to new wires/nets.

CONCLUSION

While domain-specific accelerators and reconfigurable computing continue to grow in importance, traditional hardware design methodologies will struggle to accommodate the broader participation and rapid development cycles necessary to unlock the true potential of these new architectures. While there is a growing community of tools and techniques developing in this space, we offer PyRTL as solution to improve productivity of hardware designers by providing high-level abstractions, increased opportunity for design reuse, tight integration with existing software libraries, and an instrumentation infrastructure for analysis. Our hope is to work in concert with other wonderful agile hardware tool and language developers to help further open the domain of hardware design to students, software engineers, and enthusiasts and to empower them with tools to build good hardware quickly.

REFERENCES

1. J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood, "A pythonic approach for rapid hardware prototyping and instrumentation," in *Proc. IEEE 27th Int. Conf. Field Program. Logic Appl.*, 2017, pp. 1–7.
2. P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in haskell," in *Proc. ACM SIGPLAN Notices*, 1998, vol. 34, no. 1, pp. 174–184.
3. J. Bachrach *et al.*, "Chisel: constructing hardware in a scala embedded language," in *Proc. IEEE DAC Des. Autom. Conf.*, 2012, pp. 1212–1221.
4. J. Decaluwe, "Myhdl: A python-based hardware description language," *Linux J.*, no. 127, pp. 84–87, 2004.
5. D. Lockhart, G. Zibrat, and C. Batten, "PyMTL: A unified framework for vertically integrated computer architecture research," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2014, pp. 280–292.

6. S. Jiang, B. Ilbeyi, and C. Batten, “Mamba: Closing the performance gap in productive hardware development frameworks,” in *Proc. 55th ACM/ESDA/IEEE Des. Autom. Conf.*, 2018, pp. 1–6.
7. C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, “Cλash: Structural descriptions of synchronous hardware using haskell,” in *Proc. IEEE 13th Euromicro Conf. Digit. Syst. Des., Archit., Methods Tools*, 2010, pp. 714–721.
8. C. Wolf, “Yosys open synthesis suite,” 2016.
9. G. Tzimpragos, A. Madhavan, D. Vasudevan, D. Strukov, and T. Sherwood, “Boosted race trees for low energy classification,” in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2019, pp. 215–228.
10. D. Aboye *et al.*, “PyRTLMatrix: An object-oriented hardware design pattern for prototyping ML accelerators,” in *Proc. 2nd Workshop Energy Efficient Mach. Learn. Cogn. Comput. Embedded Appl.*, 2019, pp. 36–40.
11. M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, “Complete information flow tracking from the gates up,” in *Proc. ACM Sigplan Notices*, 2009, vol. 44, no. 3, pp. 109–120.

Deeksha Dangwal is currently working toward the Ph.D. degree in computer architecture with the Department of Computer Science, University of California, Santa Barbara. Her research interests lie at the intersection of computer architecture, privacy, and information theory. She is a student member of IEEE and ACM. Contact her at deeksha@cs.ucsb.edu.

Georgios Tzimpragos is currently working toward the Ph.D. degree with the Department of Computer Science, University of California, Santa Barbara, and is a Research Affiliate with Lawrence Berkeley National Laboratory. His research interests are broadly in the field of computer architecture. Tzimpragos received the master’s degree in electrical and computer engineering from the University of California, Davis. His alma mater is the National Technical University of Athens in Greece. Contact him at gtzimpragos@cs.ucsb.edu.

Timothy Sherwood is currently a Professor of Computer Science and the Associate Vice Chancellor for Research with the University of California, Santa Barbara. He is a Co-Founder of the hardware security startup Tortuga Logic and the 2016 ACM SIGARCH Maurice Wilkes Awardee “for contributions to novel program analysis advancing architectural modeling and security.” Sherwood received the B.S. degree in computer science from the University of California, Davis, and the M.S. and Ph.D. degrees from the University of California, San Diego. Contact him at sherwood@cs.ucsb.edu.