Appears in the Proceedings of the 37th IEEE Int'l Conf. on Computer Design (ICCD-37), November 2019

PyOCN: A Unified Framework for Modeling, Testing, and Evaluating On-Chip Networks

Cheng Tan, Yanghui Ou, Shunning Jiang, Peitian Pan, Christopher Torng, Shady Agwa, Christopher Batten
School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{ct535, yo96, sj634, pp482, clt67, sr972, cbatten}@cornell.edu

***Abstract*—There is a growing interest in the open-source hardware movement to amortize non-recurring engineering costs by using plug-and-play system-on-chip (SoC) designs, where the communication among different components is provided by an on-chip interconnection network. Unfortunately, building an on-chip network (OCN) that is suitable for a specific SoC design requires the exploration of a large number of design options and involves diverse research methodologies to evaluate performance, area, energy, and timing. In this paper, we propose PyOCN, a unified framework that vertically integrates multiple research methodologies to enable productively exploring the OCN design space. PyOCN is the first comprehensive framework for modeling (e.g., functional-level, cycle-level, and register-transfer-level), testing (e.g., unit testing, integration testing, and property-based random testing), and evaluating (e.g., simulating, generating, and characterizing) on-chip interconnection networks. We use a case study based on a 64-terminal butterfly network to illustrate the key features of PyOCN and to demonstrate the framework's potential in productively modeling, testing, and evaluating OCNs.**

I. Introduction

On-chip networks (OCNs) play a significant role in chip design across many different domains. Embedded SoCs can include tens of homogeneous or heterogeneous cores to meet performance and power requirements [18, 41], high-end cloud servers can include tens to hundreds of cores to enable high-performance computing [8, 44], and accelerators can include hundreds of processing elements for domain-specific computing [12, 13, 26, 30]. At the same time, the costs of chip design and verification are rising. In response, there is growing interest in open-source hardware design based on plug-and-play SoC frameworks, where the communication between components is provided by an on-chip interconnection network.

Unfortunately, building an OCN that is suitable for a specific SoC design requires exploring a large design space (e.g., network size, channel bandwidth, topologies, routing algorithms, flow control schemes, arbitration techniques, physical floorplanning, and wire routing) using a combination of high- and low-level modeling to accurately estimate performance, area, energy, and timing. For example, OCN cycle-level simulators are widely used today and provide rich configuration options for early-stage design-space exploration [1, 3, 10, 21, 42]. However, the convenience in using CL models must be balanced against decreased accuracy and no path to real hardware implementations. There are a number of OCN register-transfer-level (RTL) generators that produce synthesizable Verilog to drive an evaluation of area, energy, and timing [11, 15–17, 29, 35]. These low-level generators can be difficult to use and lack support for fast simulation. Some OCN design frameworks combine various research methodologies together to facilitate design space exploration [6, 37]. However, area, energy, and timing characterization in these frameworks is often based on high-level first-order modeling. There is a growing need for a vertically integrated OCN framework that can effectively characterize performance, area, energy, and timing across a large design space.

This paper presents PyOCN, a unified framework for modeling, testing, and evaluating on-chip interconnection networks. The concrete contributions of this work are the following: (1) PyOCN enables multi-level modeling to facilitate rapid design-space exploration and OCN implementation; (2) PyOCN provides sophisticated test harnesses for testing OCN designs modeled at different abstraction levels; (3) PyOCN can simulate OCNs at various abstraction levels, generate synthesizable Verilog, and drive a commercial standard-cell-based toolflow for characterizing OCN area, energy, and timing.

II. Related Work

Table I summarizes the state-of-the-art OCN research methodologies and compares them to PyOCN.

A. Modeling OCNs

Existing state-of-the-art OCN simulators struggle to balance rapid design-space exploration (requiring high-level design abstractions) and accurate estimation of area, energy, and timing (requiring low-level detailed modeling).

CL Modeling – Many widely used on-chip network simulators use CL modeling for early design-space exploration while verifying functional- and cycle-level behavior [1, 3, 10, 21, 31, 42]. Unfortunately, these simulators do not support RTL modeling and cannot easily generate synthesizable Verilog, which is essential for accurate evaluation of area, energy, and timing. As an exception, Noxim [9] is a cycle-level OCN simulator developed in SystemC with some capacity for power estimation. All basic elements of the OCN in Noxim are also modeled in VHDL and are synthesized with a 65 nm CMOS standard cell library at 1GHz to provide statistical power analysis.

RTL Modeling – On the other hand, OCN generators use RTL modeling to accurately characterize area, energy, and timing, but they lack the high-level design abstractions that enable fast design-space exploration [11, 29, 35]. For example, OpenSMART [29] is an OCN RTL generator for a wide range of different network configurations. Unfortunately, simulating generated RTL can easily limit rapid design-space exploration over large parameter space.

PL Modeling – Finally, OCN frameworks rarely take physical-level (PL) modeling considerations into account (e.g., macro- and micro-floorplanning), which is critical for effectively building complex OCNs. One exception is SUNMAP [33], which enables PL modeling in OCN generation and uses a floorplanning algorithm [2] to minimize the estimated area and wire lengths for specific applications.

TABLE I. COMPARISON WITH PRIOR ART

	Framework	Modeling							Testing			Evaluating			Open-source
		Lang.	Topology	Routing	FL	CL	RTL	PL	Unit	Int.	PBT.	Sim.	RTL Gen.	ASIC Char.	
Simulation	**BookSim2 [21]**	C++	Xbar, Ring, (C)Mesh, Butterfly, Torus, Tree	DOR, Customized	○	●	○	○	○	○	○	●	○	○	●
	Garnet [3]	C++, Python	Xbar, Mesh, Customized	DOR, Customized	○	●	○	○	○	●	○	●	○	○	●
	Noxim [9]	SystemC	Mesh, Butterfly, Wireless	DOR, Odd-Even, Dyad routing	○	●	○	○	○	●	○	●	○	●	●
Generation	**FlexNoC [17]**	?	Application-specific	n/a	○	○	●	○	?	?	?	◒	●	○	○
	NoCGEN [11]	HDL	Mesh, Customized topology	DOR routing	○	○	●	○	?	?	?	◒	●	○	○
	Connect [35]	BSV	Customized topology	Customized routing	○	○	●	○	?	?	?	◒	●	○	◒
	OpenPiton [4]	Verilog	Xbar, Mesh	DOR routing	○	○	◒	○	○	●	○	◒	●	●	●
	Netmaker [34]	System-Verilog	Mesh	DOR routing	○	○	◒	○	○	●	○	◒	●	○	●
	OpenSMART [29]	BSV Chisel	Mesh, Customized topology	DOR, Source routing	○	○	●	○	●	●	○	◒	●	○	●
	OpenSoC Fabric [16]	Chisel	Mesh, Flattened butterfly	DOR routing, Concentration	○	●	●	○	●	●	○	◒	●	○	●
Charact.	**DSENT [40]**	C++	n/a	n/a	○	○	○	○	○	○	○	○	○	◒	●
	Orion2.0 [23]	C++	n/a	n/a	○	○	○	○	○	○	○	○	○	◒	●
	COSI [37]	C++	Application-specific	n/a	○	●	●	●	○	○	○	●	●	◒	●
	NetChip [6]	SystemC	Application-specific	n/a	○	●	●	○	?	?	?	●	●	◒	○
	PyOCN	PyMTL	Xbar, Ring, (C)Mesh, Butterfly, Torus, Customized topology	DOR, Source, Customized routing	●	●	●	●	●	●	●	●	●	●	●

Different state-of-the-art research methodologies for designing OCNs, which are categorized into three groups (i.e., Simulation, Generation, and Characterization). ○, ◒, and ● indicate the corresponding feature is not supported, partially supported, and fully supported, respectively. For example, OpenSoC Fabric can generate synthesizable Verilog (●) but relies on VCS for simulation (◒). In contrast, the simulation in PyOCN allows the test bench to be written in Python and eliminates any semantic gap. Lang. = language; FL = functional level; CL = cycle level; RTL = register-transfer level; PL = physical level; Unit = unit testing; Int. = integration testing; PBT. = property-based random testing; Sim. = simulation; RTL Gen. = RTL generation; ASIC Char. = ASIC characterization.

B. Testing OCNs

Debugging OCNs can be time-consuming and tedious, as common problems (e.g., deadlock, fairness) can be hard to trigger and the resulting trace can often contain hundreds of packets. Most OCN simulation, generation and characterization frameworks lack robust testing infrastructure to validate model correctness. Most of these frameworks only contain simple tests for a single router or a specific network instance. Many frameworks lack an automatic and systematic way to verify outputs.

C. Evaluating OCNs

Simulating OCNs – Simulation is provided by most OCN simulators using CL modeling [1, 3, 10, 21, 31, 42]. Some multicore simulators [7, 39] also integrate dedicated on-chip network simulators. Conventional OCN generators that generate synthesizable RTL code do not have the ability to simulate the generated model. They often require other Verilog or SystemC simulators to drive the simulation.

Generating OCNs – ARM's CoreLink Interconnect [15] and Arteris FlexNoC [17] are two commercial OCN generators that target mobile applications. NoCGEN [11] can generate VHDL code for both 2D and 3D mesh topologies based on a user-defined OCN specification in XML file format but with very limited configuration options. Connect [35] focuses on generating OCNs optimized for FPGA implementations. All the above OCN generators are closed-source, leading to limited visibility into the implementation details and the inability to extend the framework. Open-source OCN generators are emerging as part of the open-source hardware movement. OpenPiton [4] is an open-source many-core research framework that contains three 2D mesh OCNs to ensure deadlock-free operation and provide communication between the tiles for cache coherence, I/O and memory traffic, and inter-core interrupts. Netmaker [34] is written in SystemVerilog and passes parameters by including a single parameter file in all modules. It provides testbenches and simulation for the entire OCN. OpenSMART [29] is an open-source OCN generator implemented in BSV and Chisel. It can generate SMART NoCs [28] to enable single-cycle multi-hop traversals in arbitrary topologies. However, no FL and CL simulation is provided in these prior works. OpenSoC Fabric [16]

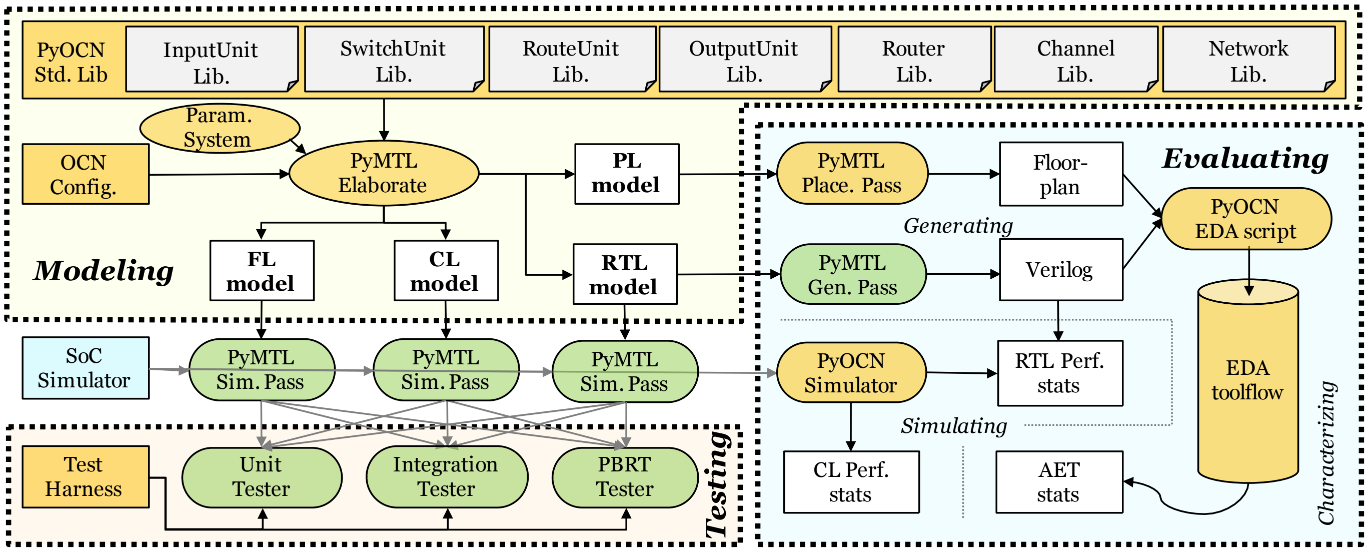

Figure 1. Overview of PyOCN Framework – PyOCN provides a library of OCN components to compose networks. PyOCN also provides unit test for each building block and integration test for the target OCNs. Property-based random test (PBRT) is used for stress testing network models. An OCN simulator is implemented for simulating different OCNs and backend scripts are provided to drive the EDA toolflows for area, energy, and timing characterization. PyOCN also features a parameterization system facilitating easy OCN configuration. Green components are included as part of the PyMTL framework. Orange components are added as part of the PyOCN framework.

provides an open-source OCN generator implemented in Chisel. It can generate both software (C++) and hardware (Verilog) models but without a native simulator. Users must work in multiple languages when writing testbenches in C++ or Verilog.

Parameterization is clearly a first-order concern in OCN simulators and generators. Most existing tools implement parameterization using a declarative OCN specification (e.g., configuration parameters in XML file format). Developers need to carefully modify the configuration file to make sure the new parameters can be properly passed down the hierarchy. This significantly increases the development effort especially for designing complex OCNs where different modules share the same parameter name (namespace collision) or the same modules produce differently parameterized outputs.

Characterizing OCNs – Power- and area-models (e.g., Orion [43], Orion2.0 [23], DSENT [40]) are widely used to characterize complete OCNs or OCN components (e.g., routers, channels) early in the design cycle. These frameworks can also be integrated into OCN generator frameworks for high-level optimization. For instance, COSI [37] is a synthesis framework for OCNs that embeds the power and area models derived from Orion to facilitate the OCN optimization. However, COSI targets synthesis without support for higher-level modeling. Similarly, SUNMAP [33] leverages XPipes [5] and Orion's power model to automatically generate SystemC descriptions of power-optimized network components. NetChip [6] integrates SUNMAP with the XPipesCompiler [20] to generate synthesizable Verilog for OCN designs. HotSniper [36] allows interval thermal simulation of many-cores. However, generation of synthesizable Verilog is not supported. High-level area, energy, and timing models enable early characterization, however, the lack of a detailed implementation leads to significant inaccuracy [24, 25] and iterative development (Orion series [23, 25, 43]).

III. PyOCN Framework

PyOCN is a unified framework for modeling, testing, and evaluating on-chip interconnection networks. Figure 1 shows an overview of the PyOCN framework and illustrates its tight integration with PyMTL [22, 32].

PyOCN extends the PyMTL framework with additional features suitable for *OCN design-space exploration*. Highly parameterized and modularized OCN components, modeled in FL, CL, RTL, and PL, serve as a standard library for building OCNs (see Section IV). In addition, PyOCN provides a comprehensive testing methodology based on unit testing, integration testing, and property-based random testing to test the FL, CL and RTL models (see Section V). To evaluate different OCN designs, PyOCN can generate synthesizable Verilog with the geometry information for floorplanning based on the RTL and PL models via the generation pass and placement pass (see Section VI). A parameterization system is implemented to allow developers to flexibly parameterize any module instance. For characterizing OCN components and networks, PyOCN provides a set of electronic-design automation (EDA) scripts to drive a commercial standard-cell-based toolflows.

PyMTL is a unified hardware modeling framework. It leverages Python for behavioral specification, structural elaboration, and verification, enabling a rapid code-test-debug cycle for hardware modeling. PyMTL allows a designer to write the design under test (DUT) and test bench completely in Python for simulation and only transit to the traditional HDL workflow to push the DUT through an FPGA/ASIC toolflow. The simulation engine written in Python drastically reduces the iterative development cycle and eliminates any semantic gap.

IV. PyOCN for Modeling OCNs

PyOCN provides a library of modular basic building blocks to compose OCNs. As shown in Figure 2, a router is composed

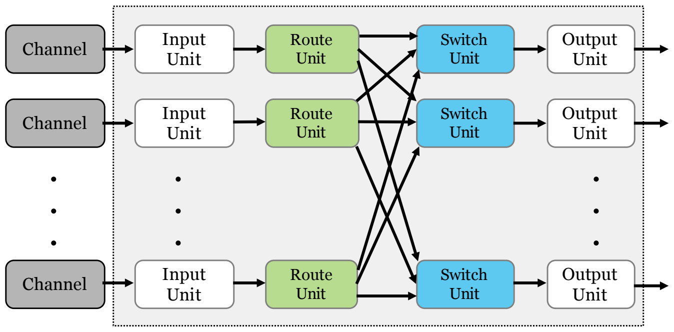

Figure 2. PyOCN Generic Router Architecture

```python
def ringnet_fl( src_pkts ):
  nterminals = len( src_pkts )
  dst_pkts = [ [] for _ in range( nterminals ) ]

  for packets in src_pkts:
    for pkt in packets:
      dst_pkts[ pkt.dst ].append( pkt )
  return dst_pkts
```

Figure 3. FL Implementation of Ring Network – Simply redistributes an array of packet lists based on the destination field of each packet.

of input units, route units, switch units, and output units. All these basic components have standardized latency insensitive interfaces so that each component can easily be replaced by user-customized components to create new networks. For example, if we want to implement a ring network with on/off flow control, instead of reimplementing the whole router, we only need to implement an input unit and an output unit that supports on/off flow control and swap them into the standard ring network which uses credit-based flow control. The modular design approach also makes it easy to unit test the router, since we can test each basic component in isolation before we integrate them into a router.

By leveraging PyMTL, PyOCN is capable of modeling and generating OCNs at different levels of abstraction, including FL, CL, and RTL, in a unified environment, which enables a user to rapidly take an OCN design from concept to implementation. This section describes PyOCN's modeling approach spanning FL, CL, and RTL modeling.

Functional-Level Modeling – An FL network is essentially a magic crossbar. Figure 3 illustrates the FL implementation of a mesh network. PyOCN provides FL network models to enable early-stage validation and fast emulation of the model. We can write tests, check them first against the FL network, and then reuse these tests to verify CL and RTL networks at later design stages. Developing test cases with validation against FL network improves the credibility of these test cases. In other words, if the CL or RTL networks fail a test, it is more likely due to an error in the CL or RTL implementation, rather than an incorrect test case. Furthermore, our FL networks can also be composed with lower-level (i.e., CL and RTL) cores, memories, and accelerators to help develop end-to-end software that runs correctly on an SoC model.

Cycle-Level Modeling – PyOCN provides CL networks to facilitate rapid design-space exploration of cycle-level performance across a wide range of microarchitectural parameters, such as topology, routing algorithm, channel latency, type/size of input queues, and type of arbiters. The CL networks are built with the CL version of basic components. Figure 4 illustrates the implementation of a CL switch unit. Instead of using an arbiter, it simply instantiates a list of integers to model a round-robin arbiter. Our CL model is almost cycle-accurate (see Table II for an example), since most of the CL building blocks are cycle-accurate.

```python
class SwitchUnitCL( Component ):
  def construct( s, pkt_t, num_inports ):

    # Local parameters
    s.num_inports = num_inports

    # Interface
    s.get = [ \
        CallerIfc(pkt_t) for _ in range(num_inports) ]
    s.give = \
        CalleeIfc(pkt_t, method=s.give_, rdy=s.give_rdy)

    # Components
    s.priority = list( range(num_inports) )

    for i in range( num_inports ):
      s.add_constraints( M( s.get[i] ) == M( s.give ) )

  def give_rdy( s ):
    for i in range( s.num_inports ):
      if s.get[i].rdy():
        return True
    return False

  def give_( s ):
    for i in s.priority:
      if s.get[i].rdy():
        s.priority.append( s.priority.pop(i) )
        return s.get[i]()
```

Figure 4. CL Implementation of SwitchUnit – It is parametrized by the packet type and the number of inputs. It uses a list of integers `s.priority` to model a round-robin arbiter.

Register-Transfer-Level Modeling – PyOCN also provides RTL implementations of multiple networks for cycle-accurate performance evaluation and ASIC/FPGA synthesis. Similar to CL networks, the RTL networks are composed using the RTL version of the basic building blocks. Figure 5 shows the implementation of an RTL switch unit in the PyMTL domain-specific language (DSL). PyMTL provides primitives similar to other hardware description languages: port-based interfaces for module encapsulation, structural connectivity for module composition, and combinational and synchronous concurrent blocks for logic description.

Physical-Level Modeling – Physical-level modeling (e.g., macro-/micro- floorplanning, cell tiling) is critical for effectively building complex OCNs. Without this kind of modeling, the structure in datapaths is destroyed by the automated place-and-route tools producing sub-optimal quality-of-results. We added a PyMTL placement pass to facilitate the physical-level modeling of the target network. The placement pass collects the geometry information of each network component and generates the floorplan script as shown in Figure 6.

```python
class SwitchUnitRTL( Component ):
  def construct( s, pkt_t, num_inports ):
    # Local Parameters
    sel_width = clog2( num_inports )
    sel_t     = mk_bits( sel_width   )
    grant_t   = mk_bits( num_inports )

    # Interface
    s.get = [GetIfc(pkt_t) for _ in range(num_inports)]
    s.send = SendIfc(pkt_t)

    # Components
    s.arbiter = RoundRobinArbiterEn( num_inports )
    s.mux = Mux( pkt_t, num_inports )(
      out = s.send.msg,
    )
    s.encoder = Encoder( num_inports, sel_width )(
      in_ = s.arbiter.grants,
      out = s.mux.sel,
    )

    # Connections
    for i in range( num_inports ):
      s.connect( s.get[i].rdy, s.arbiter.reqs[i] )
      s.connect( s.get[i].msg, s.mux.in_[i]      )

    @s.update
    def up_arb_send_en():
      s.arbiter.en = \
        ( s.arbiter.grants > grant_t(0) ) & s.send.rdy
      s.send.en = \
        ( s.arbiter.grants > grant_t(0) ) & s.send.rdy

    @s.update
    def up_get_en():
      for i in range( num_inports ):
        s.get[i].en = s.get[i].rdy & s.send.rdy & \
                      ( s.mux.sel == sel_t(i) )
```

Figure 5. RTL Implementation of SwitchUnit – The switch unit implementation reuses the RTL arbiter, encoder, and mux from PyMTL's standard library.

V. PyOCN for Testing OCNs

PyOCN provides extensive test suites to unit test the basic network components as well as complete network instances. The highly modular design of PyOCN enables rigorous unit testing for each basic building block.

In addition, our test suites can be easily reused across all modeling levels including FL, CL, and RTL because the generated networks at different levels all have standardized interfaces. Since PyMTL is embedded in Python, PyOCN is able to leverage powerful python packages to facilitate test-driven design of our OCN models. In our framework, we extensively use *pytest* [38] to generate and drive test cases and *hypothesis* [19] to perform property-based random testing. This section describes PyOCN's testing strategy spanning unit, integration, and property-based random testing.

Unit Testing – PyOCN provides unit tests not only for all network components such as routers and channels, but for basic components like input units and switch units. Figure 7 shows a simple example of one *unit test* for a router in a 4×4 mesh network. It simply injects two packets into the router and checks if they are ejected from the correct output ports. The pytest `@parametrize` decorator generates a number of test configurations from a single test definition. In this case, it generates 16 test cases that test routers with all possible positions in a 4×4 network. This test can be used for testing both CL and RTL routers. It can be reused for testing torus routers as well. The only change we need to make is to change the type of the design-under-test (DUT) in the test harness.

```python
class RingNetworkRTL(Component):
  def construct(s, pkt_t, pos_t, nrouters, chnl_lat=0):
  ...
  def elaborate_physical(s):
    N = s.nrouters
    chnl_len = s.channels[0].dim.w
    for i, r in enumerate(s.routers):
      if i < (N / 2):
        r.dim.x = i * (r.dim.w + chnl_len)
        r.dim.y = 0
      else:
        r.dim.x = (N - i - 1) * (r.dim.w + chnl_len)
        r.dim.y = r.dim.h + chnl_len
    s.dim.w = N/2 * r.dim.w + (N/2 - 1) * chnl_len
    s.dim.h = 2 * r.dim.h + chnl_len
```

Figure 6. Physical Elaboration – Floorplanning code for parameterizable ring network. Geometry information is propagated hierarchically from each router and channel instance in the network component.

```python
@pytest.mark.parametrize(
  'pos_x, pos_y',
  product( [ 0, 1, 2, 3 ], [ 0, 1, 2, 3 ] )
)
def test_simple_4x4( pos_x, pos_y ):
  ncols = 4; nrows = 4
  pkt_t = mk_mesh_pkt( ncols, nrows, nvcs=2 )

  src_pkts = [
    #     src_x  y  dst_x  y  opaque  vc  payload
    pkt_t(    0, 0,     1, 1,      0,  0, 0xfaceb00c ),
    pkt_t(    0, 2,     3, 3,      0,  0, 0xdeadface ),
  ]

  th = TestHarness( pkt_t, src_pkts )
  # Use the elegant parameter system
  th.set_param( "top.construct",
    ncols=ncols, nrows=nrows,
    pos_x=pos_x, pos_y=pos_y,
  )
  run_sim( th )
```

Figure 7. Unit Test for a Router in 4×4 Mesh - This simple test case injects two packets into the router. The test harness instantiates the router, injects the packets, and checks if the packets are ejected from the correct output ports.

Integration Testing – PyOCN provides similar tests that integrate basic components into a router, compose routers and channels into a network, and then test the network as a whole. Many test cases are reusable across different topologies as they share the same FL model and have the same interfaces. RTL networks can reuse test cases for CL networks as PyMTL supports multi-level composition of CL and RTL interfaces.

Property-based Random Testing – Property-based random testing was first popularized by the Haskell library QuickCheck [14]. It works by using a type-based random data generator for all inputs and checking if the DUT violates the given specification. Figure 8 illustrates a simple example of

```
@hypothesis.given(
  ncols = st.integers(2, 8),
  nrows = st.integers(2, 8),
  pkts  = st.data(),
)
def test_hypothesis( ncols, nrows, pkts ):
  Pkt = mk_mesh_pkt( ncols, nrows, nvcs=2 )

  pkts_lst = pkts.draw(
    st.lists( mesh_pkt_strat( ncols, nrows ) ),
    label= "pkts"
  )

  src_pkts = mk_src_pkts( ncols, nrows, pkts_lst )
  dst_pkts = meshnet_fl( ncols, nrows, src_pkts )
  th = TestHarness( Pkt, ncols, nrows,
                    src_pkts, dst_pkts )
  run_sim( th )
```

Figure 8. Property-Based Random Testing for Mesh Network Generator – This test shows a simple example of how PyOCN leverages hypothesis to test network generators. The test function randomly configures a mesh network and draws a list of packets as input. It verifies the DUT's output against the FL model which serves as an oracle.

how PyOCN leverages *hypothesis*, an open-source property-based random testing framework for Python. This tests more than a single network instance. Rather, it randomly configures mesh networks with different sizes on the fly and verifies the generated networks against the golden reference, which in this case is the FL model. *Hypothesis* produces readable and minimal counter-examples when encountering a failure. If it finds an example failing the specification, it takes that example and keeps simplifying it until it finds a minimal example that still triggers the problem.

VI. PyOCN for Evaluating OCNs

In addition to modeling and testing OCNs, PyOCN also supports evaluating OCNs using a combination of simulation (for cycle-level or cycle-accurate performance analysis), generation (for producing synthesizable RTL), and characterization (for area, energy, and timing analysis).

Simulating OCNs – The simulation in PyOCN is powered by PyMTL's simulation pass, which allows the test bench to be written in Python and eliminates any semantic gap. The simulation pass statically schedules and then calls the *@s.update* blocks every cycle. The PyOCN simulator can issue packets into different OCNs with different traffic patterns (e.g., uniform random (urandom), neighbor, partition by two (partition2), and complement) at different injection rates.

Generating OCNs – PyOCN leverages the translation pass in PyMTL to generate synthesizable Verilog from RTL OCN models. PyOCN's *parameterization system* facilitates the configuration process of OCN components. In PyOCN, model implementations are defined as Python classes. The constructor registers each module in a dictionary based on its name and hierarchical position. The parameterization system can modify any parameter in any module registered in the dictionary by tagging a specific hierarchical component name with parameters. So instead of tediously carrying all the parameters down through the whole hierarchy during construction, developers are able to parameterize only a set of components or any single component *in the middle of hierarchy* after construction but before elaboration. During elaboration time, models are elaborated based on the updated parameters in each module.

Table II. Multi-Level Simulation

Injection Rate	0.01	0.1	0.2	0.3	0.4
Performance	17.9	15.5	14.2	13.3	13.0
Accuracy	86%	87%	87%	97%	74%

Normalized simulation performance (simulated cycles per second) and accuracy of average latency measurement modeled in CL with respect to RTL. The accuracy of the CL model is slightly degraded under very high load. The ideal throughput for the mesh network is 0.5.

Characterizing OCNs – PyOCN generates both synthesizable Verilog and a corresponding floorplan script that can be used to drive a commercial standard-cell-based toolflow for area, energy, and timing characterization. The PyOCN framework includes scripts for various commercial tools including Synopsys Design Compiler, Cadence Innovus, and Synopsys PrimeTime PX in order to synthesize, place, route, and estimate energy for the given design. PyOCN leverages open-source physical IP libraries including the 45 nm NanGate standard-cell library and the FreePDK45 physical design kit.

PyOCN's integration with a standard-cell-based toolflow enables highly accurate measurement of area, energy, and timing for the placed-and-routed gate-level netlist. Specifically, area and timing are both estimated post-place-and-route after meeting timing with Cadence Innovus's internal signoff-quality static timing analysis engines. Energy is estimated using Synopsys PrimeTime PX with the RTL-level switching activity information (provided by PyOCN) and the post-place-and-route gate-level netlist. The tool statistically propagates annotated switching activity to intermediate nodes before using gate/interconnect and parasitic RC information to estimate energy.

VII. Case Study

With the help of PyOCN's standard library, an OCN can be easily configured and modeled at various abstraction levels. Currently, PyOCN supports crossbar, ring, mesh, concentrated mesh, torus, and butterfly topology models with extensive testing infrastructure. In this case study, we explore an OCN targeting a 64-terminal system.

PyOCN provides an OCN *simulator* for different topologies modeled at various levels. A developer can initially simulate the target design in CL to quickly estimate performance. Table II shows that the simulation speed for a 64-terminal mesh in CL is over 10× faster than an equivalent RTL model. The simulated performance of different topologies modeled in RTL is shown in Figure 9. In this case study, we choose to optimize for a unified random (i.e., *Urandom*) traffic pattern which might be representative of general memory traffic over an OCN interconnecting private L1 caches and a tiled, shared L2 cache. Given this context, we chose a butterfly OCN for further analysis.

PyOCN supports OCN *characterization* by providing scripts that semi-automatically takes the *generated* Verilog and net activity file to drive a standard-cell-based electronic-design-automation (EDA) toolflow for area, energy, and timing analysis. In this case study, we choose to use the FreePDK45 with the

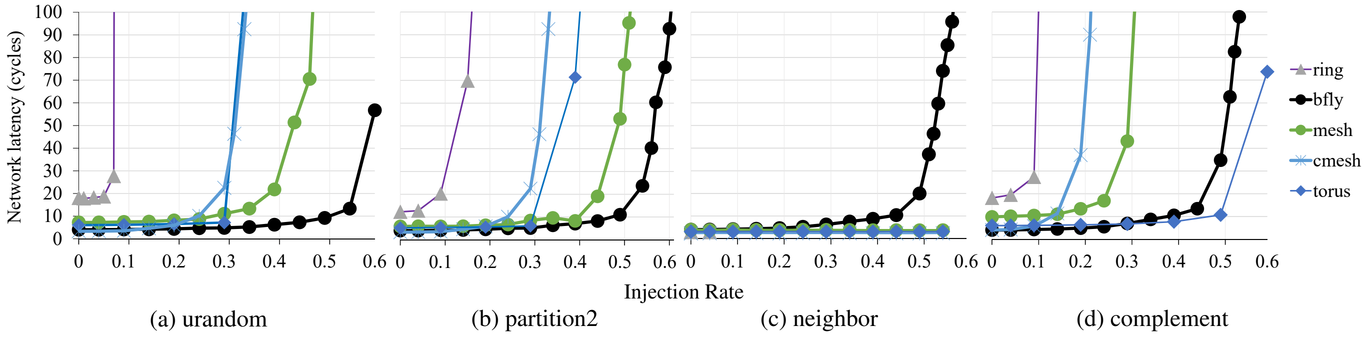

Figure 9. RTL Simulation Results – Average latency at different injection rates across different network topologies with 64 terminals. Mesh and torus both have eight rows and eight columns. Butterfly is 4-ary 3-fly. All topologies parameterize the channel latency as a single cycle and the router pipeline as a single cycle. Ring and torus leverage virtual channels to avoid deadlock.

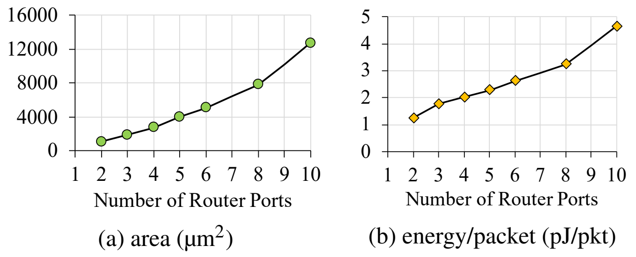

Figure 10. Router Characterization – Characterization of area and energy for routers with different number of input and output ports targeting a 500 MHz clock frequency. The channel bandwidth is 32 b/cycle. During the RTL simulation, we generate hundreds of packets that traverse from each inport to each outport without contention. The dumped net activity file is passed into the EDA toolflow to drive the energy analysis.

Nangate standard cell library. Figure 10 shows the area-energy analysis for a router with an increasing number of ports. Generally, the higher-radix routers require more area and energy per packet. We eventually decided to implement a 4-ary 3-fly rather than a 2-ary 6-fly as the zero load latency of 4-ary 3-fly is half the 2-ary 6-fly.

To place a 4-ary 3-fly butterfly, we group routers in the same row together as a router group and place them on the chip as shown in Figure 11, which is similar to the placement of the flattened butterfly topology proposed in [27]. PyOCN's *PL modeling* can provide explicit geometry information for the placement of each router. We reserve 1 mm between router groups to provide enough space for the terminals. We initially target 500 MHz and set the channel latency to be one meaning that there is no channel queues between routers.

However, using the EDA toolflow revealed that a 2 ns timing constraint is not possible due to long channels between some router groups. The corresponding critical path starts from the input unit of Router28, goes through the channel, and ends at the input unit of Router46, with a negative slack of 0.13 ns. Although the channel between Router28 and Router47 seems longer according to the logical layout, the EDA toolflow's routing algorithm ultimately meant the critical path was limited by the channel from Router28 to Router46. One straightforward way to break such critical paths is to add channel queues to channels that violate the timing constraint.

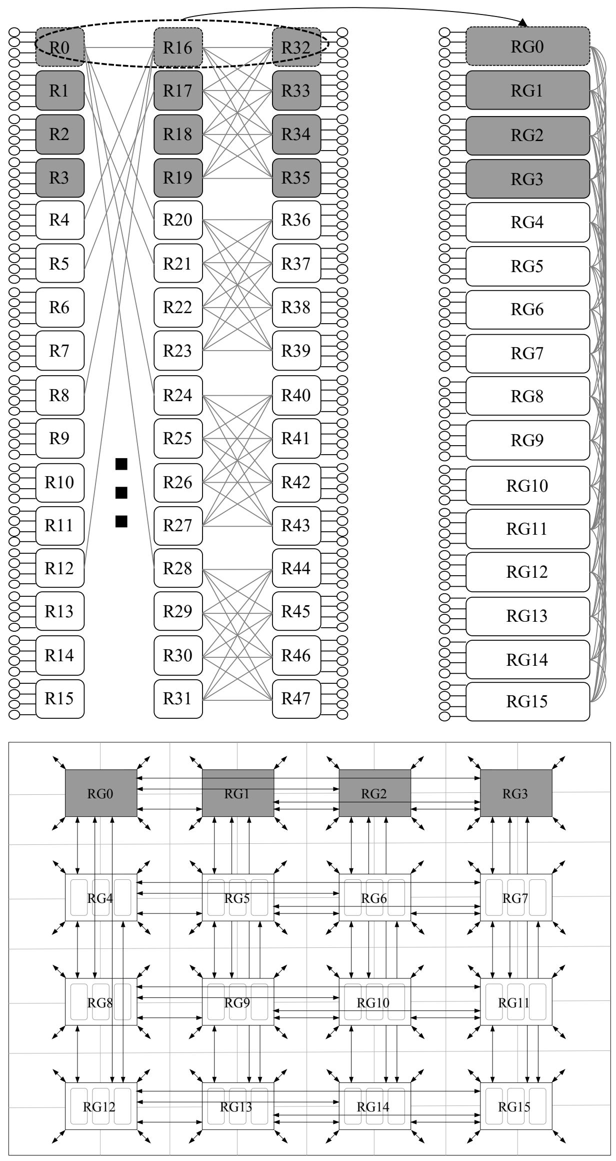

Figure 11. 4-ary 3-fly Butterfly Network – The routers in the same rows can be recognized as a router group, which can be placed onto the chip based on the placement proposed in [27]. For simplicity, we use single line with two arrows to indicate bidirectional data delivery.

```
net = BFlyNetworkRTL( pkt_t, k_ary=4, n_fly=3 )
critical_paths= [
  "channels[82]",
  "channels[114]",
  ...
]
for c in critical_paths:
  net.set_param( f'top.{c}.construct', hops=2 )
net.elaborate()
```

Figure 12. Parameterization System Example – We collect all the critical paths violating the timing constraint reported by the EDA toolflow, add them into the `critical_paths`, and use `set_param` to change the number of channel queues on these channels.

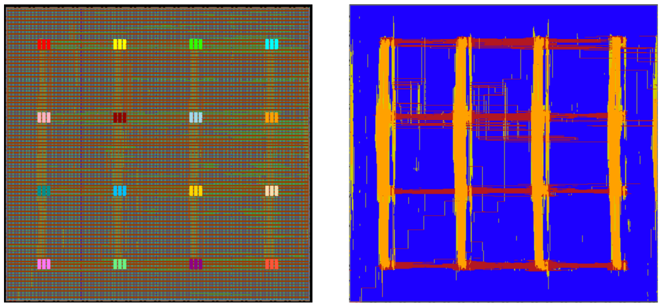

Figure 13. Post Place-and-Route Layout of 4-ary 3-fly Butterfly – The routers are highlighted. The floorplan is generated based on PyOCN PL modeling. PyOCN also provides script to semi-automatically drive the EDA toolflow to generate the final layout. The area is 4.8 mm x 4.6 mm and the operating frequency is 500 MHz @ 45 nm.

The *parameterization system* of PyOCN enables us to easily configure any network components without touching the source code of the target OCN design. As shown in Figure 12, only a few lines before the elaboration of the target OCN is needed to add channel queues to specific channels. Figure 13 shows the final layout of our target butterfly OCN, where the target frequency is achieved.

VIII. Conclusion

This paper has introduced PyOCN, a unified framework for simulating, testing, and characterizing on-chip interconnection networks. PyOCN is the first open-source framework for modeling (e.g., functional-level, cycle-level, and register-transfer-level), testing (e.g., unit testing, integration testing, and property-based random testing), and evaluating (e.g., simulating, generating, and characterizing) on-chip interconnection networks. PyOCN is an open-source framework and is available online at `https://github.com/cornell-brg/pymtl3-net`.

Acknowledgments

This work was supported in part by NSF CRI Award #1512937, DARPA POSH Award #FA8650-18-2-7852, and equipment, tool, and/or physical IP donations from Intel, Synopsys, and Cadence. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation theron. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

References

[1] P. Abad, P. Prieto, L. G. Menezo, A. Colaso, V. Puente, and J.-Á. Gregorio. Topaz: An Open-Source Interconnection Network Simulator for Chip Multiprocessors and Supercomputers. *Int'l Symp. on Networks-on-Chip (NOCS)*, May 2012.

[2] S. N. Adya and I. L. Markov. Fixed-outline floorplanning: enabling hierarchical design. *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)*, Dec 2003.

[3] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2009.

[4] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff. OpenPiton: An Open Source Manycore Research Framework. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr 2016.

[5] D. Bertozzi and L. Benini. Xpipes: A Network-on-Chip Architecture for Gigascale Systems-on-Chip. *IEEE Circuits and Systems Magazine*, Sep 2004.

[6] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. D. Micheli. NoC Synthesis Flow for Customized Domain Specific Multiprocessor Systems-on-Chip. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 16(2):113–129, 2005.

[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 Simulator. *SIGARCH Computer Architecture News (CAN)*, 39(2):1–7, Aug 2011.

[8] J. Bolaria. Xeon Phi Targets Supercomputers. *Microprocessor Report, The Linley Group*, Sep 2012.

[9] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti. Noxim: An Open, Extensible and Cycle-Accurate Network on Chip Simulator. *Int'l Conf. on Application-Specific Systems, Architectures, and Processors (ASAP)*, Jul 2005.

[10] J. Chan, G. Hendry, A. Biberman, K. Bergman, and L. P. Carloni. Phoenixsim: A Simulator for Physical-Layer Analysis of Chip-Scale Photonic Interconnection Networks. *Design, Automation, and Test in Europe (DATE)*, Mar 2010.

[11] J. Chan and S. Parameswaran. NoCGEN: A Template based Reuse Methodology for Networks on Chip Architecture. *Int'l Conf. on VLSI Design*, Jan 2004.

[12] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2014.

[13] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2016.

[14] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *ACM SIGPLAN Notices*, 46(4):53–64, 2011.

[15] CoreLink Interconnect. accessed Sep 20, 2019. https://developer.arm.com/ip-products/system-ip/corelink-interconnect.

[16] F. Fatollahi-Fard, D. Donofrio, G. Michelogiannakis, and J. Shalf. Opensoc Fabric: On-Chip Network Generator: Using Chisel to Generate a Aarameterizable On-Chip Interconnect Fabric. *Int'l Workshop on Network on Chip Architectures*, Dec 2014.

[17] Arteris FlexNoC Interconnect IP. Online Webpage, accessed Sep 20, 2019.

[18] P. Greenhalgh. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. *EE Times*, Oct 2011. `http://www.eetimes.com/document.asp?doc_id=1279167`.

[19] Most testing is ineffective - Hypothesis. accessed Sep 20, 2019. https://hypothesis.works.

[20] A. Jalabert, S. Murali, L. Benini, and G. D. Micheli. ×pipesCompiler: A Tool for Instantiating Application Specific Networks on Chip. *Design, Automation, and Test in Europe (DATE)*, Feb 2004.

[21] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally. A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2013.

[22] S. Jiang, B. Ilbeyi, and C. Batten. Mamba: Closing the Performance Gap in Productive Hardware Development Frameworks. *Design Automation Conf. (DAC)*, Jun 2018.

[23] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi. ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration. *Design, Automation, and Test in Europe (DATE)*, Apr 2009.

[24] A. B. Kahng, B. Lin, and S. Nath. Explicit Modeling of Control and Data for Improved NoC Router Estimation. *Design Automation Conf. (DAC)*, Jun 2012.

[25] A. B. Kahng, B. Lin, and S. Nath. ORION3.0: A Comprehensive NoC Router Estimation Tool. *IEEE Embedded Systems Letters (ESL)*, Feb 2015.

[26] M. Karunaratne, C. Tan, A. Kulkarni, T. Mitra, and L.-S. Peh. Dnestmap: Mapping Deeply-Nested Loops on Ultra-Low Power CGRAs. *Design Automation Conf. (DAC)*, Jun 2018.

[27] J. Kim, J. Balfour, and W. Dally. Flattened Butterfly Topology for On-Chip Networks. *Int'l Symp. on Microarchitecture (MICRO)*, Aug 2007.

[28] T. Krishna, C.-H. O. Chen, W.-C. Kwon, and L.-S. Peh. SMART: Single-Cycle Multihop Traversals over A Shared Network on Chip. *IEEE Micro*, 34(3):43–56, 2014.

[29] H. Kwon and T. Krishna. Opensmart: Single-Cycle Multi-Hop NoC Generator in BSV and Chisel. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2017.

[30] H. Kwon, A. Samajdar, and T. Krishna. Maeri: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2018.

[31] M. Lis, K. S. Shim, M. H. Cho, P. Ren, O. Khan, and S. Devadas. DARSIM: A Parallel Cycle-Level NoC Simulator. *Workshop on Modeling, Benchmarking and Simulation (MOBS)*, Jun 2010.

[32] D. Lockhart, G. Zibrat, and C. Batten. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[33] S. Murali and G. D. Micheli. SUNMAP: A Tool for Automatic Topology Selection and Generation for NoCs. *Design Automation Conf. (DAC)*, Jul 2004.

[34] NetmakerWiki. accessed Sep 20, 2019. http://www-dyn.cl.cam.ac.uk/ rdm34/wiki/index.php.

[35] M. K. Papamichael and J. C. Hoe. CONNECT: Re-examining Conventional Wisdom for Designing NoCs in the Context of FPGAs. *Int'l Symp. on Field Programmable Gate Arrays (FPGA)*, Feb 2013.

[36] A. Pathania and J. Henkel. HotSniper: Sniper-Based Toolchain for Many-Core Thermal Simulations in Open Systems. *IEEE Embedded Systems Letters (ESL)*, Aug 2018.

[37] A. Pinto, L. P. Carloni, and A. Sangiovanni-Vincentelli. COSI: A Framework for the Design of Interconnection Networks. *Design, Automation, and Test in Europe (DATE)*, Oct 2008.

[38] PyTest. Online Webpage, 2014 (accessed Oct 1, 2014).

[39] P. Ren, M. Lis, M. H. Cho, K. S. Shim, C. W. Fletcher, O. Khan, N. Zheng, and S. Devadas. Hornet: A Cycle-Level Multicore Simulator. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 31(6):890–903, 2012.

[40] C. Sun, C.-H. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic. DSENT-A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling. *Int'l Symp. on Networks-on-Chip (NOCS)*, May 2012.

[41] C. Tan, M. Karunaratne, T. Mitra, and L.-S. Peh. Stitch: Fusible Heterogeneous Accelerators Enmeshed with Many-Core Architecture for Wearables. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2018.

[42] A. Tran and B. Baas. NoCTweak: A Highly Parameterizable Simulator for Early Exploration of Performance and Energy of Networks On-Chip. Technical Report ECE-VCL-2012-2, VLSI Computation Lab, ECE Department, University of California, Davis.

[43] H. Wang, L.-S. Peh, and S. Malik. Orion: A Power-Performance Simulator for Interconnection Networks. *Int'l Symp. on Microarchitecture (MICRO)*, Nov 2002.

[44] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi GF100 GPU Architecture. *IEEE Micro*, 31(2):50–59, Mar/Apr 2011.