2021 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE OCTOBER 26-27, 2021

SimPy for Chips

A Discrete Event Simulation framework in python for large scale architectual modelling of machine intelligence accelerators

Hachem Yassine, Daniel Wilkinson, Graham Cunnigham, Iason Myttas,

Graphcore Ltd, Bristol, United Kingdom

***Abstract*— We discuss the development of a framework for modelling SoC Interconnect, chip-to-chip interconnect and other on-chip features of Graphcore's reticle-limited 7nm Intelligence Processing Unit (IPU) Machine Intelligence Accelerator using python and the excellent *SimPy* package for Discrete Event Simulation (DES). We review the advantages of this purely python approach versus more traditional approaches based on SystemC or similar, discuss extensions we have made to the default SimPy package that specializes it for modelling of ASIC designs, and show how we have applied this framework for performance modeling and for resolving issues such as head-of-line blocking and deadlocks both prior to and after tape-out, and how we extend our approach to clusters of many such devices.**

I. INTRODUCTION

Architectural modelling and prototyping of ASIC hardware in high level languages is a common technique for chip design, for architectural exploration and validation and for verification of ASICs. Here we focus on the use of Python, and in particular the popular SimPy package for Discrete Event Simulation (DES), for use in architectural exploration and validation, both of chips, and systems built from 10's or 100's of chips.

SystemC - or just plain C or C++ is also widely used for such purposes and various tools, libraries and frameworks for such tasks are available [1]. At Graphcore however, Python is widely used through the chip development process, and in general our engineers prefer to work in python where possible thanks to its easier learning curve, productivity and huge range of useful packages. As one might expect, Python has been employed for various chip design purposes such as modelling high speed SERDES [2] and for refinement from high level descriptions to RTL [3] to give just two examples. Our work by contrast is in the area of architectural exploration.

In this brief review of existing work we found that while SimPy is a well-used and popular package for Discrete Event Simulation in various disciplines, there is no real evidence of its use for silicon chip architecture or design as yet.

A. About SimPy

The SimPy [4] package offers primitive components for events, storage components such as containers and queues (suitable for modelling FIFOs and other queuing elements) and shared resources with support for prioritization and pre-emption (suitable for use in representing arbiters). Its design is based upon python generator functions and the associated python yield statement. When a yield statement is hit, program execution is suspended and the yielded value is returned to the calling function. This built in functionality of python enables SimPy to implement a Discrete Event Simulation library using python's built-in capabilities. By then building a simple simulation kernel and component library on top of this, SimPy offers an easy-to-use and easy-to-extend API for quickly building DES models of complex systems.

SimPy was first released in 2002, and at the time of writing is at version 4.0.1 with wide usage in academia and industry and a good body of documentation.

B. About the Graphcore Intelligence Processing Unit

The IPU [5] consists of two parts: the Core and the SoC. The Core is an homogenous array of thousands of 32-bit Tile Processors each with their own locally attached SRAM memory and arithmetic units to accelerate ML compute tasks, plus a deterministic, stateless all-to-all interconnect that allows each Tile Processor to communicate with all its peers on the same device.

The SoC on the 2nd generation IPU (Mk2) consists of a ring surrounding the Core, comprising a large quantity of 16 Gbps SerDes lanes for IPU to IPU connectivity, a PCI Endpoint Controller for attach to a local host system, sundry functions related to clock and reset generation, hardware synchronization between IPU devices, and proprietary interconnect solutions for control register access and data path traffic flow between Tile Processors on different chips. When measured by die area the Core occupies most of the device, with the SoC relegated to the periphery of the die on all four edges. When measured by complexity the SoC architecture is at least as complex, if not more so, than the Core due to its heterogenous nature, complex standards-based third party IP and a queued rather than deterministic interconnect. In addition, there is a requirement to build full system models featuring many IPU devices and the high speed fabric (external switches) interconnecting them.

The backbone of the SoC in the Mk2 IPU is a very high bandwidth packet-switched ring interconnect, sufficient to facilitate the array of Tile Processors to fully utilize the 1.6 Tbps of off-chip bandwidth. This on-die and chip-to-chip interconnect plus its associated SerDes links are the main focus of our work to date.

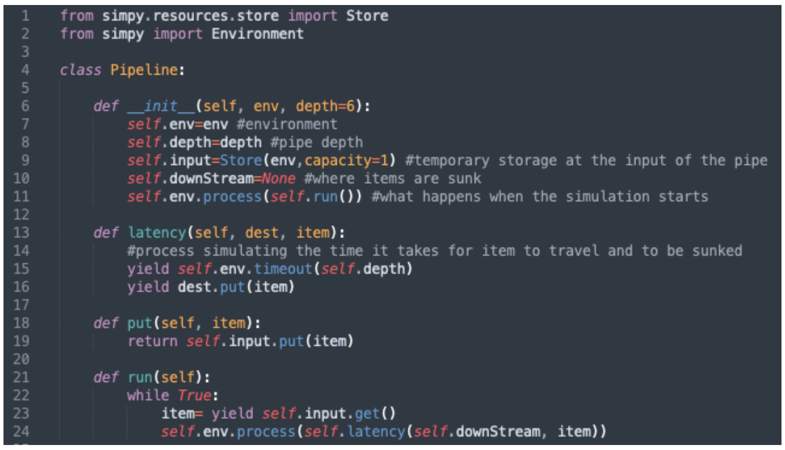
II. SIMPY BASICS

The behavior of all SimPy components is modeled using **processes**, which exist within an **environment**. Processes are python generators, which both create events and yield events (e.g. to wait for an event). An example of a simple event is a timeout, which is triggered after some simulation time has passed, allowing any process to sleep for a fixed simulation time.

A. Pipeline Example

Figure 1 shows our implementation of a simple pipeline object. Objects inserted into the pipeline emerge from the other end a set number of simulation time ticks later.

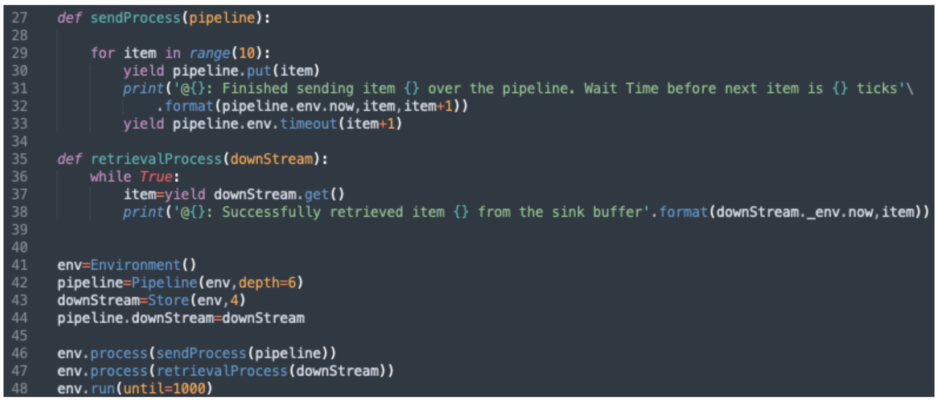
The put() method is used to send an item (any python object) through the pipeline into a downstream sink from where it can be retrieved only after the time to travel through the pipeline has passed. One process may be sending items through the pipeline while another process maybe retrieving them from the sink. Multiple objects may be in flight along the pipeline at any time and each such object will spawn an additional latency() process.



```python
from simpy.resources.store import Store
from simpy import Environment

class Pipeline:

    def __init__(self, env, depth=6):
        self.env=env #environment
        self.depth=depth #pipe depth
        self.input=Store(env,capacity=1) #temporary storage at the input of the pipe
        self.downStream=None #where items are sunk
        self.env.process(self.run()) #what happens when the simulation starts

    def latency(self, dest, item):
        #process simulating the time it takes for item to travel and to be sunked
        yield self.env.timeout(self.depth)
        yield dest.put(item)

    def put(self, item):
        return self.input.put(item)

    def run(self):
        while True:
            item= yield self.input.get()
            self.env.process(self.latency(self.downStream, item))
```

Figure 1

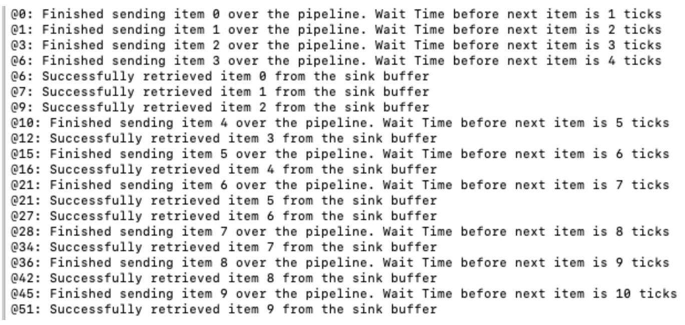
The code in Figure 2 defines two such processes (as generator functions) and adds them to the environment before running it (the items in this case are a sequence of integers). Note that the downstream is a simple SimPy **Store**, which acts as a buffer for the incoming packets and it is connected to the pipeline by direct assignment to the downStream variable.



```python
def sendProcess(pipeline):

    for item in range(10):
        yield pipeline.put(item)
        print('@{}: Finished sending item {} over the pipeline. Wait Time before next item is {} ticks'\
            .format(pipeline.env.now,item,item+1))
        yield pipeline.env.timeout(item+1)

def retrievalProcess(downStream):
    while True:
        item=yield downStream.get()
        print('@{}: Successfully retrieved item {} from the sink buffer'.format(downStream._env.now,item))


env=Environment()
pipeline=Pipeline(env,depth=6)
downStream=Store(env,4)
pipeline.downStream=downStream

env.process(sendProcess(pipeline))
env.process(retrievalProcess(downStream))
env.run(until=1000)
```

Figure 2

Running this code will result in the output in Figure 3, showing the event that happens at different timestamps (@). In this example, the pipe depth is 6 ticks, and it can be seen that the object is retrievable 6 ticks after being sent. Also note the wait time before starting to send another item over the pipeline after one item is sent, used to represent an object of non-zero size being placed into the pipeline over multiple time ticks.

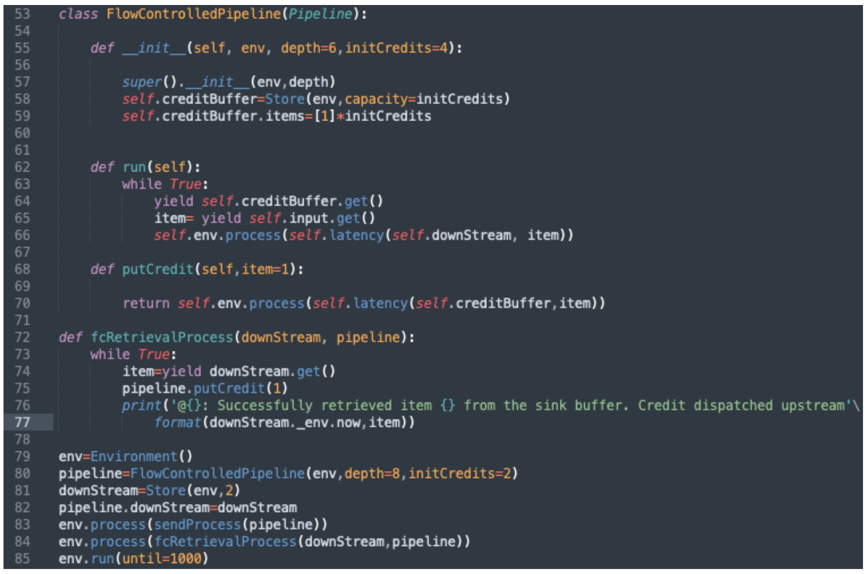


```
@0: Finished sending item 0 over the pipeline. Wait Time before next item is 1 ticks
@1: Finished sending item 1 over the pipeline. Wait Time before next item is 2 ticks
@3: Finished sending item 2 over the pipeline. Wait Time before next item is 3 ticks
@6: Finished sending item 3 over the pipeline. Wait Time before next item is 4 ticks
@6: Successfully retrieved item 0 from the sink buffer
@7: Successfully retrieved item 1 from the sink buffer
@9: Successfully retrieved item 2 from the sink buffer
@10: Finished sending item 4 over the pipeline. Wait Time before next item is 5 ticks
@12: Successfully retrieved item 3 from the sink buffer
@15: Finished sending item 5 over the pipeline. Wait Time before next item is 6 ticks
@16: Successfully retrieved item 4 from the sink buffer
@21: Finished sending item 6 over the pipeline. Wait Time before next item is 7 ticks
@21: Successfully retrieved item 5 from the sink buffer
@27: Successfully retrieved item 6 from the sink buffer
@28: Finished sending item 7 over the pipeline. Wait Time before next item is 8 ticks
@34: Successfully retrieved item 7 from the sink buffer
@36: Finished sending item 8 over the pipeline. Wait Time before next item is 9 ticks
@42: Successfully retrieved item 8 from the sink buffer
@45: Finished sending item 9 over the pipeline. Wait Time before next item is 10 ticks
@51: Successfully retrieved item 9 from the sink buffer
```

Figure 3

B. Flow Controlled Pipeline and Buffer

A richer example given in Figure 4 , essential in any queuing system, is a credit-based flow controlled version of the above. Here the pipeline will not allow anything into it before knowing that the downstream can accept the item being sent. This is involves the movement of data between buffers with a credit signal flowing in the opposite direction indicating the spare capacity in the receiving buffer; a widely used pattern of NoC interconnect design.



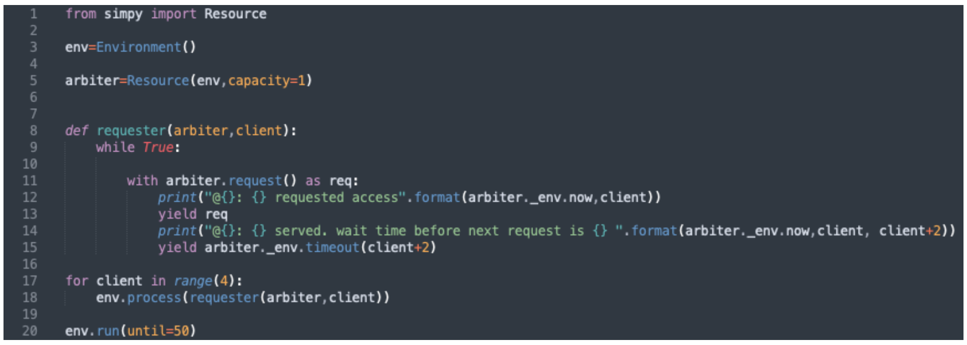
```python
class FlowControlledPipeline(Pipeline):

    def __init__(self, env, depth=6,initCredits=4):

        super().__init__(env,depth)
        self.creditBuffer=Store(env,capacity=initCredits)
        self.creditBuffer.items=[1]*initCredits


    def run(self):
        while True:
            yield self.creditBuffer.get()
            item= yield self.input.get()
            self.env.process(self.latency(self.downStream, item))

    def putCredit(self,item=1):

        return self.env.process(self.latency(self.creditBuffer,item))

def fcRetrievalProcess(downStream, pipeline):
    while True:
        item=yield downStream.get()
        pipeline.putCredit(1)
        print('@{}: Successfully retrieved item {} from the sink buffer. Credit dispatched upstream'\
            format(downStream._env.now,item))

env=Environment()
pipeline=FlowControlledPipeline(env,depth=8,initCredits=2)
downStream=Store(env,2)
pipeline.downStream=downStream
env.process(sendProcess(pipeline))
env.process(fcRetrievalProcess(downStream,pipeline))
env.run(until=1000)
```

Figure 4

Note that the pipeline will block until it extracts a credit from its credit buffer before starting to send as it did in the basic pipeline. The putCredit() is the method that the downstream buffer uses to send a credit back whenever a packet is removed from it as shown by the updated fcRetrievalProcess().

C. Simple Arbiter

An arbiter is another vital element of any SoC design, for both interconnects and functional blocks. SimPy provides a **Resource** object that can be requested by multiple processes but only one requester is fulfilled at a time. Figure 5 shows a simple arbiter with 4 clients requesting the resource at constant intervals of time.



```python
from simpy import Resource

env=Environment()

arbiter=Resource(env,capacity=1)


def requester(arbiter,client):
    while True:

        with arbiter.request() as req:
            print("@{}: {} requested access".format(arbiter._env.now,client))
            yield req
            print("@{}: {} served. wait time before next request is {} ".format(arbiter._env.now,client, client+2))
            yield arbiter._env.timeout(client+2)

for client in range(4):
    env.process(requester(arbiter,client))

env.run(until=50)
```

Figure 5

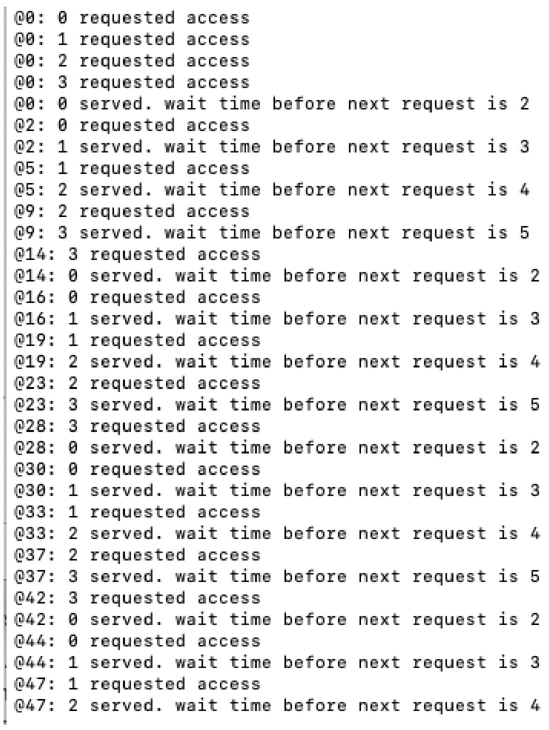
The output in Figure 6 shows the time at which each client makes request and the time at which its request is served. Note that the requests are served by the order of arrival. SimPy also provides a **PriorityResource** where each request can be assigned a priority.

III. CUSTOMISATION OF SIMPY TO ADDRESS LIMITATIONS RELATED TO SOC MODELLNG

While the preceding examples demonstrate the good fit of SimPy built in primitives for SOC modelling, we encountered a number of key limitations in terms of ease of use and performance. We have addressed these by making modifications to the SimPy package which we detail below.

A. *Delayed success of events*

In the default SimPy Event objects, Timeout is the only event that can be scheduled to succeed at a specific future time. Other events can only succeed instantaneously. In chip design, it is common to have operations that take a specified duration and hence their associated event is required to succeed at a certain time in the future. As a result, in our first Mk1 IPU implementation, we resorted to generous use explicit timeouts to poll for some event to complete. In other words, to simulate the above, we defined a two-step process that waits for a Timeout event to be successful before triggering the event of interest. This hurts performance and is not in keeping with the event-driven philosophy SimPy tries to support, and which is necessary for performant models.



```
@0: 0 requested access
@0: 1 requested access
@0: 2 requested access
@0: 3 requested access
@0: 0 served. wait time before next request is 2
@2: 0 requested access
@2: 1 served. wait time before next request is 3
@5: 1 requested access
@5: 2 served. wait time before next request is 4
@9: 2 requested access
@9: 3 served. wait time before next request is 5
@14: 3 requested access
@14: 0 served. wait time before next request is 2
@16: 0 requested access
@16: 1 served. wait time before next request is 3
@19: 1 requested access
@19: 2 served. wait time before next request is 4
@23: 2 requested access
@23: 3 served. wait time before next request is 5
@28: 3 requested access
@28: 0 served. wait time before next request is 2
@30: 0 requested access
@30: 1 served. wait time before next request is 3
@33: 1 requested access
@33: 2 served. wait time before next request is 4
@37: 2 requested access
@37: 3 served. wait time before next request is 5
@42: 3 requested access
@42: 0 served. wait time before next request is 2
@44: 0 requested access
@44: 1 served. wait time before next request is 3
@47: 1 requested access
@47: 2 served. wait time before next request is 4
```

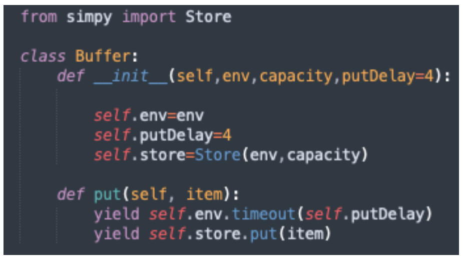
Figure 6

To allow us to build a performant API that is suitable to validate large scale systems, we extended the native Event class (**NewEvent**) by changing its succeed() method to allow delayed scheduling of the success of an event. As an example, if writing a packet into a buffer is defined as an event and the time of finishing is known, the event can be directly scheduled to be successful at that point in time, without needing an associated timeout.

This NewEvent class is used in our framework as the parent class for several custom-defined events, allowing all our models to benefit from this critical enhancement to SimPy.

B. *Limiting the use of processes*

While SimPy processes are a good way to bundle complicated events and adding them to the environment, they can make the program run slower due to the extra events that needs to be processed. Furthermore, it makes the API somewhat non intuitive. As an example, consider a buffer where it takes 4 ticks to write (put) into (for example writing a 128 bit flit into a 32-bit wide FIFO). Figure 7 is an example of building that buffer with the associated put method using a SimPy Store.



```python
from simpy import Store

class Buffer:
    def __init__(self,env,capacity,putDelay=4):

        self.env=env
        self.putDelay=4
        self.store=Store(env,capacity)

    def put(self, item):
        yield self.env.timeout(self.putDelay)
        yield self.store.put(item)
```

Figure 7

In order to simulate the put delay, the put method is made to be a process rather than a simple event as in the original Store class. Consequently, using the put() method should always be through adding a process to the simulation, ie, env.process(buffer.put(item)) which can be hard to read and debug when errors happen.

To reduce the use of processes, we modified SimPy to comprehend timing features and support complex events that are processed by these components in the background. As an example, our Buffer processes a BufferPut event in the background such that it is successful only after a space becomes available AND the time needed to prepare the buffer passes AND the time it takes to actually write the data (which depends on the size of the data and the width of the buffer) passes.

IV. SOC MODELLING FRAMEWORK

A. *What we model*

We model the SoC interconnect and chip-to-chip fabric of the Mk1, Mk2 and future IPUs as a sequence of interconnected SimPy components wrapped within additional python code of our own devising to represent the routing, ingress and egress components of our interconnect plus a number of relevant SoC blocks such as the SerDes links and PCI Express Link, and a block that bridges between our queued SoC Interconnect and our stateless, synchronous and deterministic IPU Core interconnect. In addition we model at a much more abstract level the high level behavior of external components like PCI Express switches and the host CPU memory system and PCI Root Complex.

B. *Graphcore Modelling Framework: how we structure complex hierarchical systems*

Our IPUs are reticle-limited devices containing diverse functionality, and our models are now authored by multiple architects. We have defined a framework on top of SimPy so as to facilitate hierarchical composition of complex models from components authored by different individuals, and to facilitate building models of multiple such devices in order to simulate an IPU, a chassis, rack or multi-rack cluster of IPUs.

Our models are constructed from an hierarchy of re-usable **Components**, as listed above plus function-specific components. Components may be connected together into **Units**, and Units instancing multiple other Units or Components can be constructed in an arbitrary hierarchy to model a design, reflecting the hierarchy of typical chip design processes.

Within our Components or Units, a standard set of **handles** allow access to the upstream and downstream instances (in terms of data flow) within the model. This standardisation of handles minimises re-work to individual Units when a model is restructured. Each Component or Unit has the following standard handles:

FromUp{}/FromDn{} are dictionaries containing references within the unit from which upstream or downstream components can call API methods on this Component or Unit. In a simple component, the From* references point to the component itself.

ToUp{}/ToDn{} are dictionaries containing references from which the Component can call API methods in the upstream or downstream components.

Our Unit base class also supports a **connect()** method which allows internal connections of Components or Units to be made when a Unit is constructed, specifying the source Unit, destination Unit and the source and destination ports, in the case of Units with multiple upstream and/or downstream ports. The connect() method also allows for connections to the top level of the Unit to be established.

When Units are instanced, the connections between the From handles of the Unit to the From handles of the sub-units are made (marked A) and a record is kept of the required connections for the To handles (marked C). See Figure 8 for a depiction of this hierarchy.

When the Unit itself is instanced and connected, the connections marked B are made and the method recurses down implementing the connection of the To handles (marked C).

In addition to the standardised handles to communicate with adjacent Units, our Component APIs are also standardised, with consistent use of put() and get() methods for instance, meaning that changes to our Components or Units are not required when a model is restructured. When an Arbiter component issues a put() call to its downstream handle, it neither knows nor cares whether the downstream component is a Pipeline, a Buffer or even another Arbiter.

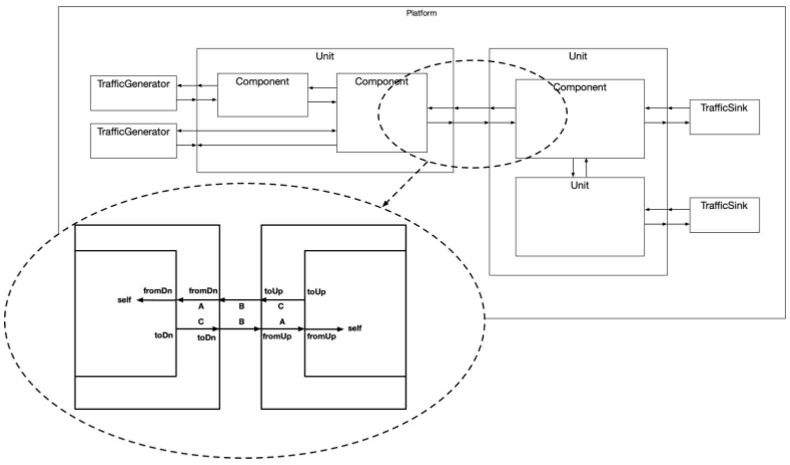


Figure 8

This lego brick approach to system modelling, allows for quick and simple (and automated) restructuring of models. In addition, use of the python *import x as y* functionality allows different implementations of a given component or unit to be swapped in easily. The result of this is that experiments to determine the system-level performance of low level implementation choices such as arbitration schemes can be carried out quickly and efficiently.

At the top level of a model, a **Platform** is constructed. This instances one or more units and allows stimulus scenarios to be defined for execution during simulation of the Platform. We do not cover the definition of stimulus in this paper but hope to present more of our work in this area in the future.

C. Graphcore SimPy Component Library

The following is a list of components that we are open sourcing as part of this paper, along with examples of simple systems built using them.

BasePacket, a generic packet that can be customized to represent data that can be transmitted or stored. It includes a helper function that calculates the time it takes to send/write the packet based on the properties of the downstream.

BaseComponent, a class that defines a unified set of variables common to all components and units and a common interface for logging and **Component**, the smallest connectable unit, extending the BaseComponent to include dictionaries of connections

Unit, extending Component with a unified connect() method to connect components and units in a Recursive manner.

Buffer, into which items can be written and read with customizable, data-dependent timings for how long each operation takes. Statistics are gathered about the access to the buffer. In addition to put and get methods, the buffer presents a peek method that can be used to block until a packet appears in the buffer without having to create a while loop of 1-tick-Timeout for as long as the buffer is empty, significantly improving performance and code efficiency. **FlowControlledBuffer** extends the Buffer with the capability to transmit credits upstream. **CreditBuffer** is a store of credits that can be used to block operations until credits are available. **StoreAndForwardBuffer** customizes the basic Buffer to operate in a store and forward mode where an item cannot be read before being fully written, as opposed to the default cut-through behaviour of the base Buffer

Pipeline, a basic pipeline that can be operated using the standard put method to send items that appear on the downstream after a specified time. Statistics are gathered about the operation of the pipeline (bandwidth).

FlowControlledPipeline extends the basic pipeline with capability to receive credits from the downstream that are checked before initiating any send operation. This is essential for credit-based interconnects which are abundant in SoCs.

Crossbar, a generic crossbar with a customizable number of input and outputs and arbitration between inputs wanting to access the same output port. The crossbar must be connected to buffers on the inputs and operates in a pull mode on the input; it peeks into the buffers and process the packets as they arrive. At the output, it can be operated in either pull mode using a get() method that abstracts all the arbitration and extract a packet from the correct input buffer (according to the arbitration schedule). In push mode, the output is connected to a buffer and the Crossbar will automatically do the arbitration and push the packets on its output buffers as dictated by the arbitration schedule. The class provides easy way to customize routing packets from input ports to output ports along with a separate masking function that requires some event to be triggered before the packet can be presented for arbitration. Finally, it provides an arbitratePkts() method to customize the arbitration for a single output port. It assumes that all packets are ready to proceed on that port and choose one of them according to the details of this function. By default, this method performs a random arbitration.

RoundRobinCrossbar inherits from Crossbar but instead of a random arbitration, implements a work-preserving round robin arbitration.

V. USE OF OUR MODELS

A. Architectural Validation and Debug of In-silico problems

Design of SoC interconnects, especially ones which extend via high speed SERDES and external networks over multiple such devices is a complex task. System behaviors can be difficult to reason about reliably a-priori, and dangers such as deadlocks, livelocks, head of line blocking, buffer over-runs, undesirable under-utilization of resources and unfair arbitration are always lurking in wait for the unwary system designer.

Prior to tape-out of the Mk1 IPU, our model enabled us to detect and resolve a number of head-of-line blocking defects in that architecture during early development. Later, a multi-IPU (Mk2) rack-level system model, combined with some of the visualization features described above, enabled us to quickly replicate deadlock situations that developed on real hardware running distributed machine learning applications over multiple IPUs. We were then able to devise a work around to re-configure the interconnect routing to avoid the cyclic dependencies causing the deadlock and to quickly resolve customer visible issues.

More recently, in our work on future IPU architecture we have been able to find and address instances of unfair arbitration that lead to under-utilization of IO and memory resources. We have also been able to use our framework to create a mock-up of the scheduler contained within third party DDR memory controller IP so as to be able to optimize system performance and determine optimal access patterns for external memory which are now being fed into development of our software tools that compile applications that run on the IPU tile processors.

B. Application Performance Modelling

Our models are used to predict the performance of critical communication patterns that arise in multi-IPU machine intelligence applications. Such patterns typically involve collective operations such as all-gather, broadcast, reduce-scatter etc. Collective operations can be optimally performed on a ring of N nodes, using neighbor only communication. The result is bandwidth optimal since all links between nodes in the ring are fully utilized and no data is sent over the same link twice. This is a canonical pattern in Machine Intelligence applications.

These collective operations form the basis for two basic types of partitioning, Data Parallelism and Model Parallelism, which are encountered in machine learning applications and affect communication intensity and overlap.

Data Parallelism involves having multiple replicas of a model, each trained on separate sets of training data (batches) in parallel. This requires that each model replica periodically averages its learned parameters with the other replicas which requires an all-reduce collective operation between all the replicas.

Model Parallelism involves partitioning a neural network across multiple IPUs, each of which handles part of the computation. This requires collective communications between the IPUs involved. The two main types of Model

Parallelism are Pipeline Parallelism and Distributed Tensor Computation [6]. Pipeline Parallelism involves assigning a subset of the overall network layers that perform a set of computations on their inputs on one node, with the results (activations) being sequentially sent to the next nodes in the pipeline that have different subsets of the layers of the neural network. Distributed Tensor Computation improves efficiency by splitting large tensors such as the inputs to or the parameters within a layer of the model across multiple devices. While there are different ways to define the tensor splits, once again, collectives are needed to obtain the final outputs for each layer in Distributed Tensor Computations and to forward them to the next stage in the model pipeline.

Typically, a large complex application will use a combination of the aforementioned forms of parallelism, which overlap with each other. Our multi-node topology is essentially a torus in 1, 2 or 3 dimensions (ring based collective operations also map optimally onto n-dimensional torus structures) and the operations that are described above for model and data parallel partitioning may use some or all the dimensions of such a torus at once which may often lead to sharing of resources. Our flexible chassis and rack level models have a vital component of our ability to understand the application use cases, the right trade-offs to make in the wider system and software architecture and the best partitioning of large models over multiple IPUs.

VI. CONCLUSION

Thanks to the intuitive API presented by SimPy and the superior engineer productivity offered by python versus possible alternatives, it was possible for a single architect to model the Mk1 IPU very quickly, and then to replicate this basic model of a single chip and its host system to build a model of first a 16 IPU cluster, and then a 64-IPU cluster. Since then as the team has grown and we have worked on new IPU technology generations against a background of a very fast moving application space we have been able to extend our framework to cope with large scale systems and quickly do what-ifs that can ask questions about large scale multi-node systems and provide explanations for their behavior based on a high fidelity underlying model of a single chip.

In our work we have found a number of shortcomings of the default SimPy implementation for system on chip modelling and have developed key extensions to the SimPy framework to alleviate these that we are making available publicly along with implementations for typical system on chip functions such as queues, arbiter and packet-switching.

Finally we have given some real world examples of the efficacy of such models for architectural validation, performance modelling and in-silico debug of large scale systems.

REFERENCES

[1] A. Silva, W. Jose, H. Neto, M. Vestias, “Modeling and Simulation of a Many-core Architecture Using SystemC,” in Conference on Electronics, Telecommunications and Computers , CETC 2013

[2] S. Katare, “A Novel Framework for Modelling High Speed Interface Using Python for Architecture Evaluation,” in 2020 IEEE REGION 10 CONFERENCE (TENCON).

[3] D. Lockhart, G. Zibrat, C. Batten, “PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research,” January 2015

[4] “SimPy – Discrete event simulation for Python,” [Online] Available: https://simpy.readthedocs.io/en/latest/

[5] Z. Jia, B. Tillman, M.Maggioni, DP. Scarpazza, “Dissecting the Graphcore IPU Architecture via Microbenchmarking”, arXiv:1912.03413v1 [cs]

[6] M. Shoeybi, M. Patwary, R. Puri, P. Legresley, J. Casper, B Catanzaro, “Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism,” arXiv:1909.08053v4 [cs]

[7] H.Yassine, G. Cunningham, I. Myttas, D. Wilkinson, “Simpy4chips,” [Online], Available: https://github.com/team-simpy4chips/simpy4chips