IPA: Floorplan-Aware SystemC Interconnect Performance Modeling and Generation for HLS-based SoCs

Nathaniel Pinckney
NVIDIA
Austin, TX, USA
npinckney@nvidia.com

Rangharajan Venkatesan
NVIDIA
Santa Clara, CA, USA
rangharajanv@nvidia.com

Ben Keller
NVIDIA
Santa Clara, CA, USA
benk@nvidia.com

Brucek Khailany
NVIDIA
Austin, TX, USA
bkhailany@nvidia.com

***Abstract*—High-level synthesis (HLS) has recently been used to improve design productivity for many units in today's complex SoCs. HLS tools and flows improve chip design productivity by enabling prototyping and automated implementation of RTL from a single codebase. Although interconnect design is a critical part of today's highly complex SoCs, HLS has not historically been used for SoC-level interconnect. One reason for this is that interconnect architecture and physical floorplan are tightly coupled, and can be difficult to estimate early in the design process.**

To address this gap, we propose *IPA (Interconnect Prototyping Assistant)*, a framework for interconnect prototyping and implementation in HLS-based SoC flows. IPA includes an application programming interface (API) and accompanying tools that automate interconnect modeling and generation for SystemC-based designs. Our framework is used during early architectural prototyping by abstracting specifics of interconnect implementation. IPA then generates interconnect models, including interfaces, for SystemC cycle-accurate simulations. If the design requires long wires between communication units, IPA automatically inserts retiming stages to meet clock frequency targets. IPA's SystemC code is fully HLS-compatible for RTL creation, and thus can be used within a full-chip HLS flow for pushbutton interconnect generation once a design point is selected.

IPA provides accurate architectural performance feedback in minutes and can generate high-quality RTL implementations for SoC interconnect in hours. We demonstrate IPA by exploring the design space for an on-chip interconnect on a micro-benchmark and a deep learning accelerator. Code is available at https://github.com/NVlabs/IPA.

***Index Terms*—high-level synthesis, on-chip interconnect, system-on-chip, network-on-chip, design methodology**

I. Introduction

Modern system-on-chips (SoCs) integrate billions of transistors and feature a multitude of heterogeneous accelerators to improve performance and efficiency, requiring hundreds or thousands of engineering years to develop. Additionally, on-chip interconnect design has become increasingly critical as transistor counts and SoC complexity increase with each process generation, while global wire delay is not improving. As large systems-on-chip and and even larger systems-on-package become more prevalent, these trends will continue to worsen, necessitating tight architectural/physical co-design.

To overcome these increased design complexity challenges, recently proposed high-productivity VLSI flows and libraries [1]–[3] use high-level synthesis (HLS) [4] at full-chip scale. With these flows, architectural models, verification models, and design implementation are combined into a single SystemC source. In this way, a single codebase can be used from early prototyping and design space exploration (DSE) all the way through RTL generation, reducing the need for careful creation and cross-verification of disparate architectural spreadsheet estimations and performance models.

Interconnect design within an SoC is a key challenge, especially during prototyping and DSE, because of interconnect's reliance on an SoC's physical floorplan. For example, network-on-chip (NoC) routers are locally connected to functional units that are physically close on a die, and retiming stages need to be inserted along long wired routes between distant units (see Fig. 1). These physical implementation details become tedious to manually implement and revise while the architecture and interfaces themselves are being prototyped and iterated upon, as area, power, and performance estimates are continually refined.

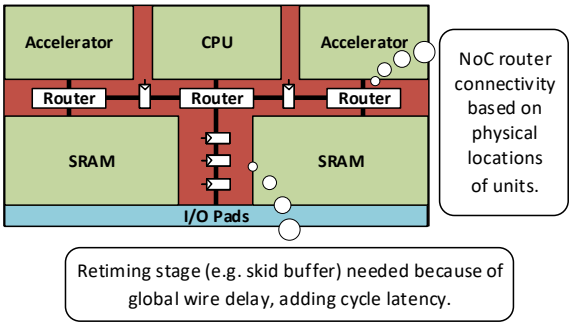


Fig. 1. Mock floorplan of a system-on-chip, with a network-on-chip connecting accelerators, CPU, SRAM, and I/O. Router connectivity is based on physical unit locations, while global wired routes may require retiming stages, such as skid buffers [5], to maintain clock frequency due to far distances.

Despite full-chip HLS flows easing the interconnect implementation burden, HLS flows still require handcrafted interconnect components and instances, along with the accompanying designer overheads of keeping them up to date as the design changes. Because of this, during early DSE and prototyping, the on-chip interconnects may be coarsely modeled or ignored entirely until a particular design point is chosen, even in HLS flows.

This work proposes a SystemC application programming interface (API) and associated tools, collectively called *Interconnect Prototyping Assistant (IPA)*, that infer connectivity and generate HLS-compatible SystemC interconnect models of different interconnect topologies. Taken together, IPA improves architecture/interconnect co-design in full-chip HLS flows.

Key contributions of this work include:

- A SystemC API abstraction layer for SoC-level communication as message passing between units, independent of on-chip interconnect implementation.
- Automatic generation of efficient SystemC and RTL for on-chip interconnects from a high-level specification supporting NoCs, crossbars, and dedicated links, with no manual integration.
- Accurate interconnect performance modeling in cycle-accurate SystemC SoC simulation, without the need for prototyping handcrafted interconnects.
- Floorplan-aware incorporation of inter-unit wire delays, back annotation into SystemC simulations, and automatic insertion of retiming stages for easier timing closure.

II. Related Work

Traditional application of high-level synthesis [4] targets functional units and subblocks, such as datapaths, within a larger RTL-based design, as shown in Fig. 2 (top). In contrast, the open-source MatchLib library [1], [3] helps enable description of all units of an SoC in HLS-compatible SystemC, with almost no hand-written, or hand-modified, RTL code needed for interconnect implementation (Fig. 2, bottom) [1]. This SoC-level flow incorporates quasi-cycle-accurate SystemC simulations of an SoC, including inter-unit interfaces, to better model performance in SystemC simulation with actual interconnect components, such as FIFO buffers and NoC routers from MatchLib [3]. This cycle-accurate approach is complementary to higher-level untimed SystemC simulations such as transaction-level modeling (TLM) [6] and enables more thorough verification and estimation of performance since the same SystemC source code is transformed into RTL during HLS. However, communication channels are manually crafted, and no mechanism exists to automatically include non-idealities in SystemC performance simulations, such as retiming stage latency for long on-chip wires.

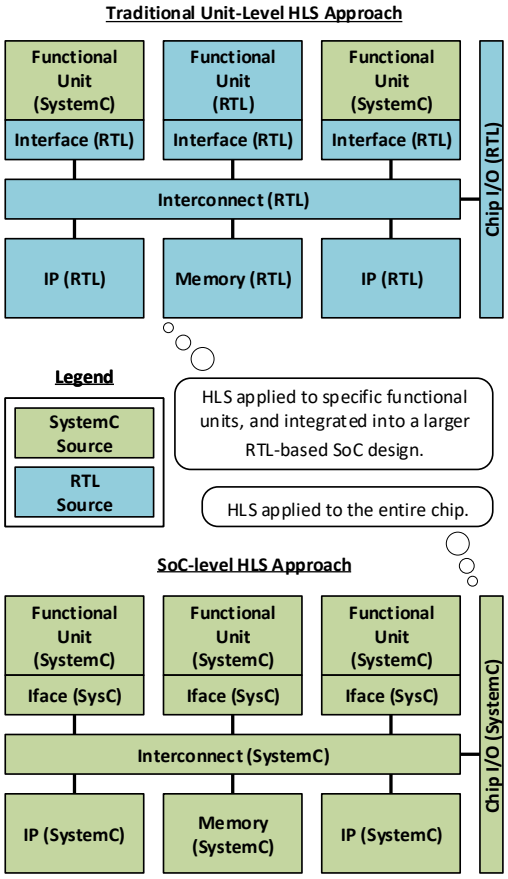


Fig. 2. Traditional HLS flows are focused on the unit-level HLS (top), with manual integration into a larger RTL-based design. SoC-level HLS flows (bottom) describe most of the chip in SystemC.

Many NoC switch and topology modelers and generators exist commercially and in academia [7]–[14]. Booksim [7] and Noxim [8] are two open-source network-on-chip simulators. Booksim is written in C++, supports diverse topology options, and reports many simulated NoC metrics including network latency and throughput. Similarly, Noxim is written in cycle-accurate SystemC and adds energy estimation. Both are intended to be used as standalone models, so neither simulator readily integrates directly into a SystemC-based design for full-chip simulations, nor do the simulators generate RTL directly. While Noxim does estimate on-chip wire energy, it does not include wired route latency in its models.

Interconnect generators typically follow a bottom-up approach: parameters (e.g., number of ports and topology) are first specified, and then the resulting hardware is generated and integrated into the system. For DSE, top-down approaches are desirable: high-level specifications can be given, with implementation details left to the tool. OpenSoC Fabric [9] builds upon Chisel [15] by adding explicit objects such as interfaces and ports to a user-created fabric object. CANAL [10] requires a directed graph of nodes to connect as input, either manually or through scripting. Neither approach integrates into a SystemC-based HLS flow. One top-down approach is NetChip [11], in which an application's traffic is characterized into a graph, mapped to a heterogeneous NoC topology, and compiled into SystemC for simulation and synthesis. However, NetChip's mappings are based on statistical traffic data at a coarse granularity, floorplanning considerations are limited to the NoC components, and NetChip's authors make no mention of integrating with SystemC designs and flows as a whole.

III. Proposed Approach

We propose building upon the MatchLib SoC HLS flow [1] with a new framework and set of tools, collectively known as IPA. IPA contains three components: (1) a SystemC API that hides the underlying interconnect implementation through simple message passing commands, along with definitions of message types, ports, units, and the top-level interconnect object; (2) an IPA Designer tool that reads connectivity data generated from SystemC elaboration, reads/writes a configuration file for the user to describe the desired interconnect and floorplan, and generates interconnect code; and (3) the generated HLS-compatible SystemC interconnect code, including top-level connectivity.

Fig. 3 shows an overview of the proposed flow with the IPA blocks shown in dashed lines added to the MatchLib flow from [1]. The proposed flow begins with all prototyping and initial architectural models in SystemC, including that of cores and functional units, relying on the IPA Library's API to pass messages between units in top-level simulations. The interconnect design space can easily be swept by modifying the IPA configuration file and re-running simulations to tabulate metrics and optimize parameters, with IPA's SystemC models estimating communication latency, capacity, and congestion. As the design space is narrowed to a specific implementation, IPA's generated interconnect code can be transformed into RTL through HLS along with the rest of the design, thus no hand integration of units into a top-level is needed. When the architecture itself is adjusted, IPA easily accommodates new or modified message types, unit types, and instances. IPA lowers the barrier to adoption of SoC HLS flows by abstracting interconnects during prototyping, exploration, and implementation, through the generation of HLS-compatible floorplan-aware SystemC interconnect models.

Each component of our proposed flow is detailed in the following subsections.

A. SystemC Library API

The proposed API provides both a means to express connectivity and message passing intent, and a mechanism to integrate generated interconnect code. An example of the API in abbreviated SystemC is shown in Listing 1 for the Producer-Consumer illustrative design

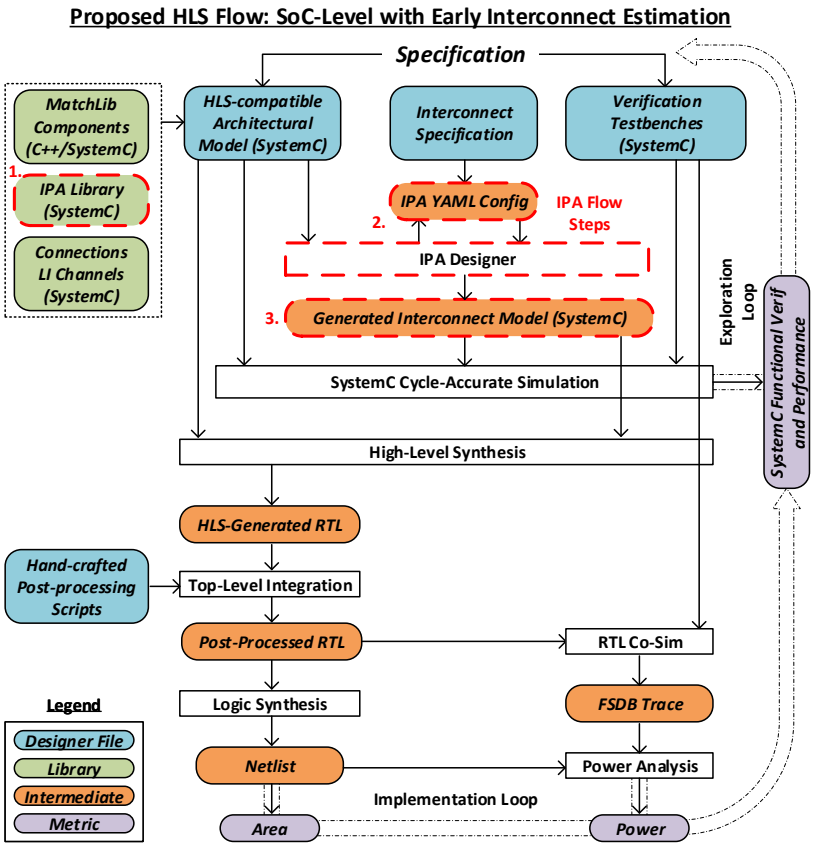


Fig. 3. The proposed IPA flow enables early incorporation of interconnect and wired route latency models into SystemC performance simulation based on high-level interconnect specifications and a preliminary floorplan.

shown in Fig. 4. The API is implemented by a series of macros, listed in Table I, which internally use templated classes, including classes for packetized messages, unit interfaces (arbitration, packetization, and serialization), and the top-level interconnect object.

User code instantiates messages with the `IC_MESSAGE` macro. The argument to `IC_MESSAGE` is a named user-defined message type defined in SystemC with `IC_ADD_MESSAGE_TYPE`, as shown in the top of Listing 1. The header file uses the API to declare a named message type and an associated payload datatype, along with a list of unit classes (reference names) and independent top-level interconnect namespaces in the system. In the listing, the header file declares a message type called `my_msg` that supports a data payload of type `unsigned int`, along with a single unit class type named `PCPart` representing one of the PC unit partitions in Fig. 4.

SystemC units communicate using the state-of-the-art Connections library [2], a latency-insensitive channel library compatible with Mentor Catapult HLS. Instead of directly specifying a payload type, units use the `IC_MESSAGE` macro to reference an interconnect message as the datatype, as mentioned above. In this example, `PCPart` has one input port, `in_port`, and one output port, `out_port`. Data can be read from the input port using a `Pop()` function call, and written with a `Push()`. The `IC_TO` macro along with the << operator is used to specify destinations for the message. If a single destination is specified, the message is sent as unicast, while if multiple << operators are used in sequence it becomes a multicast message. The example in Listing 1 sends an `unsigned int` of 42 to `PC1`'s `in_port`.

TABLE I
IPA LIBRARY API MACROS

Declaration Macros	Description
`IC_ADD_MESSAGE_TYPE (msg_type, data_type)`	Declare a new message type with payload of `data_type`.
`IC_ADD_PARTITION(part)`	Declare a new partition type.
`IC_ADD_INTERCONNECT(ic)`	Declare a new interconnect abstraction.
Message Macros	**Description**
`IC_MESSAGE(msg_type)`	Message object for instantiation or to pass as a template parameter to Connections ports.
`IC_TO(dest_idx)`	Used with << operator to specify destination of a message.
`IC_TO_CONST(dest_idx)`	Preferred over `IC_TO` for fixed destinations.
Unit Macros	**Description**
`IC_BIND_PIN(port_inst)`	Registers a Connections port of the current unit with the API.
`IC_BIND_PORT(port_inst)`	Registers a Connections port of a sub-unit with the API.
Top-Level Macros	**Description**
`IC_BIND_PART(part_inst, dest_idx)`	Registers a unit with the API.

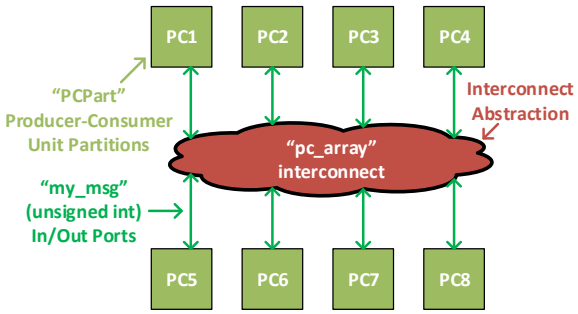


Fig. 4. Example top-level diagram of producer-consumer traffic generation partitions connected to a central interconnect.

LISTING 1
IPA LIBRARY API EXAMPLE

```
// SystemC IPA declarations
IC_ADD_MESSAGE_TYPE(my_msg, unsigned int)
IC_ADD_PARTITION(PCPart)
IC_ADD_INTERCONNECT(pc_array)

// "PC" Partition Declaration:
Connections::Out<IC_MESSAGE(my_msg)> out_port;
Connections::In <IC_MESSAGE(my_msg)> in_port;

// "PC" Partition Constructor:
IC_BIND_PIN(out_port);
IC_BIND_PIN(in_port);

// Receive Message in "PC" Partition Run Loop:
IC_MESSAGE(my_msg) m_in = in_port.Pop();

// Transmit Message in "PC" Partition Run Loop:

// Create message w/ payload 42
IC_MESSAGE(my_msg) m(42);
// add destination "1"
m << IC_TO(1);
// Send the message
out_port.Push(m);

// Top-Level SystemC Module Constructor:
IC_BIND_PART(pc1,1); // inst = pc1, dest idx = 1
// ...
IC_BIND_PART(pc8,8); // inst = pc8, dest idx = 8
```

The interconnect interface is bound to the unit's ports using the `IC_BIND_PIN` macro. Similarly, the system's top-level module binds the partition through `IC_BIND_PART`, which takes as an argument a user-specified destination unit ID for `IC_TO`. The ID as given to `IC_BIND_PART` must be unique and known at compile time, but `IC_TO` is capable of runtime-specified IDs, such as those given by an on-chip configuration register or other logic.

B. IPA Designer and Interconnect Configuration

IPA Designer is responsible for interconnect code generation through reads and writes of a YAML-based interconnect configuration file. YAML is an open human-readable file format supported using libraries by many languages, allowing for easy cross-tool integration [16]. When the SystemC user code is initially compiled and run for the first time, the API constrains the SystemC simulation to run in an elaboration-only mode that records connectivity. The IPA Designer uses this connectivity to write a template of the YAML configuration file for the user to modify, which is then read by the Designer during design exploration and implementation.

Three interconnect types are used to demonstrate the functionality of IPA, as shown in Fig. 5. Directly-connected links dedicate a wire between every pair of units for a given message type, so they are very routing-resource intensive and do not scale well to large designs. Instead, for large designs, direct links serve as a useful "traffic congestion free" baseline. In a central crossbar design, units communicate through an arbitrated crossbar and, since only one link is needed per unit, use fewer routing resources than direct links for moderate numbers of units. However, the crossbar itself will scale poorly, becoming logic-limited with higher radix designs. Lastly, uniform mesh NoCs are supported with XY routing, wormhole unicast [17], and cut-through multicast [18]. NoCs scale the best for large numbers of units, as routers are distributed throughout the design, at the cost of sensitivity to traffic congestion (since router links are shared) and routing complexity. All three types demonstrate common on-chip interconnect topologies to evaluate our proof-of-concept while additional topologies, such as ring, torus, or butterfly could be added in the future.

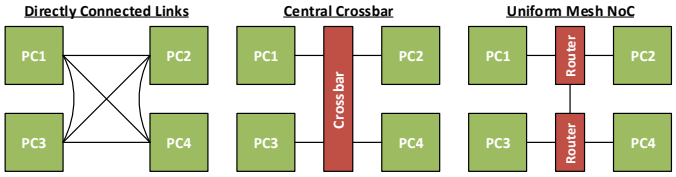


Fig. 5. The three representative interconnect types supported by IPA.

An example YAML configuration file is given in Listing 2, showing the producer-consumer array example with the `my_msg` type assigned to a mesh NoC topology, and the producer-consumer units arranged in a 4×2 array floorplan. The spacing between units is listed as 1 unit-distance along both the x and y coordinates. The NoC topology has a bus (flit) width of 64 bits, a modeled router latency of 2 cycles per hop, a spacing of 1 unit-distance between routers, and a wire propagation speed of 0.5 unit of distance per cycle (2 cycles to transverse a partition). Latency and retiming stages needed for a given wire or router link are estimated by IPA through calculating Manhattan distance between units from the user-specified coordinates in the `unit_instances` section. For NoC-based designs, each wire segment (local port link, remote links between wires, etc.) is separately calculated and summed to estimate latency and capacity, along with an additional `router_latency` per hop. Routers are instantiated and connected, both remote and local links, as appropriate for the floorplan by IPA.

LISTING 2
IPA YAML CONFIGURATION FILE EXAMPLE

```
unit_instances:
  dut_top.pc1: {xcoor: 0, ycoor: 0}
  dut_top.pc2: {xcoor: 1, ycoor: 0}
  dut_top.pc3: {xcoor: 2, ycoor: 0}
  //...
  dut_top.pc8: {xcoor: 3, ycoor: 1}
topologies:
  my_interconnect:
    groups: [my_msg]
    options: {bus_width: 64,
              router_spacing: 1,
              router_latency: 2,
              wire_prop_speed: 0.5}
    type: noc
```

When assigning multiple message types to the same interconnect topology for a crossbar or NoC, IPA Designer generates interface logic and a new packet type that includes bits indicating payload message type. In the case of a NoC topology, the Designer creates serialization and deserialization logic to transform packets into flits sized to match the user-specified `bus_width` in the YAML, as well as routing tables. Directly-connected links are not shared; instead, each message type has a set of dedicated wires between pairs of source and destination partitions with de-multiplex and multiplex logic added at the sender and receiver, respectively, to transmit on the correct link and arbitrate a receiver port between multiple senders.

Independent interconnects can also be specified, such as two independent NoCs. The interconnect configuration supports additional features, including adding extra latency and capacity to the design, the latter of which can be used as implicit buffering along communication channels.

C. Generated SystemC Interconnects

The IPA Designer generates HLS-compatible SystemC code based on the interconnect options and floorplan specified in the YAML configuration file. The API by default (before generation) elaborates the design and implements a simple message passing "magic" interconnect without any introduced latency, that can be used for early functional simulations. The generated interconnect code is a drop-in replacement for the API's default "magic" interconnect and overrides the default implementation through template specializations of the specific message types, partition interfaces, and the top-level interconnect undergoing generation. Since the API remains unchanged from the user's perspective, the generated code seamlessly integrates with the design— no hand integration of the code is needed, and the user-written source code remains unchanged.

The generated interconnect code, including the API, is fully HLS-compatible and the interconnect is intrinsically integrated in RTL, as SystemC models of the entire SoC can be translated together into RTL by HLS tools. Because the generated interconnect code is a cycle- and port-accurate model, it can be used in SystemC cycle-accurate simulations to estimate performance and for functional verification of the design. For example, SystemC simulations with the generated interconnect may expose deadlock conditions due to undersized FIFOs unable to accommodate the capacity needed to hide multi-cycle wire latencies across the chip that would not otherwise be caught until a much later RTL simulation in a traditional flow.

During early architectural exploration, it is often beneficial to model interconnect latency, capacity, and congestion without introducing specific implementation details that may cause deadlock or other functional issues unless the architecture is well-tuned. For these cases, the generated interconnect can be a simple message-passing "magic" interconnect annotated with latency, capacity, and congestion estimated by the IPA Designer. In this alternate mode, the generated code is not HLS compatible and should not be used as a substitute for performance estimation with the real generated interconnect later in the design process, but can be used for early functional verification.

Architectural exploration of performance is accomplished by simulating in cycle-accurate SystemC, revising architectural models and adjusting the interconnect YAML configuration file, and recompiling the simulation binary. Since IPA automatically elaborates connectivity, adding units, ports, or new message types requires minimal changes to the configuration file. Additionally, sweeping the interconnect design space can easily be automated through numerous scripting languages (e.g., Perl, Python, TCL) that can read or write the YAML file format.

IV. Comparison and Discussion

The proposed approach is neither a standalone simulation and modeling framework, nor a standalone network-in-chip (NoC) generator. Instead, it lies at the intersection of both modeling and generation, leveraging modeling (both initial design space exploration and later cycle-accurate simulation) in SystemC and generation through HLS. The advantage of the proposed approach is simplicity: as functional unit prototyping is rolled into hardware generation in a unit-level HLS flow, so is interconnect modeling and generation in the proposed interconnect SoC HLS flow. Combined, functional unit creation and SoC-level interconnect generation can share a single codebase and unified HLS flow.

To evaluate the simulated performance of IPA's generated mesh network-on-chips, we compared throughput and latency of an 8×8 NoC array to BookSim 2.0 [7] and Noxim [8], as shown in Fig. 6. Uniform random traffic generation is evaluated in all three tools. IPA was run in SystemC cycle-accurate simulation, and a producer-consumer traffic generation testbench was instrumented to record packet latency and total number of messages sent within 10,000 clock cycles. Packet injection rate was parameterized and swept.

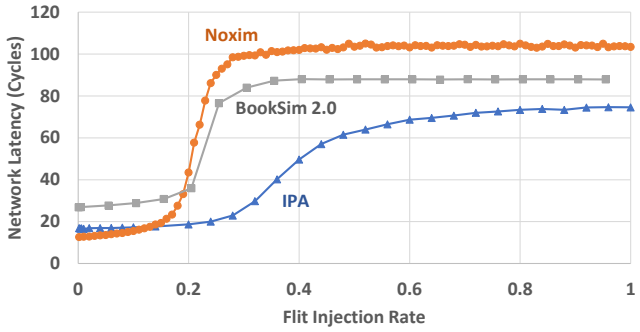
IPA's network latency, Fig. 6a, is similar to that predicted by BookSim 2.0 and Noxim, starting at approximately 17 cycles of average latency at low packet injection rates and plateauing at 75 cycles as the network links and receiver port interfaces become saturated with random traffic. IPA saturates at a higher injection rate than that of Noxim and Booksim 2.0, but Fig. 6b shows that the maximum (worst-case) latency quickly exceeds that of Noxim and Booksim 2.0, due to differences in arbitration schemes for our intended application of IPA. In other words, while the average latency is improved, the worst-case latency is degraded.

Average network throughput (Fig. 6c) is initially linear with injection rate, but then saturates. IPA's NoC saturates at a higher throughput than predicted by Noxim and Booksim 2.0. However, as with IPA's large maximum latency, differences in arbitration schemes may favor less congested links while blocking others. A more balanced arbitration scheme, through the use of a different backend router implementation, could be used when consistent performance among all networked sources and drains is desirable.

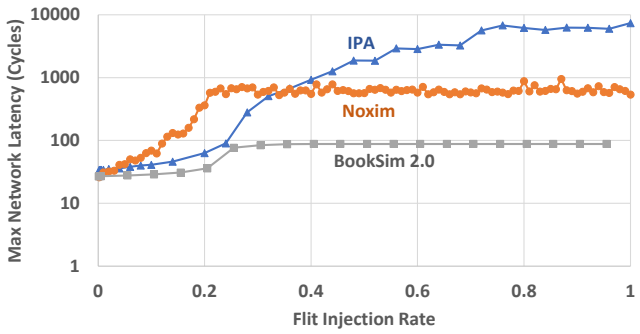
On the generation side, this work re-uses and builds upon existing open-source HLS components [2], [3] to generate the interconnect. However, the proposed technique is not limited to simple mesh NoC topologies or the current router embodiment, but could instead be expanded to more complex or performant HLS-based interconnect component generation.

Table II shows typical runtime for each step of the flow in the deep learning accelerator case study presented in the next section for a 4×4 array interconnect. Under 10 minutes of iteration time is achievable during initial architectural exploration, while implementation of the interconnect through logic synthesis is push button is under six hours. Noxim is typically on the order of seconds of runtime, and Booksim 2.0 on the order of one minute. However, IPA has the ability to integrate in with real workloads as opposed to random traffic patterns, along with the modeling and generation benefits.

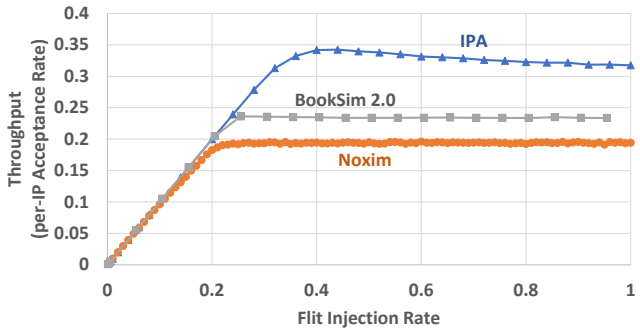
The goal of IPA is to rapidly model designer intent for different representative on-chip interconnect types and to include the impact of a floorplan on overall on-chip interconnect performance, namely latency introduced through global wire delay, culminating in a sane SystemC model and HLS-compatible code. This approach also lends itself to integration with commercial, in-house, or open-source floorplanning and global wire routing backend tools, complementing the frontend prototyping that IPA provides.



(a) Average network latency.



(b) Maximum network latency.



(c) Average network throughput.

Fig. 6. Comparison of IPA's 8×8 mesh NoC modeling versus Booksim 2.0 and Noxim.

TABLE II
TYPICAL RUNTIME FOR IPA FLOW OF 4×4 DL ACCELERATOR

Exploration Loop		Implementation Loop	
Step	**Time**	**Step**	**Time**
IPA Designer	3 minutes	High-Level Synthesis	1.5 hours
SystemC Simulation	5 minutes	Post-Processing	5 minutes
		Logic Synthesis	4 hours

V. CASE STUDIES

IPA's use for improved SystemC performance modeling is demonstrated first in a producer-consumer (random traffic generator) array example similar to Fig. 4, and then expanded to a deep learning accelerator DSE framework. Both examples are fully written in HLS-compatible SystemC and leverage IPA's API, along with the open-source Connections library [2]. The design space was swept across architectural parameters (e.g., number of units, array size, injection rate, etc.) and interconnect parameters (e.g., interconnect type, bus width, etc.), the latter using IPA's interconnect configuration file. No hand modification of the units, top-level models, or interconnect was necessary for these sweeps. Performance was estimated in SoC-level SystemC cycle-accurate simulations.

A. Producer-Consumer Array of Traffic Generators

First, to demonstrate IPA's ability to simulate interconnect type, topology, and wire latency in cycle-accurate SystemC simulations, a 16-unit producer-consumer (PC) traffic generator was implemented using IPA. Each unit generates a message to send to a different random unit with configurable injection rate of up to one message per source PC per cycle. The floorplan of the sample design arranges the PCs in an 8×2 array with a horizontal channel containing the interconnect in the center of the floorplan (see Fig. 7).

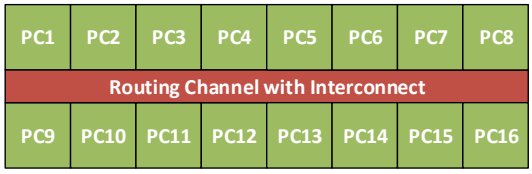
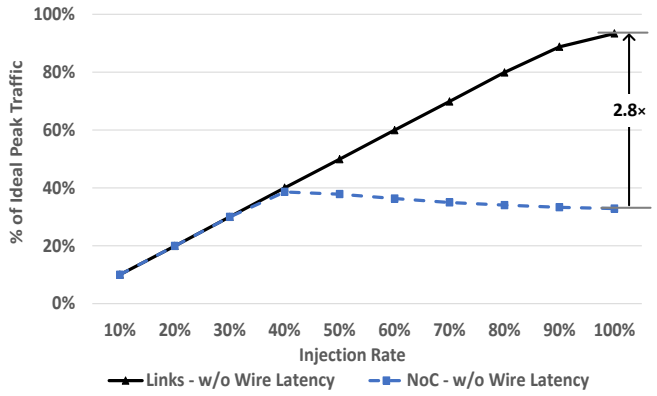


Fig. 7. Floorplan of producer-consumer example of an 8×2 array.

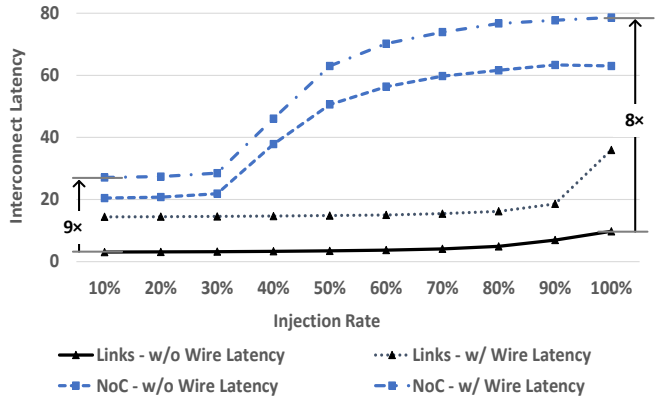
Latency and total traffic received among all units is shown in Fig. 8. Latency measures the number of cycles to transmit messages across the implemented interconnect, while total traffic shows the amount of bandwidth the interconnect can accommodate. The baseline case is directly-connected links without any wire latency— only intrinsic latency and capacity of the port arbitration and de-multiplexing logic. This baseline is compared to a NoC interconnect, with one router for every PC unit partition. The effect of additional cycles of wire latency (4 cycles per PC unit) is also simulated and shown in Fig. 8.

The direct link baseline achieves throughput of up to 93% of peak traffic, still falling 7% short of ideal because of contention at receiver ports. However, with a NoC the number of messages diverges from the baseline at around a 40% injection rate as the router network's cross-sectional bandwidth becomes saturated. Average latency remains flat in the link scenario except when injection rate is very high and messages are delayed due to receiver congestion. In contrast, the generated NoC case sees dramatic increases in average latency as the producer injection rate increases beyond 30% and the network saturates. The modeled wire delays further increase latency.

As this example shows, modeling interconnects in performance simulations is critical, as the simulated bandwidth and latency may be significantly over or underestimated when not considering interconnect specifications and wire latency. IPA allows the user to accurately model such changes with minimal effort. Without interconnect performance modeling, the bandwidth and latency simulated would be inaccurate in SystemC, and a user would need to measure performance in slower RTL simulations after running HLS.



(a) Throughput vs. injection rate for direct links and NoC. Results with wire latency included are omitted as wire latency did not significantly impact bandwidth.



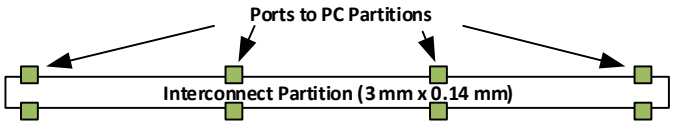
(b) Latency vs. injection rate for direct links and NoC with and without wire latency. Wire latency is modeled as 4 cycles of delay per producer-consumer partition width.

Fig. 8. Throughput and latency for producer-consumer case study.

B. Retiming Insertion

IPA-generated SystemC interconnect code, including long-wire retiming stages, was evaluated through place-and-route to assess the efficacy of retiming insertion on long wired routes between communication units. The interconnect for a 4×2 producer-consumer array in a sub-16nm production process technology was synthesized into RTL using Mentor Catapult HLS, and then physically implemented using Synopsys Design Compiler and Synopsys IC Compiler 2 (ICC2). Pin and retiming stage cell placement was accomplished through scripts specific to the design, as would be typical in traditional flows. Interconnect type was set to directly-connected links to stress wired route length. The relative wire delay for retiming stage insertion was swept in the interconnect configuration from zero retiming stages to three stages per producer-consumer partition (i.e. 9 total for the 3 mm horizontal array length). Timing was reported through ICC2's built-in static timing analysis after placement, clock tree synthesis, and routing. The placement locations of retiming stages are shown in Fig. 9, along with the reported relative max path delay (minimum clock period) for increasing numbers of retiming stages.

When no retiming stages are present, the long wired routes' slack severely limits obtainable clock speed because data must traverse from one side of the interconnect partition to the other within a single clock cycle. As retiming stages are added, worst-case path delay reduces significantly, a 67% reduction from the baseline case when three retiming stages are added per PC partition. However, each retiming stage represents an additional cycle of latency for communication. Thus, adding retiming stages, like any pipelining technique, trades clock frequency for cycle latency.



(a) Floorplan of producer-consumer (PC) place and route experiments.

(b) Zero retiming stages. (relative max path delay = 100%)

(c) One stage per PC partition length. (relative max path delay = 51%)

(d) Three stages per PC partition length. (relative max path delay = 33%)

Fig. 9. Distribution of repeater cells and max path (setup time) delay, relative to the zero retiming stage case, within a 3mm × 0.14mm interconnect partition for the 4×2 producer-consumer array as reported by ICC2.

C. Deep Learning Accelerator

To demonstrate the utility of IPA, we prototyped a deep learning accelerator (DLA) based on [19] and studied the impact of different interconnect topologies and parameters on the scalability of the designs. Fig. 10 shows the architecture diagram of the DLA. It is a spatial array architecture consisting of an array of processing elements (PEs), an array of global buffers (GBs), and a controller, all interconnected using IPA. PEs perform the compute-intensive operations and consist of an array of multiply-accumulate (MAC) units along with local memory buffers for storing weights, input activations, and output partial sums. GBs act as second-level storage and are banked to provide sufficient bandwidth to the PEs.

Different types of messages are also shown in Fig. 10. To execute a deep neural network (DNN) inference layer, the controller sends an AXI configuration to program the control registers in PEs and GBs. Weights are then transferred to the PEs, which stay stationary in the PEs throughout the execution. PEs then send `DataReq` messages to request input activations from appropriate GBs, after which input activations are streamed from GBs to PEs. A matrix-vector product operation is performed across a weight matrix and an input activation vector in the PEs to produce a partial sum vector every cycle. Partial sum vectors are temporally reduced across multiple cycles in each PE and then streamed to other PEs for spatial reduction through `PartialSum_In` and `PartialSum_Out` messages. To coordinate the spatial reduction operations, PEs send and receive `PartialSumReq_In` and `PartialSumReq_Out` messages. Once the reduction operation is complete, post-processing is performed to compute `OutputActivation`, which is stored back to GB. Finally, PEs and GBs send a `Done` signal to the controller to signal the end of computation.

To study the scalability of the DLA, the design is parameterized with a variable number of PEs and GBs in SystemC. PE SystemC fully implemented the DL accelerator design specification and is HLS compatible. The floorplans arrange the PEs into rectangular

arrays (4×4, 8×4, and 8×8) and arrange the GBs into two floorplan configurations as shown in Fig. 11: a centralized monolithic GB along the right edge of the floorplan area (similar to [20]) or distributed among the PEs. GB bandwidth is equivalent in both scenarios.

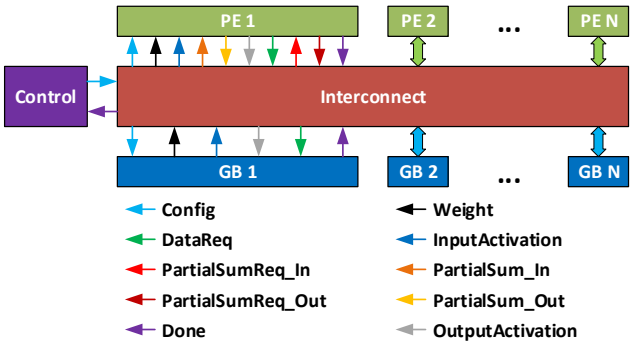


Fig. 10. Deep learning accelerator (DLA) architecture.

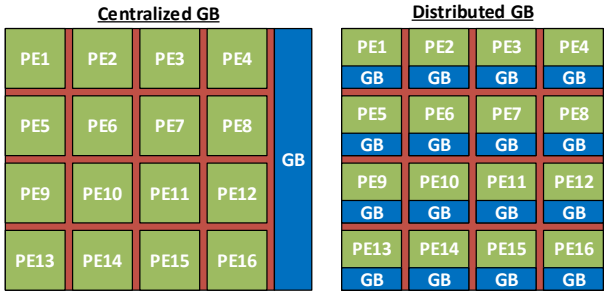


Fig. 11. Floorplan of DLA with centralized (left) and distributed (right) global buffers. Red is the routing channel area.

IPA's three representative interconnect types are considered to study impact on the deep learning accelerator's performance: direct links, a crossbar, and a uniform mesh NoC. Direct links for a large design are largely an architectural exploration baseline to estimate an upper bound on performance if wire routing resources are unlimited, but latency from wire delay due to physical distance between units is still included. In the NoC case, each router is arrayed one per PE, and for the distributed GB case the router is shared between both the PE and GB. Each PE is designed to perform 256 MAC operations per cycle with 4-bit precision for weights and activations. Evaluation is performed in a sub-16nm FinFET technology node, and Mentor Catapult HLS synthesizes the SystemC code to RTL, against which the SystemC performance models were validated in co-simulation. The parameterized DLA is evaluated using different convolutional layers from ResNet-50 using the ImageNet dataset [21].

Fig. 12 shows IPA's ability to compare system performance of ResNet-50 neural network layers on a 16-PE DLA SoC with different interconnect topologies and latencies, enabling architects to perform quick DSE for identifying challenging workloads. As shown in the figure, layers with different shapes exhibit different sensitivity to interconnect topology and latency. Layers with high memory-to-compute ratio (e.g., `res2a_branch2a`) benefit from high bandwidth and low latency interconnects, while those with low memory-to-compute ratio (e.g., `res5a_branch2b`) can effectively exploit data reuse to hide long latencies. Fig. 13 shows the performance scalability study with increasing number of PEs. By optimizing the interconnect, we achieve 3500 images/sec with a 4-bit precision, 64 PEs, and 256 MACs per PE design operating at 1.1 GHz compared to 2548 images/sec achieved by [19] when the GB is centralized. If the GB is distributed, performance further increases by 14% to 4000 images/sec. The direct link baseline is higher still, but is not physically realizable.

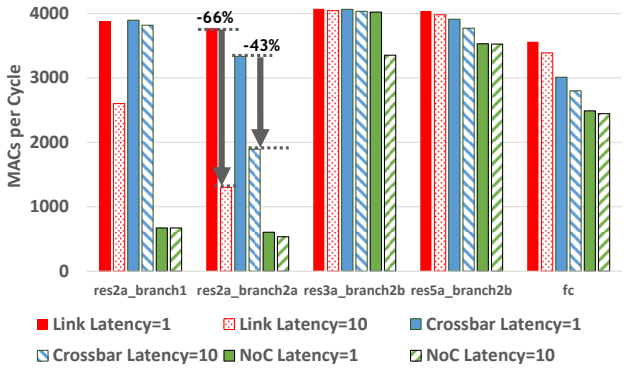


Fig. 12. Sensitivity analysis for different topologies and interconnect latencies for selected ResNet-50 layers in a 16-PE array. Some layers are sensitive to latency and topology selection, others are not.

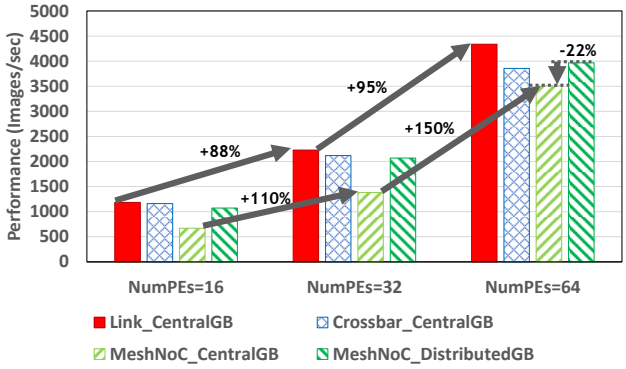


Fig. 13. ResNet-50 on the deep learning accelerator performance scaling with different interconnect topologies and floorplan.

VI. Conclusions

IPA lowers the barrier to adoption of SoC-level HLS flows by providing an API and set of tools for SystemC performance simulations to include the impact of interconnects and long wire latency without the need for hand-crafted interconnect models. IPA is useful in the early architectural exploration and prototyping phase of a design, and during implementation. The proposed approach generates HLS-compatible SystemC code to aid in RTL creation once a design point has been selected. This RTL is then used in a traditional gate synthesis and place-and-route flow to provide detailed area and power estimates. Future work includes expanding the feature set, scaling to larger systems (including 3D-integrated SoCs), and automating the downstream flow.

Acknowledgment

This material is based upon work supported by DARPA under Contract No. HR0011-18-3-0007. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

REFERENCES

[1] B. Khailany, E. Khmer, R. Venkatesan, J. Clemons, J. S. Emer, M. Fojtik, A. Klinefelter, M. Pellauer, N. Pinckney, Y. S. Shao, S. Srinath, C. Torng, S. L. Xi, Y. Zhang, and B. Zimmer, "A Modular Digital VLSI Flow for High-productivity SoC Design," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: ACM, Jun. 2018, pp. 72:1–72:6. [Online]. Available: http://doi.acm.org/10.1145/3195970.3199846

[2] "HLSLibs Homepage for a faster path to hardware acceleration." [Online]. Available: https://hlslibs.org/

[3] "NVlabs/matchlib," Mar. 2021, original-date: 2019-03-18T17:59:52Z. [Online]. Available: https://github.com/NVlabs/matchlib

[4] M. C. McFarland, A. C. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 301–318, Feb. 1990, zSCC: 0000812.

[5] Intel, "Flow Control with Skid Buffers," in *Intel® HyperflexTM Architecture High- Performance Design Handbook*, 2020, p. 56.

[6] M. Montoreano, "Transaction Level Modeling using OSCI TLM 2.0," Synopsys, Inc., White Paper, May 2007.

[7] N. Jiang, D. Becker, G. Michelogiannakis, J. Balfour, B. Towles, J. Kim, and W. Dally, "A detailed and flexible cycle-accurate Network-on-Chip simulator," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2013, pp. 86–96.

[8] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti, "Cycle-Accurate Network on Chip Simulation with Noxim," *ACM Transactions on Modeling and Computer Simulation*, vol. 27, no. 1, pp. 1–25, Nov. 2016. [Online]. Available: https://dl.acm.org/doi/10.1145/2953878

[9] F. Fatollahi-Fard, D. Donofrio, G. Michelogiannakis, and J. Shalf, "OpenSoC Fabric: On-Chip Network Generator: Using Chisel to Generate a Parameterizable On-Chip Interconnect Fabric," in *Proceedings of the 2014 International Workshop on Network on Chip Architectures*. Cambridge United Kingdom: ACM, Dec. 2014, pp. 45–50, zSCC: 0000010. [Online]. Available: https://dl.acm.org/doi/10.1145/2685342.2685351

[10] R. Bahr, C. Barrett, N. Bhagdikar, A. Carsello, R. Daly, C. Donovick, D. Durst, K. Fatahalian, K. Feng, P. Hanrahan, T. Hofstee, M. Horowitz, D. Huff, F. Kjolstad, T. Kong, Q. Liu, M. Mann, J. Melchert, A. Nayak, A. Niemetz, G. Nyengele, P. Raina, S. Richardson, R. Setaluri, J. Setter, K. Sreedhar, M. Strange, J. Thomas, C. Torng, L. Truong, N. Tsiskaridze, and K. Zhang, "Creating an Agile Hardware Design Flow," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, Jul. 2020, pp. 1–6, zSCC: 0000002 ISSN: 0738-100X.

[11] D. Bertozzi, A. Jalabert, Srinivasan Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli, "NoC synthesis flow for customized domain specific multiprocessor systems-on-chip," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 2, pp. 113–129, Feb. 2005. [Online]. Available: http://ieeexplore.ieee.org/document/1374853/

[12] J. Öberg and F. Robino, "A NoC system generator for the Sea-of-Cores era," in *Proceedings of the 8th FPGAWorld Conference on - FPGAWorld '11*. Copenhagen and Stockholm and Munich, Denmark and Sweden and Germany: ACM Press, 2011, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2157871.2157875

[13] H. Kwon and T. Krishna, "OpenSMART: Single-cycle multi-hop NoC generator in BSV and Chisel," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Santa Rosa, CA, USA: IEEE, Apr. 2017, pp. 195–204. [Online]. Available: http://ieeexplore.ieee.org/document/7975291/

[14] A. Gangwar, N. K. Agarwal, R. Sreedharan, A. Prasad, S. H. Gade, and Z. Xu, "Automated synthesis of custom networks-on-chip for real world applications," in *Proceedings of the 39th International Conference on Computer-Aided Design*. Virtual Event USA: ACM, Nov. 2020, pp. 1–9. [Online]. Available: https://dl.acm.org/doi/10.1145/3400302.3415656

[15] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *DAC Design Automation Conference 2012*, Jun. 2012, pp. 1212–1221, zSCC: 0000611 ISSN: 0738-100X.

[16] "YAML Ain't Markup Language (YAML™) Version 1.2," Oct. 2009. [Online]. Available: https://yaml.org/spec/1.2/spec.html

[17] W. J. Dally and C. L. Seitz, "The torus routing chip," *Distributed Computing*, vol. 1, no. 4, pp. 187–196, Dec. 1986. [Online]. Available: https://doi.org/10.1007/BF01660031

[18] P. Kermani and L. Kleinrock, "Virtual cut-through: A new computer communication switching technique," *Computer Networks (1976)*, vol. 3, no. 4, pp. 267–286, Sep. 1979. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0376507579900321

[19] R. Venkatesan, P. Raina, Y. Zhang, B. Zimmer, W. J. Dally, J. Emer, S. W. Keckler, B. Khailany, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, and N. Pinckney, "MAGNet: A Modular Accelerator Generator for Neural Networks," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Westminster, CO, USA: IEEE, Nov. 2019, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/8942127/

[20] B. Zimmer, R. Venkatesan, Y. S. Shao, J. Clemons, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. S. Emer, C. T. Gray, S. W. Keckler, and B. Khailany, "A 0.32–128 TOPS, Scalable Multi-Chip-Module-Based Deep Neural Network Inference Accelerator With Ground-Referenced Signaling in 16 nm," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 4, pp. 920–932, Apr. 2020.

[21] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2009, pp. 248–255, zSCC: 0027726 ISSN: 1063-6919.