Bringing Programmability to the Data Plane: Packet Processing with a NoC-Enhanced FPGA

Andrew Bitar, Mohamed S. Abdelfattah, Vaughn Betz
Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada
{bitar, mohamed, vaughn}@eecg.utoronto.ca

***Abstract*—Modern computer networks need components that can evolve to support both the latest bandwidth demands and new protocols and features. To address this need, we propose a new programmable packet processor architecture built from an FPGA containing an embedded Network-on-Chip (NoC). The architecture is highly flexible, providing more programmability than is possible in an ASIC-based design, while supporting throughputs of 400 and 800 Gb/s. Additionally, we show that our design is 1.7× and 3.2× more area efficient, and achieves 1.5× and 3.7× lower latency than the best previously proposed FPGA-based packet processor on complex and simple applications, respectively. Lastly, we explore various ways a designer can take advantage of the flexibility available in this architecture.**

I. Introduction

Computer networks have seen rapid evolution over the past decade. "Cloud computing" and the "Internet of Things" are becoming household terms, as we move to an era where computational power is offloaded from the PC and onto data centers located miles away. This surge in demand on networking capabilities has led to new network protocols and functionalities being created, updated and enhanced. The implementation of these protocols and functionalities has proven challenging in current network infrastructures, causing a demand for "programmable networks".

Software-Defined Networking (SDN) is a proposed networking paradigm that provides programmability by configuring network hardware (i.e. the "data plane") through a separate software-programmable "control plane" [1]. Various APIs have been developed to interface the control plane with the data plane, with OpenFlow [2] being a popular one in the academic community. However, the degree of programmability provided by these APIs is limited by the capabilities of the data plane. If new protocols are developed with functionalities that go beyond what is available in the hardware, then expensive hardware replacements will still be necessary.

Field-programmable gate arrays (FPGAs) have long provided a hardware-programmable alternative to fixed ASIC designs. The reconfigurability of FPGAs seems like a natural solution to the demand for programmable networks; FPGA designs can be re-programmed in hardware to serve network evolution. However, what FPGAs gain in programmability they lose in performance; Kuon and Rose quantified the FPGA's programmability overhead to be 18×-35× in area and 3×-4× in critical path delay compared to ASICs [3]. As a result, one would expect FPGAs to struggle with efficiently supporting the high bandwidth demands of modern computer networks.

This is generally true; as a whole, established network infrastructures are largely dominated by ASICs. FPGA designs have made some breakthroughs in various networking applications [4–7], thanks to the increasing transceiver bandwidth available in modern FPGAs (Figure 1). Although the transceivers

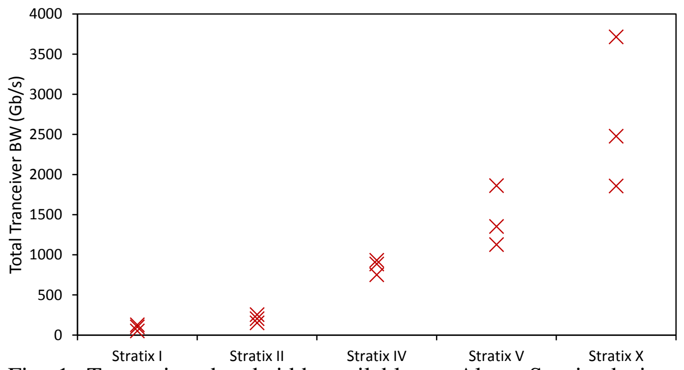


Fig. 1: Transceiver bandwidth available on Altera Stratix devices has rapidly grown with every new generation. The three data points for each generation correspond to the device models with the three highest transceiver BW.

can bring the high bandwidth data onto the chip, these designs still rely on buses made of the FPGA's soft reconfigurable interconnect to move the data across the chip. These buses are slow (typically 100-400 MHz) and therefore must be made very wide, consuming high amounts of interconnect area and creating challenging timing closure problems for the designer. Thus, it should come as no surprise that FPGAs have yet to be widely adopted throughout network infrastructures.

Recent work has looked closely at this FPGA interconnect problem and has argued for the inclusion of a new system-level interconnect in the form of a Network-on-Chip (NoC) [8]. Hardening such a NoC in the FPGA's silicon would provide a fast, chip-wide interconnect that consumes a small fraction of the FPGA area [8]. Such a NoC-enhanced FPGA has the potential to better transport high-bandwidth transceiver data in networking applications. Prior work has already shown that an efficient and programmable Ethernet switch can be built from a NoC-enhanced FPGA that far outperforms previous FPGA-based switch designs [9].

In this work, we consider another fundamental network building block: the packet processor. In general terms, a packet processor is a device that performs actions based on the contents of received packetized data. The flexibility of this unit is imperative to the future programmability of computer networks. Recent works [4, 10–12] have proposed various ways to build packet processors that support SDN/OpenFlow, with the majority using match tables that can be configured to support various types of processing (Section II-B). Our design takes a different route. By interconnecting multiple, protocol-specific processing modules through a NoC embedded in an FPGA, we develop a new form of packet processor design that provides a high degree of flexibility for network evolution. In effect, our design brings programmability directly to the data plane.

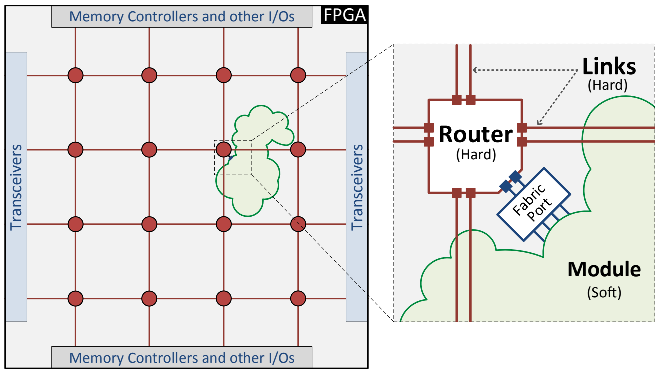
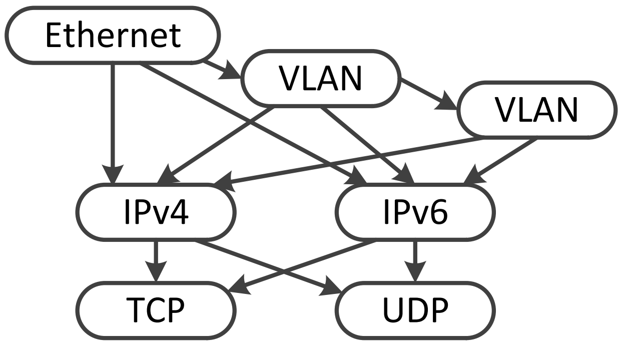
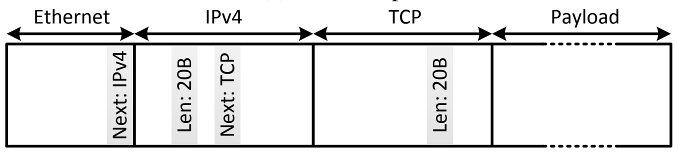


Fig. 2: A NoC-Enhanced FPGA. The embedded NoC is implemented in hard logic, and connects to modules on the FPGA through a FabricPort [8].



(a) Parse Graph



(b) TCP/IP Packet

Fig. 3: An example parse graph commonly seen in enterprise networks, and a packet belonging to that parse graph. A parse graph can be used to represent the combinations of protocols that a packet processor may see in a certain application.

Our focus is to explore this new form of packet processor design. To this end, we make the following contributions:

1) Propose a new packet processor architecture that maximizes hardware flexibility while efficiently supporting modern network bandwidths (400G and 800G).
2) Evaluate the architecture by implementing common packet parsing and processing use-cases.
3) Compare its resource utilization and performance to the best previously proposed FPGA packet processor.
4) Explore how a designer can take advantage of this architecture's flexibility.

II. Background

A. NoC-Enhanced FPGA

Existing FPGAs contain only fine-grained programmable blocks (lookup-tables, registers and multiplexers) from which an application's interconnect can be constructed. Previous work has proposed the inclusion of an embedded NoC to augment this fine-grained interconnect so as to better interconnect an FPGA application. Embedded NoCs have been shown to improve both area and power efficiency, and ease timing closure compared to traditionally-used soft buses configured from the FPGA's programmable blocks [13–15].

Figure 2 shows an embedded NoC on the FPGA. The NoC routers and links run approximately four times as fast as the FPGA fabric; therefore, we use a special component called the FabricPort to bridge both width and frequency in a flexible way between the embedded NoC routers and soft FPGA modules [8]. Furthermore, the processing modules connected to the NoC need not operate using the same clock frequency or phase – the FabricPort essentially decouples the modules connected together through the NoC. The routers are packet-switched virtual-channel routers that perform distributed arbitration and switching of packets coming from modules connected to the NoC. These routers also contain built-in buffering and credit-based backpressure, and so can automatically respond to bursts of data and heavy traffic on its links while maintaining application correctness.

The NoC we use has been evaluated on a large 28-nm Stratix V FPGA, and can run at 1.2 GHz in that process technology [8]. The NoC links are 150 bits wide each and so can transport up to 180 Gb/s in each direction between any two routers, of which we have 16 as illustrated in Figure 2. Because it is implemented in efficient hard logic, this NoC only consumes 1.3% of the core area of an Altera Stratix V FPGA. The FabricPort for this NoC can connect a module running at any FPGA frequency and any width between 1 and 600 bits to the NoC routers at 150 bits and 1.2 GHz. It does so using a combination of clock-crossing and width adaptation circuitry that formats data into NoC packets in an efficient way [8].

B. Match Table Packet Processors

Depending on the application, a packet processor must be able to process packets of a certain set of protocols. These protocols vary depending on the type of packet and layer they represent on the OSI stack. For example, Ethernet is one of the most common layer 2 protocols found in modern networks. The protocol at each layer contains information regarding the type and location of the protocol found in the next higher layer (Figure 3b). The various different combinations of protocols that may be received in a single packet can be represented by a parsing graph [11] (Figure 3a).

As network protocols are changed, added and replaced, a programmable packet processor must be able to be reconfigured to support different parsing graphs. The OpenFlow standard comes with specifications for architecting an SDN-compatible packet processor [16]. The architecture is based on a cascade of match-action "flow" tables: memory holding all possible values of relevant packet header fields, and instructions for what action to be taken once matched. Figure 4 illustrates the high level design of this architecture. The pipeline begins by matching fields whose locations in the packet header are known. When a field is matched, the entry contains information regarding where to look in the next match table for the protocol of the next layer, as well as any other corresponding actions to be taken. This model requires match tables to contain all possible combinations of header protocols and field values that the processor may face. All of the tables in the pipeline are populated by the control plane, such that lookups in a Table j can depend on information from any Table i so long as $i < j$.

There are two important limitations to this programmable packet processor design. When built in an ASIC, the number

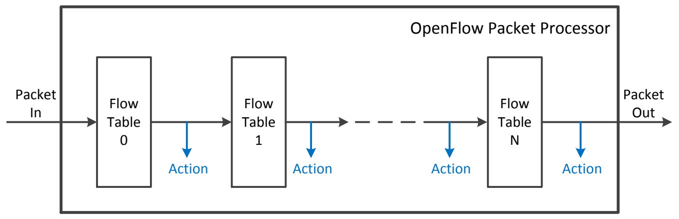


Fig. 4: High-level overview of OpenFlow's programmable packet processor architecture [16].

of tables along with their width and depth are fixed upon chip fabrication. Consequently, should a new protocol be created that requires fields that are larger than the table width, a number of entries that exceeds the table depth, or more pipeline stages than are available, then the packet processor can not be configured to support such a protocol. Similarly, the action set available is also set upon fabrication. Any new actions required by newly developed protocols could not be supported.

Recent work has revealed two possible ways to address these limitations. Bosshart *et al.* proposed building in additional programmability into an ASIC-based packet processor through the use of ternary content-addressable memory (TCAMs) [10]. Their design – called RMT (Reconfigurable Match Tables) – can be viewed as an upgrade to the OpenFlow model. It involves mapping logical match (or flow) tables to physical ones by allowing a logical table to use resources from multiple pipeline stages. This mapping process allows the width and depth of flow tables to be customized for specific sets of protocols. Additionally, their design implements a reduced instruction set that, when used in combinations in a very-long instruction word, can execute a wider array of actions than OpenFlow.

The RMT design mitigates, but does not solve OpenFlow's limitations. As the authors admit in the paper, their design is still restricted by the number of pipeline stages, table sizes, and action units made available to the chip upon fabrication. These restrictions cannot be avoided when building a design from ASIC technology. In contrast, Attig and Brebner have previously shown that a programmable packet parser can be built on an FPGA that supports 400 Gb/s bandwidth [4]. Their design – which they refer to as PP[1] – uses a design style similar to OpenFlow, with a cascade of processing stages containing tables with entries that encode what fields to extract from a packet. Using an FPGA allows their design to be reconfigured to support different table configurations and actions.

Our programmable packet processor design also leverages FPGA technology to avoid the limited flexibility of ASIC-based designs. However, unlike the PP design, we elect to avoid using a similar approach to OpenFlow, which was originally proposed to bring programmability to ASIC-based network infrastructures. We instead embrace the full reconfigurability of the FPGA, creating a fully-programmable packet processor that is more efficient than the FPGA-based PP design and more flexible than the ASIC-based RMT design.

III. The NoC Packet Processor

A. Design Overview

Instead of match tables that each support a set of protocols, our design implements multiple processing modules, with each module dedicated to processing a single parsing graph node's protocol (Figure 3a). Packets are sent to the modules corresponding to the protocols found in their headers. Each processing module determines what actions to take for its protocol, and the type and location of the protocol processing module for the packet's next OSI layer. Figure 5 depicts a general representation of this packet processor design.

The FPGA reconfigurable fabric is used to implement the processing modules. The flexibility of the fabric allows for the modules to be fully customized and later updated, as existing protocols are enhanced and new protocols are added. It is by updating the modules that processing rule updates are made. For example, in order to modify the supported parse graph, each module's routing decision logic is updated to match the new set of edges connecting its corresponding node in the graph. Module updates are made by reconfiguring (or partially reconfiguring, see Section V-B) the FPGA.

As in any packet processor, the processing modules perform some set of actions when packet fields are matched. However, since each processing module is dedicated to a single node in the parsing graph, its matches and actions are tied solely to what is relevant to that node. Field matches and routing decisions at a particular protocol node can be stored in tables at the module. These tables only need to contain entries relevant to actions that can take place for that protocol, unlike the OpenFlow flow tables, which must have entries for all possible field matches at that stage. In effect, a flow table is broken up and spread across several processing modules. The module tables' widths and depths, as well as the module's actions, can be customized to exactly fit the requirements of the module's protocol. We argue that having these "fine-grained" tables that are specifically customized for a protocol will allow for a more efficient design than OpenFlow's "coarse-grained" flow tables, as illustrated by the following two examples.

Example 1: Consider a programmable packet processor that reads the IP address from either an IPv4 or IPv6 header in order to make a packet routing decision. In an OpenFlow implementation, the addresses to be matched from both the IPv4 and IPv6 header must be contained in a single match table (or copied over multiple tables). This table must therefore be sized to fit the entries containing the 128-bit IPv6 addresses, causing entries for IPv4 addresses (only 32 bits) to have at least 96 wasted memory bits per entry. In our separated processing module architecture, the IPv4 and IPv6 modules contain their own respective tables that are sized perfectly to fit the width of their corresponding match fields.

Example 2: Suppose a programmable packet processor is built to currently support only Ethernet processing, but will, in future, need to process other protocols as well. Using OpenFlow's match-action style of design would require over-provisioning the architecture with wider and deeper tables, and more actions and processing stages than are currently needed for just Ethernet. On the other hand, NoC-PP effectively implements a "use only what you need" design style: the architecture can be provisioned for exactly the type of processing it currently needs, and can be updated for future needs by adding/updating processing modules through partial reconfiguration. The NoC-PP design can begin with only instantiating Ethernet processing modules, while adding new processing modules later as more protocols are introduced.

[1]Attig and Brebner use "PP" to refer to the language used to program their architecture. For simplicity, we shall use PP to refer to their design as a whole.

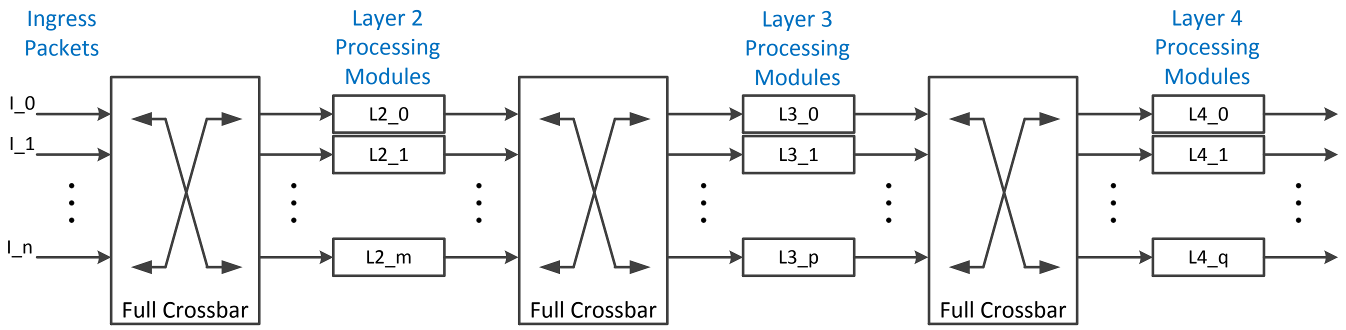


Fig. 5: High level representation of module-based packet processor design. Packets are switched between processing modules corresponding to the protocols found in their headers.

B. The NoC-Crossbar

Interconnecting the processing modules is a challenging problem on current FPGA devices. Multiplexers, which are necessary for the full crossbars in Figure 5, are synthesized poorly on the FPGA's fabric, often consuming relatively high chip area and running considerably slower than they would if built in ASIC technology. A full crossbar that is wide enough to support the very high bandwidth of data entering modern packet processors would be very difficult, if not impossible, to efficiently synthesize in the FPGA fabric.

To address this problem, we draw inspiration from previous work that used an embedded NoC in an FPGA as a crossbar for an Ethernet switch [8, 9]. This NoC – described in Section II-A – can function as the full crossbar in our packet processor design. Not only is it already designed to transfer packets across the FPGA, it also includes a built-in flow control mechanism that can handle scenarios of adversarial traffic, such as prolonged bursts of packets. For example, if a certain processing module is busy and cannot accept packets from the NoC, the NoC will hold packets destined for that module in a buffer at the connecting router. Should that buffer become full, packets will then be buffered at downstream NoC routers. If the NoC cannot accept packets at one of its routers due to a buffer being full, then it can also send a backpressure signal to the modules connected to that router.

With a capacity of 180 Gb/s at each of its links, the NoC can transport high bandwidth data throughout the FPGA, to and from processing modules in our design. Processing modules are connected to NoC routers, as illustrated in Figure 6, with the NoC's FabricPort used to bridge the frequency of the processing modules and the frequency of the NoC (see Section II-A). Moreover, we can combine multiple modules at a single router node using the FPGA's soft logic for arbitration. For the remainder of the paper, we shall refer to this NoC packet processor as "NoC-PP".

C. Inter-Module Information Passing

Thus far, we have described a mechanism for processing protocols with modules that are independent of one another. However, packet processing requires at least some information to be passed from one protocol to another. As a bare minimum, information regarding where the next protocol's header is located in the packet is determined in the previous protocol header and therefore must be passed to the next protocol's processing module (see Figure 3b). There are many other possible scenarios where information from a lower-level

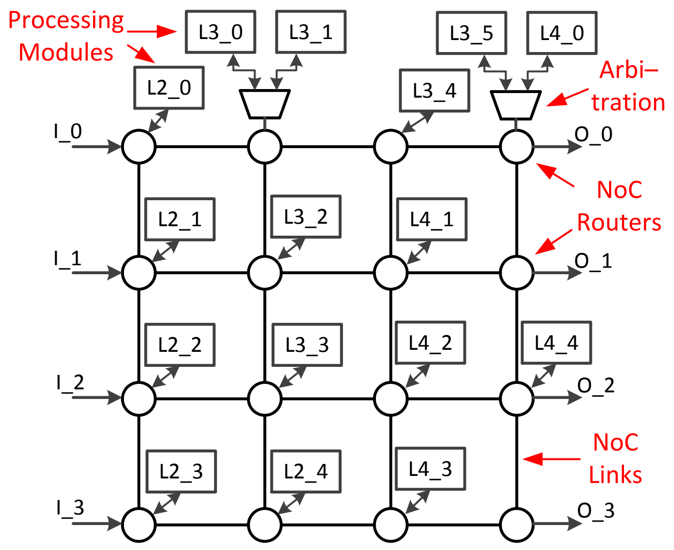


Fig. 6: The "NoC-PP" architecture: the embedded NoC serves as the crossbar for the module-based packet processor design. NoC routers and links are "hard", i.e. embedded in the chip, while the processing modules and arbitration logic are synthesized from the FPGA "soft" fabric.

protocol may be used later in the processing stages. For example, in packet classification, a classification decision may be made after a combination of fields are parsed from each of the packet's headers.

To provide a mechanism for information-passing between processing modules, the NoC-PP design adds a blank header flit to every packet upon entering the processor. This header flit can hold any combination of parsed fields from the packet as it proceeds through the processor. Data stored in the header flit are maintained across processing modules until they are overwritten. For example, our design stores a 7-bit "data offset" field in this header flit that is updated by every processing module. This field tells the next processing module where its header is located in the packet. The header flit is removed just before the packet exits the processor.

D. Design Example: Eth-IPv4/IPv6-TCP

In order to evaluate this design, we implemented a packet processor that supports processing of several common network protocols: Ethernet, VLAN, IPv4, IPv6, and TCP. Table I lists the features implemented in each of the protocol processing modules. Each packet going through the processor will visit a processing module for each protocol found in its header. For example, consider an Ethernet/IPv4/TCP packet. The

TABLE I: Functions implemented in each protocol processing module in our NoC-PP design.

Protocol Module	Implemented Processing Functions
Ethernet/VLAN	1. Parse MAC source and destination 2. Maintain a MAC table 3. Extract priority code identifier (PCP) in VLAN header 4. Determine layer 3 protocol from Ethertype
IPv4	1. Compute checksum and drop packet if results in error 2. Decrement time to live (TTL) and drop packet if zero reached 3. Determine total length of header 4. Parse source and destination IP addresses 5. Determine layer 4 protocol
IPv6	1. Decrement hop limit and drop packet if zero reached 2. Parse source and destination IP addresses 3. Determine layer 4 protocol
TCP	1. Parse source and destination ports 2. When receiving a request to establish a connection, generate and send a reply ACK message over TCP/IP

packet is first brought onto the chip through the FPGA's transceivers, which perform clock recovery and serial to parallel data conversion. It is then transported to the embedded NoC through the FPGA's soft interconnect, where the FabricPort (Section II-A) performs clock conversion to bring the data from the slower FPGA fabric to the fast NoC. The NoC's links and routers steer the packet to a router connected to an Ethernet processing module, where the FabricPort again performs clock conversion to bring the packet back to the FPGA fabric. Once Ethernet processing is complete and the next layer protocol is determined, the packet is then brought back to the NoC to be sent to an IPv4 processing module, and finally a TCP processing module, before being sent back out through the FPGA's transceiver.

The processing modules are designed with a data path width of 512 bits running at 215 MHz, providing an overall processing throughput of 100 Gb/s.[2] There are two possible ways for this design to support a higher network bandwidth: increase the supported throughput of the individual processing modules or instantiate multiple instances of the processing module throughout the NoC. In our design, we duplicate the 100G processing modules to provide higher overall processing bandwidths, such as 400G and 800G (Figure 7). This ability to duplicate individual processing modules provides a key form of design flexibility not found in previously proposed packet processor designs. If a designer is aware of the expected frequencies of traffic of each of the different protocols, then he/she can duplicate the different processing modules to the appropriate degree. For example, if IPv4 packets are currently much more frequent compared to IPv6 packets, then the design could instantiate more IPv4 processing modules than IPv6. This design flexibility is explored in Section IV-C.

Besides module duplication, the NoC-PP design also allows for the easy addition and/or removal of protocols, thanks to the logical and physical decoupling of processing modules by the NoC. Say a new protocol has been developed and must be supported by the packet processor. A processing module for this protocol can then be designed and added to the NoC-PP by connecting it to any of the routers in the NoC, with little other modification to the rest of the design. Similarly, if a protocol no longer needs to be supported, then its processing modules can simply be removed from the design.

[2]Raw throughput is 110 Gb/s, with 10 Gb/s of capacity used for the information-passing header added to each packet.

IV. Evaluation

A. Simulation and Synthesis Setup

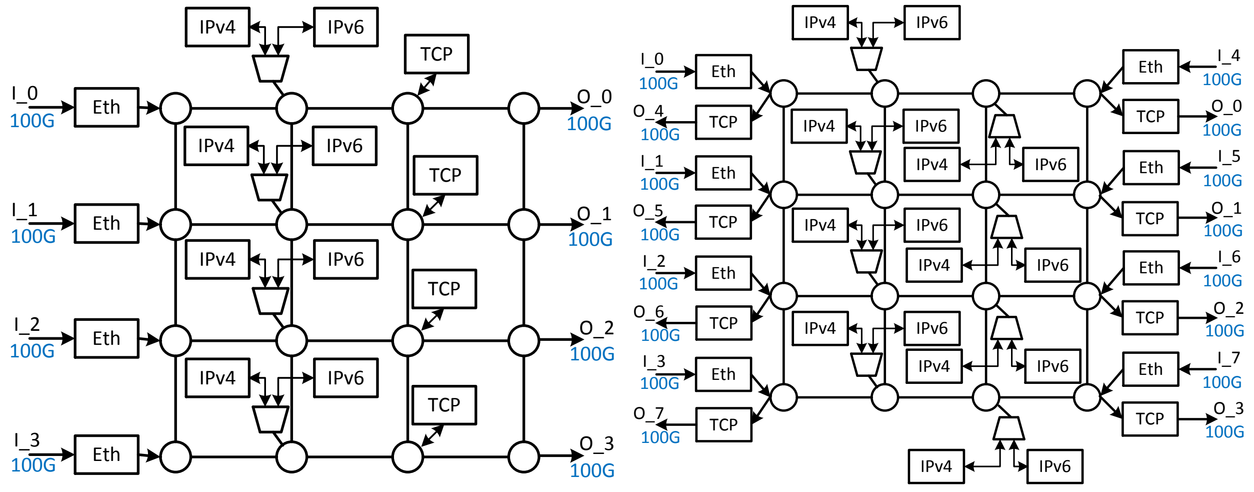
We use **`RTL2Booksim`** [17] to connect the packet processing modules to the embedded NoC, and perform a cycle-accurate simulation in ModelSim. This allows us to gather our performance data in cycle counts; however, we also need to find the operating frequency of our packet processing modules to be able to quantify the overall performance of NoC-PP. Furthermore, we want to realistically model any physical design consequences of connecting these modules to embedded routers (such as interconnection choke-points or possible frequency degradation). To do so, we emulate the existence of routers in an Altera FPGA (Stratix V 5SGSED8K1F40C2) by creating 16 design partitions – one for each router – that have the same size, location and interconnection flexibility as an embedded router. We then connect our packet processing modules to them the same way they are connected in NoC-PP and compile the design in Quartus II v14.0 to collect the area and frequency results presented in this section.

B. Design Efficiency

We measure the hardware cost and performance of the NoC-PP design and compare it to Attig and Brebner's PP design. PP, however, only provides packet parsing functionality. It extracts fields from the packets but does not perform any form of action after the extraction, besides determining where the next header is located. Consequently, we also synthesized a modified version of the NoC-PP design that only contains parsing functionality, thus providing a fair comparison. We compare to two versions of the PP design: (1) the smallest, "JustEth", which only performs parsing on the Ethernet header, and (2) one of their biggest, "TcpIp4andIp6", which performs parsing on Ethernet, IPv4, IPv6 and TCP [4].

Table II contains hardware cost and performance results of the NoC-PP and PP designs. Hardware cost is measured using resource utilization as a percentage of an Altera Stratix V-GS FPGA. Attig and Brebner's experimental results, originally presented as a percentage of a Xilinx Virtex-7 870HT, are converted to equivalent Stratix V-GS numbers. To perform this conversion, we use equivalent logic element/logic cell counts on each device, which we have found to accurately reflect the logic capacity for both vendors across a large number of designs. The resource utilization results for the NoC-PP design also include the resources consumed by RAM blocks and the embedded NoC [8]. The table also contains results for the full packet processor described in Section III-D at 400G and 800G, referred to as "TcpIp4Ip6-Processor" (illustrated in Figure 7).

Overall, the NoC-PP proves to be more resource efficient and achieves better performance compared to the PP architecture. For the smaller application (JustEth), the NoC-PP design is 3.2× more efficient, whereas for the larger application (TcpIp4Ip6), it is 1.7× more efficient. NoC-PP also reduces latency by 3.7× and 1.5× compared to PP for JustEth and



(a) 400G (b) 800G

Fig. 7: The NoC PP design for an Ethernet/VLAN/IPv4/IPv6/TCP packet processor (Eth=Ethernet+VLAN). Processing modules run at 100G, and are instantiated four times to support 400G processing, or eight times to support 800G processing.

TABLE II: Comparison of the NoC-PP and PP architectures

Application	Architecture	Resource Utilization (% FPGA)	Latency (ns)	Throughput (Gb/s)
JustEth	NoC-PP	3.6%	79	400
	PP [4]	11.6%	293	343
TcpIp4Ip6	NoC-PP	9.4%	200	400
	PP [4]	15.6%	309	325
TcpIp4Ip6 -Processor (400G)	NoC-PP	14.4%	230	400
TcpIp4Ip6 -Processor (800G)	NoC-PP	25.8%	232	800

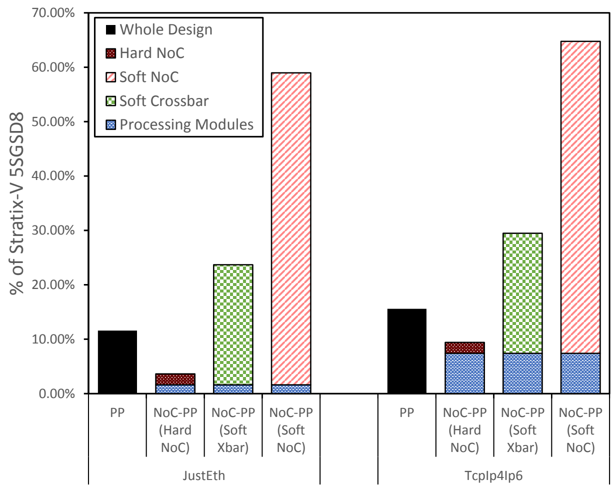


Fig. 8: Area breakdown of NoC-PP when using a hard NoC, a soft NoC or a soft custom crossbar (Xbar).

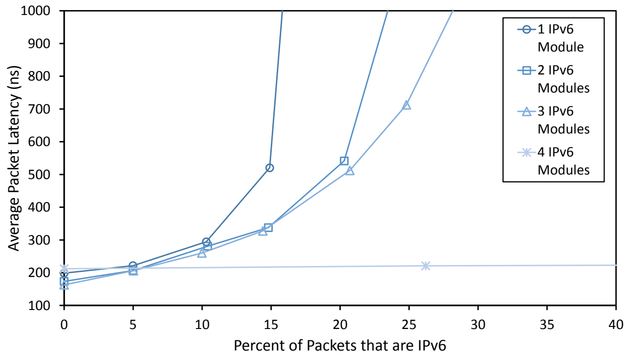
TcpIp4Ip6, respectively. Table II also shows that our full-featured packet processor is still more efficient than the more basic packet parser presented in prior work. Lastly, it is worth noting that the 800G design achieves a throughput greater than any previously reported packet processor built from an FPGA. Thus, the module-based NoC-PP architecture provides significant advantages for FPGA packet processors.

It is also important to determine what brings these efficiencies to NoC-PP; is it the new module-based packet processor architecture, the introduction of the hard NoC, or a synergistic fusion of the two? To answer this question, we began by replacing the hard NoC in our design with an equivalent soft NoC, and separately quantified the cost of the NoC and the processing modules. We also built another iteration of our design using three customized soft crossbars (as in the block diagram shown in Figure 5), each with a radix of eight and containing 10-flit deep buffers at the ports in order to handle scenarios of backpressure. As can be seen in Figure 8, the costs of the soft NoC and the soft crossbar are 29× and 11× greater than that of the hard NoC, respectively. The significantly higher cost of building NoC-PP's interconnection network out of the reconfigurable FPGA fabric is due to the fact it runs at a considerably lower clock frequency compared to the hard NoC and must therefore use wide datapaths to transport the high bandwidth data. Switching between these wide datapaths requires large multiplexers and wide buffers that consume high amounts of resources. The design therefore achieves significant savings by hardening this interconnect in the embedded NoC.

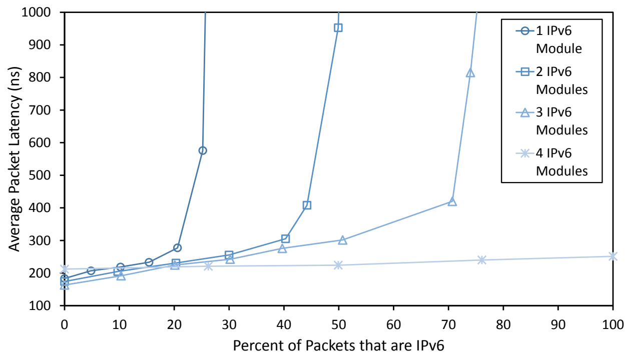
Since the processing modules form a small fraction of the design cost when using a soft interconnect, NoC-PP can therefore achieve significant overall savings when replacing the soft interconnect with a hard NoC. On the other hand, the PP design uses a feed-forward design style. Rather than switching between protocol modules, PP uses tables containing "microcode" entries for all possible protocols that must be processed at that stage. Thus, no wide multiplexing exists in the design that can be efficiently replaced by a hard NoC. The logic and memory within each stage form the majority PP's hardware cost, which would not change if a hard NoC was introduced. We therefore conclude that the efficiencies from NoC-PP stem from a synergistic fusion of using the hard NoC with our module-based packet processor architecture.

C. Design Flexibility

The highly modular design of NoC-PP allows for the easy addition and removal of protocol processing modules. This flexibility can be used to introduce new network protocols, modify protocol processing modules, and duplicate existing



(a) NoC Router Buffer Depth = 10 flits



(b) NoC Router Buffer Depth = 64 flits

Fig. 9: Average packet latency through the NoC-PP design as a function of percentage of total traffic using IPv6, for four different degrees of IPv6 module duplication. Four IPv6 modules (each running at 100G) is sufficient to fully support any rate of IPv6 traffic running at 400G.

protocol modules in order to support higher bandwidths. The TcpIp4Ip6-Processor design example duplicates each 100G processing module four and eight times in order to fully support 400G and 800G processing, respectively, for each protocol. Depending on the application, full duplication of each processing module may not be necessary. It is possible, for example, that a low percentage of the traffic seen by the packet processor uses IPv6[3]. The designer therefore has the flexibility to reduce the number of IPv6 processing modules in order to free up chip area for other modules.

To assess the impact of module duplication on performance, we use **`RTL2Booksim`** to simulate the design with varied degrees of duplication of the IPv6 processing module, swept across percentage of the total traffic that is IPv6. A bursty, 400G traffic pattern is used to emulate real Internet traffic, with packet payload size set to be 512 bytes. The design is first simulated using a NoC consisting of the same parameters (link width, mesh radix, router buffer depth) as those chosen in previous work [8]. Figure 9a illustrates the results. The "knee" in each curve indicates the maximum percentage of IPv6 that can be supported at 400G; after this point, the NoC saturates, causing the source to be frequently backpressured.

Originally, it was expected that the design would be able to support up to a fraction of IPv6 packets equal to the number of IPv6 modules implemented divided by four, given it takes four modules to fully support a 100% IPv6 packet rate (i.e. 25% for one IPv6 module, 50% for two, etc.). However, we see in Figure 9a that the NoC saturates at a much earlier point in each curve. This can be attributed to the fact that the NoC is not an entirely non-blocking crossbar; packets compete for resources and when one does not receive access to its desired link, it must be held in buffers within the NoC. In this case, the NoC routers have buffers that can hold 10 NoC flits [8], while each injected packet forms 40 NoC flits. Thus, when one packet is waiting for a NoC link to become free, it must be stored in four buffers, consequently congesting four upstream routers. In the case when four IPv6 modules are implemented, each injection point has its own IPv6 module for its IPv6 packets. Thus, there is no competition for resources in the NoC. As soon as there is a mismatch in number of injection points and number of IPv6 modules, packets may compete for resources when two are sent to one IPv6 module at the same time.

[3]As of July 2015, Google measures that only ∼7% of accesses to their website are through IPv6 [18].

One way to mitigate this congestion is to increase the buffer depth in the NoC routers, as has been studied in previous work [19]. In our application, if a router's buffer could hold an entire packet, rather than just a quarter, it would prevent upstream routers from becoming congested as well. We test this theory by adjusting the simulated NoC buffer size to 64 flits, rather than 10. As can be seen in Figure 9b, increasing the buffer size mitigates the NoC's blocking nature, allowing it to saturate according to the availability of IPv6 processing bandwidth in the design.

The impact of increasing the buffer size of the NoC on its hardware cost is illustrated in Figure 10. Increasing NoC buffer size allows the NoC-crossbar to provide more switching capability for larger packets at a moderate hardware cost. However, this scaling technique requires architectural changes to an embedded NoC. Thus, it is crucial that a manufacturer of a NoC-enhanced FPGA appropriately provisions the NoC to serve potential applications. By looking at important FPGA networking applications such as switching [8, 9] and packet processing, we hope to better guide the architectural choices of an embedded NoC.

Overall, we see that the flexibility that NoC-PP provides to support reprogrammability for network evolution surpasses that of any conceivable ASIC-based packet processor. NoC-PP is only limited by the amount of resources available on the entire FPGA. Unlike the RMT design, NoC-PP is not limited by the table sizes and action set established upon chip fabrication. New actions and match field values can be reprogrammed into new or existing processing modules at any time. Modules can be modified, replaced and even duplicated by reconfiguring the FPGA fabric. Not only does NoC-PP manage to exceed the flexibility of the ASIC-based RMT, but it also matches and surpasses RMT's supported bandwidth of 640G.

V. Design Enhancements

A. *Virtual Channel Priority Scheme*

Virtual Channels (VC) in a NoC are separate FIFO buffers located at every router port. They allow packets arriving at or being sent along a common physical link to be stored in separate buffers. The embedded NoC used in our design employs two VCs, as previous work has shown that NoC congestion is reduced by ∼30% when a second VC is used [21]. With two VCs, data flowing through the packet processor can be sorted into two groups of traffic. This can be especially useful for establishing a packet priority scheme. Packet priority

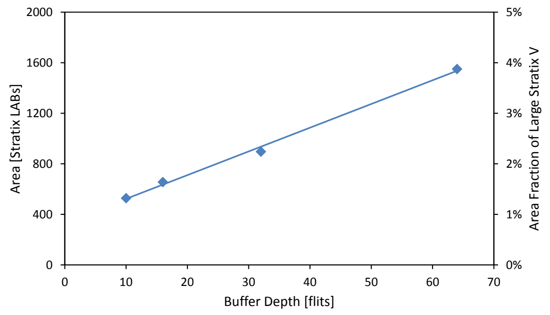


Fig. 10: Embedded NoC chip area, measured in equivalent Stratix LABs and fraction of a Stratix V-GS, scaling with buffer depth given a fixed link width of 150 bits [20].

is prevalent in many different modern network protocols, including VLAN and TCP. In NoC-PP, one VC can be reserved for packets given priority based on the contents of their headers, while all other traffic is routed along the other VC. Thus, priority packets can bypass non-priority packets when the NoC is congested, thereby giving priority packets a faster path through the design.

B. Partial Reconfiguration

A key advantage to OpenFlow's flow table architecture is the capability of updating processing rules "on-the-fly". In other words, the table entries can be updated with new match-action rules even while the processor is receiving packets. This is trickier in the NoC-PP architecture. Although NoC-PP may still contain tables that can perform soft updates, its decreased reliance on memory means that more of its processing has been moved to dedicated logic in the FPGA fabric. Typically, modifying an FPGA design requires complete reconfiguration of the device, during which the transceivers cannot accept any data. Making processing rule updates through a complete reconfiguration would require the packet processor to be effectively "paused".

Partial reconfiguration [22] presents a potential solution to this limitation. Unlike the monolithic PP design, NoC-PP has both logically and physically partitioned processing modules. The modules share a common interface and are decoupled by the NoC, making the design highly amenable to partial reconfiguration of individual processing modules. While soft flow table updates allows for processing rule changes, partial reconfiguration makes both rule and architecture changes possible. To perform any architecture changes to PP, such as modifying table sizes or the action set, a complete reconfiguration would be necessary. On the other hand, NoC-PP can partially reconfigure individual processing modules while the remaining modules remain operational. Thus, partial reconfiguration would open up the possibility of modules being updated/added/removed while the packet processor remains "live". An investigation into partial reconfiguration in a NoC-enhanced FPGA is beyond the scope of this paper and is left for future work.

VI. Conclusion

As network performance becomes ever more crucial to data processing and mobile applications, there is an ever-increasing demand on network infrastructures to evolve. The NoC-PP packet processor architecture proposed in this work focuses on maximizing hardware flexibility, thus providing a platform that is very amenable to network evolution. By using an FPGA augmented with an embedded NoC, we have proposed a modular packet processor design that is capable of being reprogrammed to support different combinations of actions and header field matches. Unlike ASIC-based OpenFlow architectures, it is not limited by the action set and table sizes fixed upon chip fabrication. Its flexibility significantly exceeds that of the RMT ASIC architecture, while matching, and even surpassing, its supported bandwidth (400G/800G vs. 640G). Compared to the best previously proposed FPGA-based packet processor, NoC-PP proves to be 1.7× and 3.2× more resource efficient, and achieves 1.5× and 3.7× lower latency on complex and simple applications, respectively.

The NoC-PP architecture is only possible given the inclusion of a hardened NoC embedded into an FPGA. This hard NoC is both fast and small, enabling easy cross-chip communication. Though an embedded NoC has yet to be adopted in FPGAs, we hope that the compelling applications explored in this and previous work [8, 9] present a convincing case for its inclusion in future FPGA devices. Such a NoC-enhanced FPGA could revolutionize SDN by paving the way for a fully-programmable data plane.

Acknowledgments

This work is funded by Altera, NSERC and Vanier CGS.

References

[1] B. Nunes *et al.*, "A survey of software-defined networking: Past, present, and future of programmable networks," *Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.
[2] N. McKeown *et al.*, "Openflow: enabling innovation in campus networks," *SIGCOMM*, vol. 38, no. 2, pp. 69–74, 2008.
[3] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *TCAD*, vol. 26, no. 2, pp. 203–215, 2007.
[4] M. Attig and G. Brebner, "400 gb/s programmable packet parsing on a single FPGA," in *ANCS*. IEEE Computer Society, 2011, pp. 12–23.
[5] S. Byma *et al.*, "FPGAs in the cloud: Booting virtualized hardware accelerators with openstack," in *FCCM*. IEEE, 2014, pp. 109–116.
[6] S. Zhou, Y. R. Qu, and V. K. Prasanna, "Large-scale packet classification on FPGA," in *ASAP*. IEEE, 2015.
[7] S. Mühlbach *et al.*, "Malcobox: Designing a 10 gb/s malware collection honeypot using reconfigurable technology," in *FPL*. IEEE, 2010, pp. 592–595.
[8] M. S. Abdelfattah, A. Bitar, and V. Betz, "Take the highway: Design for embedded NoCs on FPGAs," in *FPGA*. ACM, 2015, pp. 98–107.
[9] A. Bitar *et al.*, "Efficient and programmable ethernet switching with a NoC-enhanced FPGA," in *ANCS*. ACM, 2014, pp. 89–100.
[10] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *SIGCOMM*, vol. 43, no. 4. ACM, 2013, pp. 99–110.
[11] G. Gibb *et al.*, "Design principles for packet parsers," in *ANCS*. IEEE, 2013, pp. 13–24.
[12] J. Naous *et al.*, "Implementing an OpenFlow switch on the NetFPGA platform," in *ANCS*. ACM, 2008, pp. 1–9.
[13] M. S. Abdelfattah and V. Betz, "Networks-on-Chip for FPGAs: Hard, Soft or Mixed?" *TRETS*, vol. 7, no. 3, pp. 20:1–20:22, 2014.
[14] ——, "Power Analysis of Embedded NoCs on FPGAs and comparison With Custom Buses," *IEEE TVLSI*, 2015.
[15] ——, "The Case for Embedded Networks-on-Chip on Field-Programmable Gate Arrays," *IEEE Micro*, vol. 34, no. 1, pp. 80–89, 2014.
[16] Open Network Foundation, "OpenFlow Switch Specification."
[17] M. S. Abdelfattah *et al.*, "Design and simulation tools for embedded nocs on fpgas," in *FPL*. IEEE, 2015.
[18] Google IPv6 Connecitivity Statistics. [Online]. Available: https://www.google.ca/intl/en/ipv6/statistics.html
[19] M. Coenen *et al.*, "A buffer-sizing algorithm for networks on chip using TDMA and credit-based end-to-end flow control," in *CODES*. ACM, 2006, pp. 130–135.
[20] M. S. Abdelfattah, "NoC Designer," 2013. [Online]. Available: http://www.eecg.utoronto.ca/~mohamed/noc_designer.html
[21] M. S. Abdelfattah and V. Betz, "The Power of Communication: Energy-Efficient NoCs for FPGAs," in *FPL*, 2013, pp. 1–8.
[22] C. Kao, "Benefits of partial reconfiguration," *Xcell journal*, vol. 55, pp. 65–67, 2005.
[23] Video Over IP Reference Design. [Online]. Available: www.altera.com