Exploring Domain-Specific Architectures for Network Protocol Processing

Artem Ageev
MPI-SWS
Saarbrücken, Germany
aageev@mpi-sws.org

Mehrshad Lotfi Foroushani
MPI-SWS
Saarbrücken, Germany
mlotfi@mpi-sws.org

Antoine Kaufmann
MPI-SWS
Saarbrücken, Germany
antoinek@mpi-sws.org

***Abstract*—Existing architectures for network protocol processing incur bottlenecks in communication-intensive cloud applications. Software processing competes for scarce CPU cycles with applications, fixed protocol offloads such as RDMA lack required flexibility, and SmartNICs are inefficient, expensive, or power-hungry. The design space for alternative architectures remains largely unexplored, and choosing a suitable architecture is complicated by rapidly evolving application and transport protocols. We draw parallels to machine learning acceleration and argue for a similar approach: configurable domain-specific architectures (DSAs) combined with a domain-specific high-level programming model decoupling applications from architectures. We propose Kugelblitz, an implementation and design exploration framework for network DSAs enabling flexible and efficient protocol offload. Kugelblitz comprises abstractions for separately specifying hardware configuration and protocols, a hardware RTL generator, and a compiler to generate runtime configurations to implement specific protocols on specific hardware configurations.**

***Index Terms*—Protocol offload, packet processing, hardware acceleration, domain-specific architectures, design exploration.**

I. Introduction

Cloud systems fundamentally rely on efficient network communication — for scaling across multiple servers, accessing back-end services, and communicating with the outside world. While network bandwidth steadily increases, the required transport and application layer processing also proportionally consumes more processor cycles. Offloading part of this processing to dedicated hardware on the network interface controller (NIC) is a proven approach to reduce CPU overhead. For example, even basic NICs include functionality to compute and validate protocol checksums, most data center NICs include many additional features to avoid particularly expensive processing steps in software [1]. Full offloads, such as RDMA [2] adapters and TCP offload engines [3] go a step further and implement the complete protocol functionality for data transfers on the NIC ASIC, only leaving application-layer and control-path processing on the CPU.

Unfortunately, the lack of flexibility in these "fixed" offloads causes operational challenges and limits efficiency. Fixed ASIC offloads, by definition cannot adapt to new protocols or changing application requirements and do not even allow bug fixes [4], [5]. Further, even with full transport layer offload, such as a TCP offload, application protocols still need to be implemented on the host processor, consuming processor time.

To address this, SmartNICs offer programmability by integrating network processors (NPUs) [6], [7], or FPGAs [8], [9] onto the NIC. Both have found adoption in the public cloud; Amazon deploys ARM-based SmartNICs in their Nitro architecture [10] and Microsoft deploys FPGAs with Catapult [5]. NPUs comprise many smaller processors cores augmented with additional acceleration blocks, e.g. content-accessible memory or crypto [11]. They only perform well for processing that can be effectively parallelized [12], which is often not the case for complex stateful protocols like TCP. FPGAs provide a hardware programming model with much greater flexibility to adapt the architecture to different protocols, e.g. with tailored pipelines for specific protocols. But FPGAs operate at lower clock rates and require larger chip area for the same functionality compared to ASICs [13].

This is reminiscent of ML acceleration before domain-specific architectures (DSAs) [14], such as the TPU [15], gained acceptance. Compared to GPUs and FPGAs, DSAs set new records in performance, energy efficiency, and chip area, for a broad range of ML workloads by offering efficient hardware primitives for tensor operations common in the ML domain. ML has also converged on high-level domain-specific programming models that decouple applications from specific hardware [16]–[18], and capture semantics at the mathematical level without binding to specific implementation choices. This combination has enabled rapid innovation and adoption for architectures, by providing a quick and low-risk path to adoption. For architecture research this also enables robust evaluation, as the same model can directly be evaluated and compared on different architectures, even automatically [19].

We argue this is also a promising path for network protocol processing acceleration. Just as TPUs enable fast and energy-efficient tensor operations for a wide range of ML algorithms, *network accelerators should enable efficient packet processing for a broad class of protocols* with domain specific hardware primitives. To innovate and effectively evaluate we need a *portable programming model for network protocol processing*, that decouples protocols from specific architectures.

In this paper we present Kugelblitz, a framework for designing, implementing, evaluating, and comparing domain specific architectures for network protocol processing. We first define an abstract domain-specific programming model for protocol processing. Next, we design a highly parametrized hardware

model for specifying a broad range of domain-specific protocol processing architectures, and implement an RTL generator for this model. Finally, we propose a compiler that takes a protocol specification and hardware parameters, and generates an implementation of the protocol for the specific architecture.

II. Background and Motivation

A. Related Work: Reconfigurable Switches & P4

Programmable switches, such as RMT [13], Barefoot Tofino [20], and Cavium XPliant [6], demonstrate that flexible packet processing is feasible with manageable overhead, even at the required aggregate bandwidths of terabits in switches. These architectures rely on pipelines of match-and-action stages, where each stage performs configurable table lookups, and based on the results modifies a vector of packet fields traversing the pipeline. Designed for typical switch processing, these architectures employ table lookups, simple parallel processing, and minimal memory for keeping state across packets.

The P4 [21] programming language provides a portable interface for reconfigurable switches. The match-and-action paradigm is also central to P4, where processing is described as a sequence of table lookups and actions triggered based on lookups. Actions in P4 can execute multiple parallel operations without data dependencies. Programmers have to manually split sequential operations across stages. Ongoing development of P4 is lifting restrictions, introducing additional flexibility, and adding support for other network devices, such as NICs.

B. End-Host Processing is Complex

Requirements for end-host protocol processing for modern cloud applications are fundamentally different from switches: complex and inherently stateful. For example, TCP, the dominant protocol in clouds and data centers in general, is notorious for complex processing and state management. We find that even just handling common case TCP data reception requires more than 70 operations, with multiple chains of more than 10 dependent operations. Each packet for a connection also reads and updates the same connection state and only behaves correctly if it operates on the most recent version of the state.

C. Cloud Applications need Flexible Processing

Finally, much recent work has demonstrated that fixed protocol offload is insufficient, as protocols and application requirements evolve. Proposals range from offloading TCP splicing functionality to accelerate proxies [22], extending RDMA with application operations [23], to offloading RPC protocols [24] and even application logic [25], [26]. But currently much of this work is limited by and consequently guided by the drawbacks of FPGA and NPU SmartNICs.

D. Technical Challenges

We identify three open technical challenges for flexible network protocol offload in the cloud and data centers:

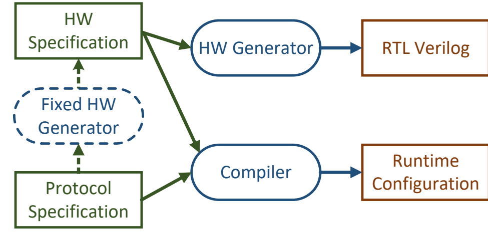

Fig. 1. Kugelblitz system overview.

a) Wide and largely unexplored design space: First, beyond SmartNICs based on NPUs and FPGAs the design space is largely unexplored (at least in published work). We originally set out to design a concrete flexible offload architecture, but quickly realized that even making educated guesses for good design directions choices is a challenge.

b) Manual protocol implementation does not scale: When exploring different designs, re-implementing protocols for different architectures requires significant effort. We found that this is often even the case for small changes, such adding or removing a few ALUs, or changing the depth of a pipeline.

c) Evaluating overall system efficiency: Finally, evaluation for protocol offload architectures is a challenge as we generally need full system results to draw meaningful conclusions. Offloading protocol processing from the CPU to dedicated hardware will reduce host cycles by design. But if the area and power required for the dedicated hardware outweighs the overheads for the additional CPU cores then this is a net loss. Even more so for comparing two offload designs that require different processing on the host CPU.

III. Kugelblitz Design

We seek to address these challenges with Kugelblitz in four core components as shown in Figure 1. The *abstract protocol specifications* are Kugelblitz's programming model for architecture-agnostic implementation of network offloads. Our *hardware configurations* are the programming interface for specifying concrete hardware architecture design points in Kugelblitz. The *hardware generator* generates full RTL implementations from hardware configurations provided as input. Finally, the *protocol compiler* creates a runtime configuration implementing the specified protocol offload on the specified hardware configuration.

With this architecture, Kugelblitz aims to enable efficient and systematic exploration of the design space for flexible offload. We can quickly generate full RTL implementations for different architecture configurations without laborious manual implementation. Next, by decoupling protocol implementation from architecture, Kugelblitz enables rapid and systematic ”apples-to-apples” comparisons of different architectures for a reference corpus of protocols implemented once.

The Kugelblitz hardware architecture integrates into existing NICs as a ”bump in the wire”, interposing between the Ethernet MAC and the regular NIC logic. Kugelblitz can inspect, modify, and drop packets from the network before

passing them to the NIC, and vice versa for outgoing packets. Microsoft Azure uses a similar architecture with their Catapult FPGAs for implementing network virtualization and management [5]. We leave DMA integration, which would enable even broader offload functionality, as future work.

A. Protocol Specifications

Hardware-agnostic protocol specifications are essential for effectively exploring the hardware design space. Our key design goal for this programming interface is to capture all necessary protocol detail for the compiler without binding to specific implementation choices. For example, while the LLVM [27] intermediate representation is portable across many processor architectures, it is inherently processor-centric and captures implementation choices, e.g. by requiring sequential ordering of instructions. We instead aim for abstractions that allow the compiler to understand the specification at a semantic level and leave maximal flexibility for implementation on a broad range of different architectures.

To this end, we use a data-flow graph (DFG) abstraction for protocol processing. DFGs only capture data dependencies without enforcing additional ordering constraints, and as a result explicitly expose parallelism. We specify protocols with two separate DFGs, once for receive processing and one for transmit. In addition, Kugelblitz protocols includes declarations for elements keeping state across packets and allowing software on the host (control plane) to configure processing.

1) State Declarations: Kugelblitz includes two types of state elements: arrays and lookup tables. A protocol can include multiple state elements or none at all.

Arrays declarations consist of a name, the element size, and the number of elements. Kugelblitz protocols can read from and write to arrays by index, as can the software control plane. Only one read access followed by at most one write access are allowed in each processing direction. For arrays with write accesses, the semantics guarantee atomic operation between read and write with regard to other packets.

Lookup tables translate a key to an index in the table, or an error if the key is not present. The declaration comprises a name, key length, and table size. Protocols can read lookup tables, while writes are only possible from the control plane. Lookup tables are typically combined with arrays of the same size to translate the returned index into a value, but can also be used separately if only the existence of a key is checked.

2) RX & TX Processing Graphs: Each protocol specification consists of two separate data-flow graphs, one for receive and one for transmit. A Kugelblitz DFG specifies the complete processing for an individual packet in the corresponding direction. Our semantics guarantee that packets appear to be processed sequentially and in isolation, compiler and architecture ensure this.

DFGs consist of nodes connected by directed edges indicating data flow from one node to another. Each node can have multiple inputs, a single output, and attributes specified at declaration time. We represent data as bit vectors and each edge is typed with the number of bits it carries. These types are enforced in the compiler and all conversions are explicit.

	Inputs	**Outputs**	**Attributes**
Conversions			
Constant		value	value, bits
Slice	value	result	offset, bits
Merge	a, b, ...	value	
Zero Extend	value	result	bits
Sign Extend	value	result	bits
Compute			
Unary	operand	result	operation
Binary	left, right	result	operation
Conditional/mux	cond., t-value, f-value	result	
Input / Output			
Read Packet		result	offset, length
Write Packet	value		offset, length
Read Control		result	register
Write Control	value		register
State			
Table Lookup	key	index	table
State Read	index	value	memory
State Write	index, value		memory

TABLE I
KUGELBLITZ PROTOCOL DATA-FLOW GRAPH NODE TYPES.

Table I shows the 15 node types in Kugelblitz. To start with, *read packet* extracts *len* bit from the packet at *offset*. *Read control* provides access to the metadata specified as the node attribute, such as the packet length. Packet and control reads always access the original packet data and metadata even when writes are present. Protocols modify outgoing packets with the complementary *write packet* and *write control*. *Write control* can additionally also indicate that the packet should be dropped, the original version should be forwarded, or that it should be sent out in the other direction (loopback). *State read* and *state write* provide access to the element at *index* of the array specified in the node attribute. Finally, *Table lookup* takes the key as an input and searches the table specified in the attribute, returning the index or a value indicating failure.

Most of the other node types are similar to prior data flow graphs, so we omit a description here for brevity, except for two: *Slice* extracts *len* bits from the input starting at *offset*. *Merge* concatenates the bit vectors from all inputs.

B. Hardware Specifications

The second core abstraction in Kugelblitz are specifications for hardware configurations that serve as inputs for the hardware generator and the compiler.

1) Hardware Model: Our current design exclusively targets spatial pipelines that combine ALUs, registers, and routing elements, that each packet traverses in the same order. While Kugelblitz hardware specifications do support completely static pipelines where no runtime configuration is possible, the main focus is on reconfigurable pipelines, where the operations for each ALU as well as input/output routing are configured

at runtime. Such pipelines have proven successful at handling high packet rates in programmable switches [6], [13], [20].

The Kugelblitz hardware model comprises a separate pipeline for transmit and receive and a central pool for memory elements, and the configuration reflects this structure. The pipeline architecture combines a *parser* for extracting a parallel vector of pipeline inputs per packet from the serial stream of flits arriving from the network or NIC. From there, packets traverse a sequence of parallel *registers*, *ALUs* and *routing elements*. At the end, packets arrive at the *de-parser*, which reassembles the modified packet and prepares it for serialization into flits for transmission. Finally, for *state memory* array and lookup table elements, the pipeline configuration specifies connection points and the separate state configuration contains attributes such as memory capacity.

2) Pipeline Configuration: For each pipeline the configuration specifies the parser and de-parser attributes, and lists the registers, ALUs, and routing elements, and their connections.

In our initial design, we support a basic parser and de-parser model. The *parser* extracts a prefix of configurable length from the packet and routes it to a configurable group of registers at the beginning of the pipeline, along with an additional register for the packet length. Conversely, the *de-parser* takes its inputs, from configurable registers at the end of the pipeline and then serializes the packet. The de-parser inputs are the (potentially) modified packet prefix of a configured maximal length, length of the prefix to use, and offset and length of the rest of the original packet that should be appended to the prefix when serializing the updated packet.

Registers, ALUs and, routing elements are the main source of flexibility in the Kugelblitz hardware model. *Registers* break up the pipeline into individual stages. The pipeline configuration contains a list of registers, with each register's bit width and the routing element where the register obtains its input. *ALUs* perform the main data processing in Kugelblitz, and have up to three inputs from routing elements, and a single output. The hardware configuration for the ALU lists all opcodes that it supports, and the runtime configuration selects the specific opcode to use for each ALU. A *routing element* selects one of multiple inputs to pass along to its output based on its runtime configuration. For each routing element, the configuration specifies a list of all possible inputs, registers, ALUs, or state access units.

Within the pipeline we also configure three types of *state access units*: table lookup, memory read, memory write. The *Table lookup unit* configuration specifies the table to access and a routing element providing the lookup key, and outputs the index. The *Memory read unit* configuration specifies the memory instance and a routing element for the address, and outputs the value. Conversely, the *memory write unit* configuration also includes a routing element for the value, but does not generate an output.

The pipeline configuration is highly flexible, e.g. through routing elements ALUs can obtain their inputs from registers, state access units, or even other ALUs. ALUs can also span multiple pipeline stages if inputs come from a different stage,

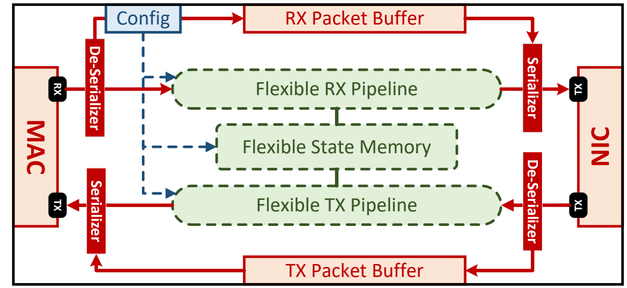

Fig. 2. Kugelblitz hardware architecture.

in which case the instantiated ALU uses internal pipeline registers. Similarly, multiple ALUs can be chained without registers in between, resulting in larger combinatorial components.

3) State Memory Configuration: Finally, the configuration for state memory comprises a list of memory instances. For each *lookup table instance*, the configuration specifies the width of the key and size of the table memory, along with the implementation strategy (currently we only support hardware CAM) Each *memory instance* is configured by a width and size of the memory. The control plane is responsible for initializing and updating state memory instance contents.

C. Hardware Generator

The Kugelblitz hardware generator then takes a specific hardware configuration specification and generates RTL for use in either simulation or synthesis. We implement the hardware generator in Chisel [28]. The generator comprises implementations of individual Kugelblitz hardware components with Chisel primitives, and leverages Scala for metaprogramming to combine components into a full RTL implementation according to the input configuration. We make extensive use of the Chisel parameter system to hierarchically pass relevant subsets of the configuration into subsystems and components. As Kugelblitz hardware configurations reflect hardware structure, the generator first iterates over all elements in the configuration and instantiates them, and then in a second step connects them according to the configuration.

For routing elements and ALUs that need runtime configuration, the hardware generator instantiates a register to hold the configuration. The generator also creates infrastructure to connect configuration registers to a management bus for access from the control plane. State memory instances are also connected to the same configuration bus. We expose this configuration bus as memory mapped registers over PCIe. In addition to the RTL, the hardware generator also produces an address map that lists the addresses for all individual registers and memory instances, as the concrete address layout depends on the specific hardware configuration.

Figure 2 shows how the generated components for pipelines and memory integrate into the complete architecture. Packets enter Kugelblitz through a (typically very wide) AXI stream from either the Ethernet MAC or the NIC. From there, the de-serializer stores them in the packet buffer and passes the

flits along to the parser. When packets exit the de-parser, the serializer reassembles the prefix, based on the control information also arriving with the packet, with the original packet in the packet buffer before streaming out the flits via the outgoing AXI stream. We expose the configuration via an AXI-lite interface externally connected to PCIe via vendor IP.

D. Protocol Compiler

Our compiler takes a protocol specification and a hardware configuration as an input, and generates the necessary runtime configuration for the specific hardware to implement the protocol. The compiler first internally builds a graph representation of the hardware pipeline represented by the configuration. Now the compilation is essentially a graph embedding/search problem of finding a feasible assignment of operations to ALUs, state elements to memory instances, along with a configuration of routing elements to respect the data flow. The I/O nodes are constrained to be assigned to fixed registers at the beginning and end of the pipeline. This mapping is not always feasible depending on hardware and protocol specification. Further, the combinatorial search space for mappings is typically large depending on the configuration.

Our work-in-progress prototype uses an SAT solver to search for feasible mappings. Support for memories is currently still incomplete.

E. Fixed Hardware Configuration Generator

We also implement a converter to generate *fixed hardware configurations* from a protocol specification. These fixed configurations are primarily intended to serve as baselines to evaluate the cost of flexibility compared to fixed architectures.

This converter takes the data flow graph and builds up a hardware configuration that matches the structure of the data flow graph. It instantiates an ALU for each compute node only supporting the corresponding operation. For each edge, we instantiate a register, and set up routing elements with only one option for routing. Finally, we create memory instances for each state declaration and also connect access units in the pipeline. The resulting hardware configuration only requires runtime-configuration for initializing state memories, all other configuration registers are omitted when generating RTL.

IV. Preliminary Evaluation

We now describe our evaluation methodology for Kugelblitz, and give a few initial results to demonstrate feasibility.

A. Goals

For a convincing evaluation of Kugelblitz we need to measure three metrics with a range of combinations of protocols and hardware configurations. First, we are primarily interested in end-to-end performance, including overall throughput, latency, and CPU utilization with multiple connected hosts. Next, we need results from hardware synthesis, especially the required chip area and timing analysis confirming how fast the design can run. Finally, we want to evaluate overall system power consumption, including host and generated architecture.

B. Methodology

Achieving these evaluation goals is a challenge. For an ideal evaluation we would tape out each design, and deploy the chip on a board in real servers. Unfortunately, this is not conducive to a broad design exploration. Instead, we rely on a combination of full system simulations, VLSI synthesis and timing as well as power analysis, and FPGA deployment.

For simulation, we have obtained the SimBricks [29] modular simulation framework from the authors, and integrate an RTL simulation of Kugelblitz into SimBricks. SimBricks combines this simulator with other simulators such as gem-5 into a full end-to-end system, including cycle-accurate synchronization to enable meaningful performance measurements. We use the results from the timing analysis of the VLSI synthesis to set the simulation frequency, and then measure overall system performance. Finally, we use this simulation to also generate signal activity files for VSLI power analysis.

To measure realistic power results for the host CPU, we also deploy Kugelblitz configurations on a NIC with a very large Xilinx UltraScale+ FPGA. We also use this setup for testing and to validate simulation results where possible.

C. Early Results

At this point, we have working end-to-end simulation of Kugelblitz and have implemented two protocol offloads: IPv4 network address translation and a TCP firewall. Because of incomplete state memory support in the compiler, we have so far only evaluated fixed hardware configuration end-to-end. In addition, we have synthesized configurations for our FPGA board in combination with the Corundum open source FPGA NIC [30]. We also have initial VLSI synthesis results.

V. Conclusions and Future Work

Kugelblitz enables rapid and principled design exploration for reconfigurable network protocol offload. The hardware generator automatically generates full RTL implementations from high-level hardware specifications. By decoupling protocol implementation from hardware architecture, we can automatically combine multiple protocol specifications with a broad range of architectures. We have also outlined a methodology for evaluating end-to-end performance, power, and hardware cost, without the need for a full tape out. Our early stage results demonstrate the feasibility of our approach.

Naturally, much work remains to be done and there are many open challenges. In the compiler we anticipate scalability challenges when mapping large complex protocols to large hardware configurations, because of the large search space. Besides our SAT-based mapping we plan to explore other search algorithms. We would also like to explore program synthesis techniques for more flexible compilation, where there is no 1-to-1 correspondence between nodes in the protocol and in the hardware graphs. A second open challenge is integrating DMA into Kugelblitz, to overcome the limits of the bump-in-the-wire architecture and support a broader range of offloads.

Acknowledgment

We thank Emiliano Luci for contributing a graphical viewer and editor for Kugelblitz dataflow graphs and configurations, and Jonathan Mace for continued feedback on this work.

References

[1] P. Shinde, A. Kaufmann, T. Roscoe, and S. Kaestle, "We need to talk about NICs," in *14th Workshop on Hot Topics in Operating Systems*, ser. HOTOS, 2013.

[2] RDMA Consortium, "Architectural specifications for RDMA over TCP/IP," http://www.rdmaconsortium.org/.

[3] A. Currid, "TCP offload to the rescue," *ACM Queue*, vol. 2, no. 3, Jun. 2004.

[4] J. C. Mogul, "TCP offload is a dumb idea whose time has come," in *9th Workshop on Hot Topics in Operating Systems*, ser. HOTOS, 2003.

[5] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure accelerated networking: SmartNICs in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI, 2018.

[6] Cavium Corporation, "XPliant Ethernet Switch Product Family," http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.

[7] Netronome, "NFP-6xxx flow processor," https://netronome.com/product/nfp-6xxx/.

[8] "Product: Innova Flex," https://support.mellanox.com/s/productdetails/a2v50000000XzBrAAK/innova-flex.

[9] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *41st Annual International Symposium on Computer Architecture*, ser. ISCA, 2014.

[10] "AWS re:Invent 2018: Powering next-gen EC2 instances: Deep dive into the Nitro system (CMP303-R1)," https://www.youtube.com/watch?v=e8DVmwj3OEs, 2018.

[11] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson, "The next generation of Intel IXP network processors," *Intel Technology journal*, vol. 6, no. 3, pp. 6–18, Aug. 2002.

[12] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading distributed applications onto SmartNICs using IPipe," in *2019 ACM SIGCOMM Conference on Data Communication*, ser. SIGCOMM, 2019.

[13] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *2013 ACM SIGCOMM Conference on Data Communication*, ser. SIGCOMM, 2013.

[14] J. Hennessy and D. Patterson, "A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development," https://www.acm.org/hennessy-patterson-turing-lecture, 2018.

[15] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *44th Annual International Symposium on Computer Architecture*, ser. ISCA, 2017.

[16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI, 2016.

[17] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI, 2018.

[18] "Open Neural Network Exchange Format: The new open ecosystem for interchangeable AI models," Retrieved May 2020 from https://onnx.ai/, 2020.

[19] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, "A hardware–software blueprint for flexible deep learning specialization," *IEEE Micro*, vol. 39, no. 5, pp. 8–16, Sep. 2019.

[20] Barefoot Networks, "Barefoot Tofino," https://barefootnetworks.com/products/product-brief-tofino/.

[21] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[22] Y. Moon, S. Lee, M. A. Jamshed, and K. Park, "AccelTCP: Accelerating network applications with stateful TCP offloading," in *17th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI, 2020.

[23] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso, "StRoM: Smart remote memory," in *15th ACM European Conference on Computer Systems*, ser. EuroSys, 2020.

[24] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou, "Dagger: Efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics," in *26nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2021.

[25] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, "High performance packet processing with FlexNIC," in *21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2016.

[26] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "KV-Direct: High-performance in-memory key-value store with programmable NIC," in *26th ACM Symposium on Operating Systems Principles*, ser. SOSP, 2017.

[27] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization*, ser. CGO, 2004.

[28] 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 *49th Annual Design Automation Conference*, ser. DAC, 2012.

[29] H. Li, J. Li, K. Jang, and A. Kaufmann, "Reproducible host networking evaluation with end-to-end simulation," arXiv preprint arXix:2012.14219, 2020.

[30] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen, "Corundum: An open-source 100-Gbps NIC," in *28th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM, 2020.