NetCL: A Unified Programming Framework for In-Network Computing

George Karlos
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
g.karlos@vu.nl

Henri Bal
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
h.e.bal@vu.nl

Lin Wang
Paderborn University
Paderborn, Germany
lin.wang@upb.de

***Abstract*—The emergence of programmable data planes (PDPs) has paved the way for in-network computing (INC), a paradigm wherein networking devices actively participate in distributed computations. However, PDPs are still a niche technology, mostly available to network operators, and rely on packet-processing DSLs like P4. This necessitates great networking expertise from INC programmers to articulate computational tasks in networking terms and reason about their code. To lift this barrier to INC we propose a unified compute interface for the data plane. We introduce C/C++ extensions that allow INC to be expressed as kernel functions processing in-flight messages, and APIs for establishing INC-aware communication. We develop a compiler that translates kernels into P4, and thin runtimes that handle the required network plumbing, shielding INC programmers from low-level networking details. We evaluate our system using common INC applications from the literature.**

***Index Terms*—Distributed computing, In-network computing, Programmable data plane, Programming languages, Compilers**

I. Introduction

The past decade has witnessed a surge of programmable data plane (PDP) networking devices [1]–[4]. PDP devices allow for custom packet processing beyond traditional protocols, a paradigm shift that has facilitated great networking innovation [5]–[11]. Leveraging their high performance and convenient on-path placement, researchers have investigated the offloading of computational tasks inside the network during data movement—a new paradigm known as in-network computing (INC) [12]. INC has been shown to improve throughput, latency, and energy efficiency, in applications like aggregation [4], [13]–[15], caching [16], [17], coordination [18]–[22], query processing [23], and ML inference [24], [25], among others. As device capabilities continue to advance, this list is likely to grow.

Despite its numerous successes, INC remains a niche enterprise, only available to networking experts [26], largely due to its challenging programmability. PDP devices employ high-performance packet processing chipsets programmed in domain-specific languages like P4 [27] and NPL [28]. These languages are based on the match-action abstraction [29] and expose low-level constructs tailored to network function development. Consequently, imperative code becomes difficult to express directly. Attempting to "hack" compute logic into such low-level networking terms adds considerable cognitive strain and leads to verbose and brittle code that is hard to reason about. PDP code is also non-portable [30], with each target exposing its own abstractions and APIs, even for simple functionality. Finally, the absence of OS-like abstractions and virtualization mechanisms [30], [31] requires INC programmers to define the complete networking behavior of the device. This involves forwarding decisions that depend on the physical network and would normally be the operator's responsibility.

Recent efforts on higher-level PDP abstractions [32]–[35] fall short for INC as they fundamentally focus on packet processing and protocol handling. Studies specifically addressing INC [36], [37] mostly follow a bottom-up approach, building on primitives tailored towards existing applications, and do not solve the two-language problem. The situation is analogous to the pre-CUDA [38] era of GPGPU programming with pixel shaders [39]. We believe that if PDP devices are to serve as compute accelerators, they should have a dedicated compute API [40].

In the spirit of compute acceleration APIs like CUDA [38] and OpenCL [41], we propose NetCL, a unified programming framework for INC, based on extending C/C++. NetCL features a compute-centric model wherein INC is expressed as kernel functions processing in-flight messages on PDP devices. NetCL intuitively couples in-network execution with message passing and offers a declarative API for application-specific forwarding. Low-level networking details facilitating communication are handled by NetCL runtime mechanisms, allowing programmers to focus on application logic.

In the rest of the paper, we first provide background on PDP and INC and motivate our idea (§II). Then, we present NetCL's design and workflow (§III), system and in-network execution model (§IV), and programming model and API (§V). We then discuss the implementation of the NetCL compiler targeting Intel Tofino switches, and the NetCL runtime (§VI). We evaluate our system on representative applications from the INC literature (§VII), and conclude with a discussion (§VIII), review of related literature (§IX), and final remarks (§X).

II. Background and Motivation

P4 is the leading PDP language, enjoying multi-vendor adoption [3], [42]–[46]. P4 PDPs are collections of programmable forwarding elements, that, together with fixed function ones, are organized as a pipeline, as shown in Figure 1. Processing starts with header parsing, followed by user-defined *ingress* control logic that uses parsed headers to make an *egress* port selection. Some architectures (e.g., Tofino)

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 © 2024 IEEE

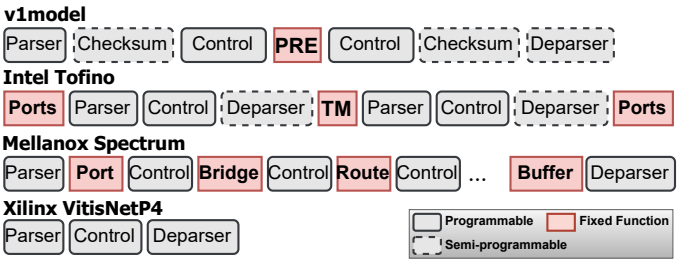

Fig. 1: Block arrangements of various P4-programmable dataplane architectures. Packet path from left to right.

offer an additional egress processing stage. Before exiting the pipeline, parsed headers are serialized by the deparser control.

Parsers are programmed as finite-state machines. Control blocks mix imperative code with match-action tables (MATs) and target-specific externs. MATs execute code based on matching conditions over headers and metadata. They compactly express complex if-else chains and usually translate to hardware-based lookup operations. MATs are runtime reconfigurable only from the control plane [47].

Central to P4 is the concept of a P4 architecture [48]. This is vendor-provided code defining each programmable block's interface and the layout of the pipeline. For instance, Intel's Tofino Native Architecture (TNA) [49] offers six programmable blocks. Lines 30-43 of Figure 2 are their user-defined implementations, instantiated in lines 44-45. Functionality beyond MATs is exposed as black-box externs, which are also part of the P4 architecture. The objects `Register`, `RegisterAction` and `Hash` at lines 5, 6, and 8-10, are TNA externs for stateful memory and hashing.

P4-programmable chips range from ASICs [3], [50], to NPUs [45], [46], to FPGAs [44]. Intel's Tofino ASIC is based on the RMT [51] architecture. Parsed headers and metadata are placed on a bus called the Packet Header Vector (PHV) and carried through each block sequentially. Control blocks consist of 12-20 hardware match-action stages. At each stage, multiple parallel lookup operations over the PHV (match) determine the inputs to a VLIW instruction (action). Each stage has its own set of resources like SRAM, TCAM, and hash engines. Tofino's traffic manager (TM in Figure 1) multiplexes the ingress and egress of 2-4 physical, share-nothing pipelines. The fixed pipeline structure and per-stage strict timing constraints enable processing at line rate, as well as bounded, predictable per-packet latency.

We illustrate the shortcomings of using P4 for INC with a common use case: an in-network cache between a key-value store (KVS) server and clients. Such a cache can serve billions of queries per second at microsecond latency [16]. A simplified version (handling only `GET` queries) is shown in Figure 2.

Programming abstractions. In the simplest case, looking up a key in the cache involves a MAT with an exact match on the key's header field, that, on a hit, writes to the value's (placeholder) header field (lines 22-26). Linear stateful memory is exposed through special objects that require the programmer to define the access operation (lines 5-7). The lack of array-like declarations and looping constructs increase

```
1 header cache_t {bit<8> Op;bit<32> K;bit<32> V;bit<1> Hit;bit<1> Hot;}
2 struct headers_t {eth_t ETH; ipv4_t IP; udp_t UDP;... cache_t Cache;}
3
4 control CountMinSketch(inout headers_t H) {
5   Register<bit<32>, bit<16>>(65536) Cnt0;
6   RegisterAction<bit<32>,...>(Cnt0) Incr0 = {
7     void apply(inout bit<32> m, out bit<32> o) {m = m |+| 1; o = m;}}
8   Hash<bit<16>>(HashAlgorithm_t.CRC16) Hash0;
9   Hash<bit<16>>(HashAlgorithm_t.CRC32) Hash1;
10  Hash<bit<16>>(HashAlgorithm_t.XOR16) Hash2;
11  apply {
12    bit<32> c0 = Incr0.execute(Hash0.apply({H.Cache.K}))
13    bit<32> c1 = Incr1.execute(Hash1.apply({H.Cache.K});
14    bit<32> c2 = Incr2.execute(Hash2.apply({H.Cache.K}));
15    if (c1 < c0) c0 = c1;
16    if (c2 < c0) c0 = c2;
17    H.Cache.Hot = c0 > THRESH) ? 1 : 0; }}
18
19 control Cache(inout headers_t H) {
20   CountMinSketch() CMS;
21   action CacheHit(bit<32> v) { H.Cache.V = v; H.Cache.Hit = 1; }
22   table cache { key = {H.Cache.K : exact}
23                 actions = {NoAction; CacheHit;}
24                 default_action = NoAction();
25                 entries = { 1: CacheHit(42); 3: CacheHit(42);
26                             2: CacheHit(42); 4: CacheHit(42);}}
27   apply { if (cache.apply().miss) CMS.apply(); }}
28
29 // main ingress control
30 control In(inout headers_t H, ...) {
31   apply {
32     if (H.Cache.isValid()) {
33       if (H.Cache.Op == GET_REQ) {
34         Cache.apply(H);
35         if (H.Cache.Hit) send_back(); else send_to_server();
36     }}
37     if (H.ETH.isValid()) { ... }
38     if (H.IP.isValid())) { ... }
39     // L2/L3 forwarding, ACL, Firewall, LB ...
40 }}}
41 control Eg(...) { ... }
42 parser InParser(...) { ... }  control InDeparser(...) { ... }
43 parser EgParser(...) { ... }  control EgDeparser(...) { ... }
44 Pipeline(InParser(),In(),InDeparser(),EgParser(),Eg(),EgDeparser()) P;
45 Switch(P) main;
```

Must repeat definitions for every hash function used.

INC requires interfacing with forwarding code.

How to program the remaining forwarding behaviour?

Fig. 2: Simplified in-network read-only cache in P4 for Tofino [3]. Networking, target-specific, and non-standard P4 code highlighted.

verbosity and duplication [52], as code needs to be manually unrolled (lines 4-16). This often leads to extensive use of the preprocessor for code generation [53] that can quickly lead to errors [54]. Compositional effects of the aforementioned can quickly lead to obscure code that is hard to reason about.

Multi-language programming. Using a distinct device language can introduce subtle bugs, as the programmer must associate types and operation semantics with those of the host language. Additionally, modifying MATs, such as for cache eviction, is done via the control plane using RPC-based APIs like the P4Runtime [47]. In the end, the programmer juggles at least three different APIs even for simple functionality.

Target-specific programming. P4 code is tightly coupled to the underlying architectures [30]. In Figure 2 one can see a lot of TNA-specific code. In particular, the instantiated control blocks and their parameters must respect TNA's declarations. The stateful memory and hashing abstractions of lines 5-10 are also TNA-specific. Moving to another target requires rewriting most of the program. Even worse, different targets may support different P4 subsets [55], impose different constraints on the subset they do support [56], and even require non-standard P4 constructs [57]. Line 7 is an example of the latter.

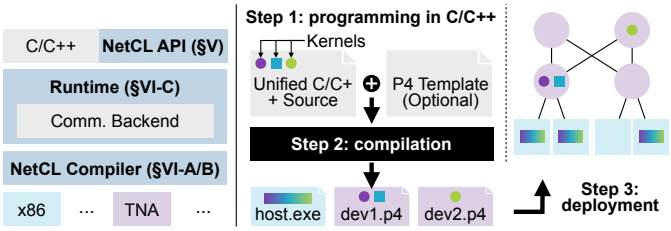

Fig. 3: An overview of the NetCL system: (left) the NetCL stack, (right) the NetCL workflow.

Networking concerns. INC code involves forwarding. For instance, we want to reflect cache hits back to the sender and only forward misses to the KVS server (line 35). Given the monolithic structure of P4 programs, the programmer must either author the entire program or incorporate the cache control directly into an existing program. The former requires filling the gaps in lines 37-43 with, usually, hundreds of lines of low-level networking code. The latter involves new header definitions and parser states, table extensions, and revised control logic. While it is manageable for small programs, real-world P4 programs span thousands of LoC [58] with non-trivial interactions. Embedding INC in those is an enormous task [30]. Moreover, unless the program is explicitly designed with modularity in mind, knowledge of the physical topology is also required. For instance, to implement `send_to_server` we need to know the switch port the server is attached to.

III. System Design and Overview

The primary objective of NetCL is a developer-friendly API for INC. This entails several requirements. On the one hand, (**R1**) it should look like one is programming a computing element rather than a packet processor. On the other hand, PDP devices do process packets inside the network. Thus, **(R2)** computation is triggered by communication—a fact that cannot be completely obscured. We also want (**R3**) some form of control over forwarding, which is crucial to INC applications, and (**R4**) expose PDP features that enhance performance, while (**R5**) staying target-independent as much as possible. Moreover, we want (**R6**) remote access to device memory without vendor-specific control plane APIs. Finally, we want to **(R7)** minimize dependence on physical network details, knowledge of which lies in operators. Figure 3 depicts the main components of our proposed system.

C/C++ API for INC programming. We extend C/C++ with abstractions that allow embedding in-network computations into application code in the form of *computational kernels* (**R1**). Kernels are offloaded to PDP devices to compute *on* messages. Figure 4 shows how NetCL simplifies programming the in-network cache, reducing the LoC and cognitive burden of the programmer compared to Figure 2. The kernel definition focuses purely on the cache logic, and the count-min-sketch is implemented without worrying about target-specific objects. Efficient MAT-based lookups are exposed as function calls (**R4**), and the required attention to networking is limited to a `reflect()` call which returns the message to its sender

```
1  _managed_ unsigned cms[CMS_HASHES][65536];
2
3  _net_ void sketch(unsigned k, unsigned &hot) {
4    unsigned c[CMS_HASHES];
5    c[0] = ncl::atomic_sadd_new(&cms[0][ncl::xor16(k)], 1);
6    c[1] = ncl::atomic_sadd_new(&cms[1][ncl::crc32<16>(k)], 1);
7    c[2] = ncl::atomic_sadd_new(&cms[2][ncl::crc16(k)], 1);
8    for (auto i = 1; i < CMS_HASHES; ++i)
9      if (c[i] < c[0]) c[0] = c[i];
10   hot = c[0] > THRESH ? c[0] : 0;
11 }
12
13 _net_ _lookup_ ncl::kv<unsigned, unsigned> cache[] = {{1,42}, {2,42},
14                                                       {3,42}, {4,42}};
15
16 _kernel(1) _at(1) void query(char op, unsigned k, unsigned &v,
17                              char &hit, unsigned &hot) {
18   if (op == GET_REQ) {
19     hit = ncl::lookup(cache, k, v)
20     return hit ? ncl::reflect() : sketch(k, hot);
21   }
22 }
```

Fig. 4: Complete NetCL device code implementing Figure 2.

(**R3**). To trigger this computation, client code (part of the same program), performs a special `send()` operation to send a message to the KVS server, that, on its way, passes through, and computes on, the device housing the cache (**R2**), by only specifying its ID. Thus, the programmer only declares intent, leaving deployment concerns to the network operator (**R7**).

Runtime. NetCL's host runtime handles communication and exposes a simple API for sending and receiving NetCL messages. We have implemented a UDP communication backend over POSIX sockets, but others (e.g., DPDK [59]) are certainly possible. The host runtime also facilitates control of (remote) device memory, exposed as simple read/write operations using their respective identifiers (**R6**). The device runtime implements NetCL forwarding (e.g., the `reflect()` call in Figure 4), and other NetCL APIs, on a per-target basis.

NetCL compiler. Our compiler translates NetCL kernels into P4 code and is based on LLVM [60]. Host and device code is translated into a common IR form, thereby homogenizing the meaning of types and operations. Moreover, LLVM's target-independent IR eases extending NetCL to more targets. We develop backends for Intel's Tofino Native Architecture (TNA) [49] and P4lang's v1model [61] switches. We selected those two as representatives of opposite extremes because the former is a highly-constrained (but highly-performant) 12-stage switching ASIC, and the latter a software emulator that will execute *any* valid P4 program.

NetCL workflow. First, the user writes a C/C++ application consisting of host and device code. To do so, the programmer first assumes a topology, e.g., a single switch with the KVS server and clients connected to it. Then, the programmer defines kernels for each device on that topology, using the NetCL extensions for kernel placement (the `_at` specifier in Figure 4). Next, the program is compiled once for each device. The user must supply a NetCL-aware P4 base program. This could be the actual device program or simply a template that the network operator will use later on to merge into the actual device program [30], [31], [36]. In either case, it needs to be properly annotated so that code can be emitted into it. Host code is also compiled following a standard

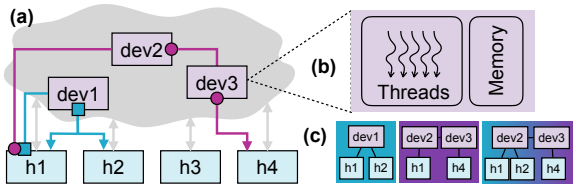

Fig. 5: (a) System model. (b) Device model. (c) Assumed topologies for different computations.

C/C++ workflow. Finally, the generated programs are placed onto physical network nodes. That is, the assumed (abstract) topology gets mapped to the real network, via a deployment system managed by the network operator.

IV. System Model and In-Network Execution

A NetCL system consists of *hosts*, *devices*, and *computations*. Hosts and devices communicate through message-passing, with implicit all-to-all connectivity. Hosts may only send messages to other hosts. Devices may send messages to anyone. Both hosts and devices may receive messages from anyone. Computations occur on devices and are triggered *only* by incoming messages specifically requesting them.

NetCL augments the $send_{i \to j}(m)$ communication primitive with *send-through* and *send-compute* semantics. Message sending becomes $send_{i \to j}(c, d, m)$, which reads "*send message* ***m*** *from host* ***i*** *to host* ***j*** *through device* ***d*** *and perform computation* ***c***". The execution of c at d may alter a message and/or alter its path (e.g., send it to another device), but it cannot alter its source, or request a different computation from a subsequent device.

Figure 5(a) shows a NetCL system with four hosts, three devices, and two computations: square (■) and circle (●). Host `h1` initiates the computations with $send_{1 \to 2}(■, 1, m')$ and $send_{1 \to 4}(●, 2, m)$, respectively. The ■ message triggers a local computation at `dev1`, which results in a multicast to hosts `h1` and `h2`. The ● message first triggers a *local* computation at `dev2`, which results in the message being forwarded to `dev3`. At `dev3`, another local computation occurs, at the end of which the message is forwarded to its original destination, host `h4`.

An in-network computation may reside on multiple devices, with potentially different behaviors at each [20]. Nodes participating in single-device computations like ■ always form a star topology as the leftmost one in Figure 5(c). Multi-device computations require the programmer to implicitly define an abstract topology and steer messages *as if* the network looked that way. The middle topology of Figure 5(c) is a possible one for ●. If ■ and ● are part of the same application, then the more compact topology in the rightmost part of Figure 5(c) is also possible. The abstract topology is an important tool when reasoning about the program during development—e.g., what is the meaning of `reflect()` at `dev3`? It captures the INC traffic patterns of an application and can be later used to drive deployment on the real topology.

A fundamental rule of NetCL is that *no implicit computation* is possible. Every local computation must be explicitly requested by some participating node. For instance, consider

Specifiers	Applies to	Meaning
`_kernel(u8 c)`	$\mathcal{F}$	Declare a kernel, part of computation `c`
`_spec(u8 x)`	$\mathcal{A}$	Declare the specification of a pointer argument
`_at(u16 l...)`	$\mathcal{F}$,$\mathcal{M}$	Place an entity at `l`
`_net_`	$\mathcal{F}$,$\mathcal{M}$	Declare device function or memory
`_managed_`	$\mathcal{M}$	Declare managed memory
`_lookup_`	$\mathcal{M}$	Declare lookup memory
Builtins	Type	
`msg`	`struct { u16 src; u16 dst; u16 from; u16 to; ... }`	
`device`	`struct { u16 id; u8 kind; }`	
Lookup Types (only allowed as `_lookup_` arrays)		Match when
`struct kv<K,V> { K k; V v; }`		`x == k`
`struct rv<R,V> { struct { R lo; R hi;} r; V v; }`		`lo <= x <= hi`
Device Lib	Desc	Example
Actions	Declarative forwarding	See Table II
Atomics	Memory operations	`ncl::atomic_add()`, `ncl::atomic_inc()`
Lookup	Read `_lookup_` memory	`ncl::lookup()`
Math/Binary	Special ops	`ncl::sadd()`,`ncl::bit_chk()`,`ncl::rand<u8>()`
Intrinsics	Target externs	`ncl::tna::crc64()`, `ncl::v1::csum16r()`
Host Lib	Desc	Example
Managed	Remote managed memory	`ncl::managed_read()`, `ncl::managed_write()`
Message	NetCL messages	`ncl::message()`, `ncl::pack()`, `ncl::unpack()`

TABLE I: Overview of the main NetCL C/C++ extensions and APIs. We use shorthands $\mathcal{F}$,$\mathcal{A}$,$\mathcal{M}$ for Function, Argument and Memory, `uW` for `W`-bit unsigned integers and `T<P...>` for `template<typename P...> T`.

$send_{1 \to 4}(●, 3, m)$. Assuming the middle topology of Figure 5(c), m goes through `dev2` before it reaches `dev3`. However, it is only `dev3` that executes ●. The message at `dev2` is a no-op. From the perspective of `dev3`, the previous hop of this message is `h1`. In general, the previous hop of a message is either its source (always a host) or the last device that computed on it.

The device model is straightforward; see Figure 5(b). A device consists of logical threads of execution. When a message is received it is assigned to a thread that processes it uninterrupted. Threads have private memory and may access statically allocated global memory through atomic transactions. There is no synchronization like locks or barriers. Targets supporting additional features may expose them as intrinsics.

V. Programming Model

A NetCL program consists of *host code* and *device code*. The former executes on hosts and the latter is offloaded to PDP devices. Hosts, devices, and computations are identified by unique numerical IDs. NetCL exposes a small set of language extensions, builtins, and APIs, summarized in Table I.

A. Network Kernels and Computations

The `_kernel(c)` specifier declares a kernel function, part of computation `c`. A kernel function describes how a device thread processes a NetCL message. Message data is accessed through the kernel's arguments, which may be values, pointers, or references of fundamental types [62], except `void`. Figure 4 (on line 16) shows a kernel definition for computation `1` that processes `GET` requests. Argument `k` is a lookup key, passed by value since it is only read. Modifications to a by-value scalar argument are device-local. That is, all devices, or hosts, receiving the message, will always see its original value. In contrast, `v` is passed by reference, as it will store the lookup result, and updates are visible to all receivers.

```
1 struct sockaddr_in server;
2 char in[2048], out[2048], hit = 0, op = GET_REQ;
3 unsigned key = 2, val = 0;
4 ncl::message m(1,2,1,1);
5 ncl::pack(out, m, {&op, &key, nullptr, nullptr, nullptr});
6 sendto(socfd, out, m.size, 0, &server, sizeof(server));
7 recvfrom(socfd, in, m.size, 0, ...);
8 ncl::unpack(in, m, {&op, &key, &val, &hit, nullptr});
```

Fig. 6: Host code querying a KVS with in-network caching. Computation 1 (Figure 4) at device 1 explicitly requested on line 4.

Specifications. Kernel arguments are associated with a specification, denoting the number of elements they occupy. Specifications are inferred from the argument type. The attribute `_spec` can be used to explicitly specify pointer arguments, otherwise `1` is assumed. Scalars always have a specification of `1`. The collection of specifications of a kernel's arguments together with their types, forms the specification of the kernel. Below are some examples:

```
1 _kernel(1) void a(int x[3]);                // [3][int]
2 _kernel(2) void b(int x[4]);                // [4][int]
3 _kernel(3) void c(int _spec(4) *x);         // [4][int]
4 _kernel(4) void d(int x, int y[2], int *z); // [1,2,1][int,int,int]
```

Kernels `a` and `b` have different specifications, meaning that array-to-pointer decay [63] does not apply to kernel declarations. Kernel specifications define the layout of the messages the kernel computes on and are used by the runtime to construct such messages. Therefore, kernels of the same computation are required to have matching specifications. Kernel `b` and `c` above could belong to the same computation but `a` and `d` could not.

Net functions. Device code may be further grouped into net functions, declared using the `_net_` specifier, and callable only from device code. Net functions are not associated with any specific computation and may be freely used by any kernel. Argument passing for net functions follows standard C/C++ rules, and thus, the `_spec` attribute has no meaning and is ignored when present. The kernel of Figure 4 invokes the `sketch` net function (on line 18) to update a count-min sketch.

Invoking computations. Kernels are not invoked directly. Instead, a computation starts by sending out a message, as discussed in §IV, that triggers kernel executions on its path. Figure 6 shows an example invocation of the computation of Figure 4, initiated by a KVS client (host `1`). Computation `1` at device `1` is requested explicitly, on line 4. Message data is then packed into a buffer and sent out over a standard socket. On lines 7-8, the reverse occurs to unpack the KVS response. The `pack` and `unpack` procedures are aware of kernel specifications (supplied by the compiler). To avoid unnecessary copying the programmer may supply `NULL` to ignore an argument during packing/unpacking. For instance, `val` and `hit` are only read from a response and can be ignored during packing. `hot` is only relevant to the server and can be ignored in both cases.

Actions. Kernels must exit with an action, i.e., a next-step decision about the message, similar in fashion to XDP [64]. Table II summarizes the available actions. Actions may only appear in return statements, and any path in the kernel's code that does not explicitly return an action has the implicit action (`pass()` by default) returned. In Figure 4 the kernel exits on line 20 with a `reflect()` and on line 22 with an implicit `pass()`. The `multicast()` action requires an explicit *multicast group ID* to be supplied. While a multicast group may only consist of adjacent nodes, `send_to_host()`, `send_to_device()` `reflect_long()`, and (implicitly) `pass()` and `reflect()`, may target distant nodes. The *no-implicit-computation* rule (§IV) guarantees that no intermediate device(s) will compute on the message as a result of those actions. The message will be treated as a no-op until it reaches its target node.

Action	Description
`ncl::drop()`	The message exits the network immediately
`ncl::send_to_host(u16 h)`	Send the message to host with id `h`
`ncl::send_to_device(u16 d)`	Send the message to device with id `d`
`ncl::multicast(u16 gid)`	Multicast the message to a neighbor group
`ncl::repeat()`	Execute the kernel again
`ncl::reflect()`	Send the message back to the previous node
`ncl::reflect_long()`	Send the message back to its source host
`ncl::pass()`	Let the message continue to its destination

TABLE II: NetCL's action API for declarative forwarding.

B. Memory

Kernels access *local* and *global* device memory. Local memory is thread-private. It consists of local variables and arguments and has the lifetime of a single kernel execution. The value of default-initialized local variables is undefined. If needed the user must explicitly initialize them, however, depending on the target, this could be costly.

Global memory consists of global (or static local) variables marked as `_net_` or `_managed_`. It is shared by all threads, zero-initialized, and has the lifetime of the program. Global memory is accessed atomically, without ordering guarantees, using normal indexing and assignment operations. In addition, NetCL offers a rich set of read-modify-write (RMW) atomics, including Fetch-and-Arithmetic/Logical/Bitwise and Compare-and-Swap operations. Most atomics have a conditional version and one that returns the value after the operation, instead of the old one. For instance, the `atomic_sadd_new` calls in Figure 4 atomically perform saturated addition, write the result back to memory, and then return it.

While `_net_` memory is writable only by device code, `_managed_` memory can also be written by host code. In Figure 4, one might want a runtime configurable count-min-sketch threshold. Adapting the kernel for "thresh-update" messages is a viable solution but increases kernel complexity, and can incur additional communication when reliability is required. It can also increase resource consumption and critical path length. Instead, a `_managed_` variable can be used:

```
1 _managed_ unsigned thresh;
2 // from host code...
3 ncl::device_connection c;
4 ncl::managed_write(c, &thresh, 512);
```

Managed memory allows for reliable (remote) accesses using the device's control-plane mechanisms under the hood. It is the appropriate mechanism for slow-path operations like kernel configurations, resets, checkpointing, and so on.

Global arrays may be further specified as `_lookup_`. While normal arrays are indexed, lookup arrays are *searched*. Lookup

arrays are NetCL's abstraction over match-action tables and can only be accessed through special `lookup` functions. A scalar array marked as `_lookup_` acts as a set; looking up a key is testing for membership:

```
_net_ _lookup_ unsigned a[] = {1,2,3};
lookup(a, 2); // true
lookup(a, 5); // false
```

The special types `kv` and `rv` (Table I) allow lookup arrays to act as hashmaps, with exact and range matches, respectively:

```
_net_ _lookup_ ncl::kv<int,int> a[] = { {1,2}, {2,3} };
_net_ _lookup_ ncl::rv<int,int> b[] = { {{1,10},1}, {{11,20},2} };
int x = 42, y = 42;
lookup(a, 2, x);   // true, x = 3
lookup(b, 21, y);  // false, y = 42
```

While, lookup memory could, in principle, be writable by device code, P4 does not yet allow modifying MATs from the data plane. For this reason, `lookup()` is the only lookup memory operation that is currently available to device code. Host code can insert, remove, or modify `_managed_ _lookup_` memory entries. However, global memory cannot be freed or resized. Its capacity for the lifetime of the program is statically determined by its declaration. For instance, in Figure 4, `cache` has a capacity of four entries.

C. Network Locations and Multi-Device Programming

Kernels, net functions, and global memory are associated with a location set containing the device IDs each entity is *placed* at. By default, all declarations are location-less. That is, their location set is $\emptyset$. Any device we compile for will include *all* location-less entities. Explicit locations may be specified using the `_at(...)` specifier. For instance, the kernel of Figure 4 is explicitly placed at device `1`.

To prevent ambiguity, *no two kernels for the same computation may exist at the same location*. Let KERNELS_c be the set of `_kernel(c)` declarations, C^k the computation ID of a kernel k, and $\text{LOC}(d)$ the location set of declaration d. Placement validity of a kernel declaration k is defined as:

$$\begin{aligned}\text{VALID}(k) \iff \text{LOC}(k) = \emptyset \wedge \text{KERNELS}_{\text{C}^k} = \{k\} \vee \\ \forall_{k' \neq k}\ \text{LOC}(k') \neq \emptyset \ \wedge\ \text{LOC}(k) \cap \text{LOC}(k') = \emptyset\end{aligned} \tag{1}$$

Net functions and memory may only be referenced if they are placed in at least the locations of the code referencing them. Let $D = \text{NETFUNCTIONS} \cup \text{MEMORY}$, $\text{USE}(d)$ be the set of references of d and, for any reference u, let $\text{USER}_u \in \text{NETFUNCTIONS} \cup \text{KERNELS}$ be the function declaration u occurs in. Reference validity w.r.t. location is defined as:

$$\begin{aligned}\forall_d \forall_{u \in \text{USE}(d)} \text{VALID}(u) \iff \text{LOC}(\text{USER}_u) \subseteq \text{LOC}(d) \\ \vee\ \text{LOC}(d) = \emptyset\end{aligned} \tag{2}$$

Violations of those rules result in compilation errors. Some examples are shown below:

```
_net_ _at(1,2) int m[42];
_kernel(1) _at(1,2) a(){ m[0] = 1; }// valid
_kernel(1) b() {}                   // invalid placement because of a
_kernel(2) c() { m[0] = 42; }       // invalid reference. m only at 1,2
```

Multi-location entities have a copy placed on each device. For instance, the kernel `a` above will execute on both device `1` and `2`. If different behaviors are required, either the kernel should be written in an SPMD fashion, by branching on the `device.id` builtin, or two kernels should be used. The latter is more readable and almost always costs fewer resources. Moreover, when it comes to memory, writes are local only:

```
_net_ _managed_ _at(1,2) int m; // 2 m versions at 1 and 2
int a = 0;
ncl::device_connection dev1, dev2;
ncl::managed_write(dev1, &m, 1);
ncl::managed_write(dev2, &m, 2);
ncl::managed_read(dev1, &m, &a); // a = 1
```

NetCL does not offer any consistency guarantees between copies. If required, it must be achieved by other means [65].

D. Restrictions and Target-Specific Concerns

High-performance PDP devices trade programmability for performance, a fact that cannot be completely obscured. NetCL's approach is to be as unrestricted as possible at the language level and have the compiler reject programs on a per-target basis. For instance, NetCL allows arbitrary integer multiplication and division. While software switches [61] or FPGAs [44] may support them, ASICs like Tofino only support those that can be converted to shifts. Other restrictions stem from targeting P4. For instance, to match feed-forward P4 pipelines, device code may not contain `goto` statements or recursion, and only loops that the compiler can fully unroll are allowed. Moreover, P4 does not expose addressable memory. To translate an access correctly, the compiler must be able to infer a base object and a regular offset. For that reason, it rejects pointer arithmetic and pointer casting in device code.

At the language level, NetCL imposes no restrictions on memory accesses. However, targeting Tofino introduces an important one. Tofino stateful memory is a stage-local resource, meaning that once a stage is over, its memory is no longer accessible. This affects NetCL device code in two ways. First, no global memory object may be accessed more than once, unless accesses are mutually exclusive:

```
_net_ int m[42];
_kernel(1) void b(int x) { x = (x > 10) ? m[0] : m[1]; }    // valid
_kernel(2) void a(int x) { x = m[0] + m[1]; }               // invalid
_kernel(3) void c(int x) {                                  // depends
    if (x > 10) { x = m[0]; }
    else {/* ...calculate i... */ x = m[i]; } }
```

Whether the access pattern of kernel `3` above is valid depends on the length of the `i` calculation. If it is too long, and based on code dependencies, it may not be possible to place `m` on a single stage, as its accesses are spread out.

Global memory objects should also be accessed in the same order in all paths. Dependent accesses that violate ordering and cannot be reordered result in the program being rejected:

```
_net_ int m1[42], m2[42];
_kernel(2) void b(int x) {
    if (x > 10) {  x = m1[0] + m2[x]; }
    else {         x = m2[x] + m1[0]; } }      // can be reordered
_kernel(1) void a(int x) {
    if (x > 10) {  x = m1[0]; x = m2[x]; }
    else {         x = m2[0]; x = m1[x]; } }   // cannot be reordered
```

The design of NetCL's atomic API is influenced by Tofino. Tofino's hardware stages perform memory operations on

Stateful ALUs (SALUs). SALUs take input from memory and the PHV, execute a small microprogram, and output to both memory and the PHV. While NetCL atomics like `atomic_cond_add_new` may seem redundant at first, they allow handling the condition and return of results in one stage. Conditionally executing an `atomic_add`, and then performing an addition, to achieve the same result, would increase the critical path length, and thus require more stages. Given that there are 12-20 stages on Tofino chips, this is rarely acceptable. Targets that cannot support such atomics are again free to reject the program with an appropriate error message.

E. Putting It All Together

In Figure 7 we show how NetCL can be used to compactly program in-network AllReduce, an application that typically requires hundreds of lines of P4. The basic idea is straightforward. Workers stream packets that contain a chunk of their values (e.g., 32) to a top-of-rack switch, requesting aggregation at a certain slot. The switch keeps track of the workers seen so far, per slot. After intermediate aggregations packets are dropped. After the last aggregation, the result is broadcast. Adding reliability to this scheme requires two things. First, there are two versions for each slot. One holds the current running aggregation and the other holds the previously completed one. This allows retransmissions of the last result in case of packet loss. Second, workers now request slots in an alternating bit fashion. Correctness relies on the fact that no worker can be ahead of another by more than 1 slot. More details can be found in [13].

Our code uses three arrays. The `Agg` array holds the aggregation slots. `Bitmap` keeps track of the workers seen so far, and is needed to detect retransmissions. `Count` is used to track how many values have been aggregated, per slot. Besides the slot version and values, workers also provide their mask (`1 << worker_id`), which saves the kernel from computing it. Moreover, there are two indices for identifying a slot. `bmp_idx` is used to index `Bitmap` and `agg_idx` is use for `Agg` and `Count`. This is due to the slot duplication mentioned above. While we could again compute `agg_idx` locally by offsetting `bmp_idx` based on `ver`, providing a precomputed value saves up resources.

On lines 8-15, the worker is added to the bitmap of one version of the slot, and removed from the other, effectively preparing the second version for phase transition when the slot completes. Notice that in both branches, `Bitmap` (sub)arrays, are accessed in the same order.

Lines 17-32 handle the aggregation. On line 19, values are added only if the worker is not seen in the requested slot version. This atomic call allows the condition to be checked at SALUs. Moreover, it returns the new value only if the operation was performed; otherwise, it returns the old value. This serves two purposes. First, if it is the last aggregation, the result is written to the message. Second, if it is a retransmission from the worker, it is the old result—the one that was lost—written in the message. When a slot completes, the old counter is one, in which case the message is multicast. Note that, when line 27 returns `1`, the actual count in memory has become `0`. This gives us a nice way to detect retransmissions for completed slots as the count will stay at `0` until it is reset on line 20. Thus when the old count is `0`, the message is sent back to the worker, otherwise dropped.

```
1  _net_ uint16_t Bitmap[2][NUM_SLOTS];
2  _net_ uint32_t Agg[SLOT_SIZE][NUM_SLOTS * 2];
3  _net_  uint8_t Count[NUM_SLOTS * 2];
4
5  _kernel(1) void allreduce( uint8_t ver, uint16_t bmp_idx,
6                             uint16_t agg_idx, uint16_t mask,
7                             uint32_t _spec(SLOT_SIZE) *v) {
8    uint16_t bitmap;
9    if (ver == 0) {
10     bitmap = ncl::atomic_or(&Bitmap[0][bmp_idx], mask);
11     ncl::atomic_and(&Bitmap[1][bmp_idx], ~mask); }
12   else {
13     ncl::atomic_and(&Bitmap[0][bmp_idx], ~mask);
14     bitmap = ncl::atomic_or(&Bitmap[1][bmp_idx], mask);
15   }
16
17   if (bitmap == 0) {                              // slot starts now
18     for (auto i = 0; i < SLOT_SIZE; ++i)
19       Agg[i][agg_idx] = v[i];
20     Count[agg_idx] = NUM_WORKERS - 1;
21   } else {
22     auto seen = bitmap & mask;
23
24     for (auto i = 0; i < SLOT_SIZE; ++i)
25       v[i] = ncl::atomic_cond_add_new(Agg[i][agg_idx], !seen, v[i]);
26
27     auto cnt = ncl::atomic_cond_dec(&Count[agg_idx], !seen);
28     if (cnt == 0)                                 // slot finished earlier
29       return ncl::reflect();
30     if (cnt == 1)                                 // slot finished
31       return ncl::multicast(42);
32   }
33   return ncl::drop();
34 }
```

Fig. 7: In-network AllReduce a la SwitchML [13].

VI. Implementation

We have implemented a NetCL compiler and host/device runtimes—steps 1, 2 in Figure 3—and left application deployment for future work. Our compiler is built on Clang [66] and LLVM [60] using a common offloading design [67]–[69], shown in Figure 8. The LLVM intermediate representation (IR) generated by Clang is processed by two separate pipelines. The device pipeline produces P4 programs. The host pipeline produces an executable that is linked with the host runtime.

A. Frontend

Our frontend extends Clang's semantic analysis with the NetCL extensions, checks for placement/reference validity (§V-C), and ensures device code is not mixed with host code. During host-side compilation, a set of rewrites insert records about device code (e.g., kernel specifications), that are later used by the runtime to carry out tasks like message packing/unpacking (§V-A). It also empties, but does not remove, kernel and net function bodies and resizes all device memory to a single element. This way, those entities can be referenced at runtime by taking their address (e.g., as in `managed_read`), yet they do not occupy unusable space.

B. Backend

In general, there are no guarantees that a given program will *fit* an RMT pipeline like Tofino [56]. Thus, a certain

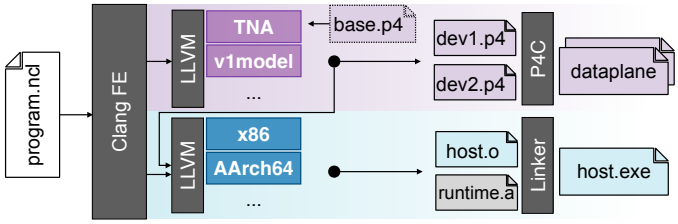

Fig. 8: High-level compiler architecture.

amount of trial and error cannot be avoided even for hand-written P4 code. In addition, any compiler targeting Tofino is faced with two major oddities. First, Tofino's ISA and other low-level architectural information needed for code generation are proprietary. The only alternative is to use P4 as a form of high-level assembly and rely on Intel's P4 compiler to generate binaries. Second, Intel's P4 compiler is also proprietary. In light of the aforementioned challenges, our strategy is to apply heuristics we have gathered from relevant literature, and our own experience in developing P4 code for Tofino. Ideally, we want to reject programs that would confidently not compile to Tofino, even if hand-written, and produce P4 code that does not inhibit the P4 compiler from fitting the program. To that end, we provide several compiler flags to control certain transformations, which the programmer can use to recompile their program and try the P4 compiler again.

Our backend performs over 20 custom passes mixed with an equal number of LLVM passes. To stay within the page limit, we will keep the discussion to a high-level description, and leave the details for an extended version of the paper.

P4-compilable CFG. The first stage is common to all P4 targets. For every kernel, we first inline all `_net_` function calls, unroll loops, and materialize known values (e.g., `device.id`). Then, we run a set of peephole optimization, and instruction simplification and DCE passes [70]. The main goal is for the CFG to become a DAG; otherwise, a relevant error is issued. While per-packet latency primarily depends on the number of stages and enabled components at each, instruction simplification is still beneficial as Tofino ALUs are generally restricted to simple arithmetic operations. The `v1model` target, being a software switch, also benefits from those [71]. Reaching this point guarantees the program can compile for the `v1model` target; Tofino requires extra work.

Tofino specifics. All global memory allocations are translated to objects that will be placed on individual hardware stages by the P4 compiler. Before checking memory accesses for mutual exclusion and ordering violations (§V-D) we attempt two optimizations. First, we apply a coarse-grained version of access-based memory partitioning [72]. Global arrays are split on the outer dimension if all accesses use constants on that dimension. The second optimization is memory duplication. Since Tofino (and P4) does not support data plane updates to MATs, we treat all non-managed lookup memory as constant, and create a copy for each access, removing access dependence on a single stage. While the same could be applied to `_managed_ _lookup_` memory, it requires device support for bulk atomic updates from the control plane. We are currently not aware of whether this is something supported by Tofino and thus, have not implemented duplication for `_managed_ _lookup_` memory. Duplication could lead to excessive resource consumption and thus can be turned off if needed.

We then check for memory violations. First, for any two accesses to the same global memory object, we perform an approximate distance check. We count the minimum number of conditional branches required to reach each access from the entry block. If the difference exceeds a certain threshold we issue an error. This serves as an approximation of an access's relative position in the pipeline. Thus, if two accesses are too far apart, we assume they cannot be placed on the same stage. Second, for any two accesses to different global memory objects, we check that their relative order is the same in all CFG paths. If a path with a reverse order is found we abort compilation and issue an error. This is similar to how Lucid [33] handles access order violations, except that we do not assume the declaration order to be the intended order.

We also perform some instruction reordering. We hoist instructions computing the same value to a common dominator, as long as their operands are available in that block. Moreover, we perform aggressive speculation for instructions that produce values and do not modify memory, hoisting them to the earliest possible block. The combination of these two may reduce critical path length. We found that speculation is what allowed one of the major programs in our evaluation to fit Tofino, by reducing its stage requirements. On the downside, speculation may put pressure on PHV allocation during P4 compilation. Thus, it can also be turned off.

Certain IR patterns, if translated directly to P4, may produce inefficient code. We run a pass that tries to detect those and convert them into intrinsics the code generator understands. For instance, byte swaps generated as bit-slice concatenations [48] can be done in a single stage, and counting leading zeros/ones can be done with a longest-prefix-match (LPM) table. We found that direct translation of some `icmp` predicates [73] with dynamic operands, may produce code that does not compile for Tofino. We transform those into subtractions followed by an MSB check. We also found that sometimes, placing bitcasts on hash engines instead of ALUs allows the P4 compiler to fit a program that otherwise would not. However, since we were not able to derive any rules for when to apply those, we added compiler flags to toggle them.

Optimizations may produce unstructured CFG [74], which cannot be translated to P4 since the latter does not support arbitrary jumps. We solve this with a CFG structurization algorithm [75] based on predicate variables. Finally, we eliminate ϕ-nodes [76] by introducing a fresh variable for each, a store instruction before the terminators of its incoming blocks, and replacing them with load instructions.

Code generation. The code generator translates the LLVM IR to P4. It produces header definitions for kernel arguments, their parsers/deparsers, and a single control block that includes all kernel code for a given location. Largely due to structurization, the codegen loop is simple. For every kernel, we traverse the CFG in reverse postorder (i.e., topological order since

C++

```
if (!(a | b)) ...
```

```
B[i] = atomic_or(&A[i], mask);
```

```
if (lookup(t,k,cacheline)) ...
```

```
v[dyn_idx] = 42;
```

IR

```
%t1 = or i16 %a, %b
%t2 = icmp eq i16 %t1,0
br i1 %t2, ...
```

```
%a.i = getelementptr i16, %A, %i
%a = call i16 @ncl_atomic_or_16(...)
%b.i = getelementptr i16, %B, %i
store i16 %a, i16* %b.i
```

```
%b = call i1 @ncl_lookup(...)
br i1 %b, ..., ...
```

```
%ptr = getelementptr i32, %v, %dyn_idx
store i32 42, i32* %ptr
```

P4

```
bit<16> a;
bit<16> b;
bit<16> tmp;
action or_16_16_16() {
  tmp = a | b; }
...
or_16_16_16();
if (tmp == 0) { ... }
```

```
bit<16> i; bit<16> tmp; bit<16> mask;
Register<bit<16>> A;
Register<bit<16>> B;
RegisterAction<bit<16>,...>(A) ra_or_A = {
  void apply(inout bit<16> m, out bit<16> o)
  { o = m; m = m | mask; } }
action mem_1() { tmp = ra_or_A.execute(i);}
action mem_2() { B.write(i, tmp); }
...
mem_1(); mem_2();
```

```
bool t_hit;
action lu_r_16(bit<16> v) {
    cacheline = v; }
table t {
    key = {k: exact}
    actions = {lu_rd_16;}}
...
t_hit = t.apply().hit;
if (t_hit) { ... }
```

```
box<bit<32>> v[2]; // header stack
action write_0() {v[0].value = 42;}
action write_1() {v[1].value = 42;}
table idx_tbl {
  key = { dyn_idx: exact}
  actions = {write_0;write_0;NoAction}
  entries = {0 : write_0();
             1 : write_1(); } }
...
idx_tbl.apply();
```

Fig. 9: Example code generation.

Fig. 10: Structure of a NetCL-over-UDP packet.

there are no loops) and construct lexical scopes when needed. A kernel's entry block is the top-level scope. Unconditional branch targets are generated in the scope of their predecessor. Conditional branch targets create a new sub-scope, and sinks are generated in the scope of the nearest common dominator of its predecessors. When multiple kernels are present, we generate a top-level switch statement, branching on a message's computation ID (provided by the runtime).

Figure 9 shows some examples of code generations. Most instructions are generated as P4 actions storing the result on a local variable. Global memory is generated as `Register` objects, and accesses as `RegisterAction` objects (Figure 2). Lookup memory is generated as MATs. Local arrays and message memory are generated as header stacks [48]. Unlike `Registers`, header stacks are not dynamically indexed. To dynamically index into header stacks we use index tables, as shown in the rightmost column of Figure 9. An added benefit of this approach is that we get runtime bounds-checking for free.

C. Runtime

The host runtime implements the mechanisms required to interface with a device's control plane to facilitate `_managed_` memory interactions, as well as message packing and unpacking (Table I). For both, it uses the device code records embedded by the compiler in the host code. NetCL messages are crafted based on the chosen NetCL communication backend and the available kernel specifications (§V). Figure 10 shows the layout of a NetCL packet for the UDP communication backend that we have currently implemented.

The device runtime is a small piece of P4 code that handles NetCL headers and controls the execution of NetCL kernels. Together with the generated P4 code it is meant to be embedded in, and invoked by, a base P4 program. For this work, we have adopted a simple annotation-based mechanism for embedding the runtime. For larger programs, we expect code-composition techniques [30], [31] to be more appropriate. However, this is currently out of scope.

Part of the NetCL header is the 4-tuple (`src`, `dst`, `from`, `to`). The `src` and `dst` are host identifiers, and `from` and `to` are device identifiers. When the runtime is invoked, it expects to find those fields in the parsed headers, according to a fixed naming scheme. It first checks if `to` matches `device.id`, and if so, a P4 control block is invoked to execute the kernel (if any) matching the requested computation id, which is also part of the NetCL header. Based on the action selected by the kernel (Table II), or the absence thereof, the 4-tuple is updated accordingly. This 4-tuple is the interface between the NetCL device runtime and the device's base P4 program. The latter is now expected to forward the message. The NetCL header may not be further modified.

We have implemented a simple base P4 program that uses a configurable UDP destination port range to distinguish NetCL messages and, since deployment was out of the scope, assumes that the abstract topology (§IV) is the real topology. When a NetCL message is identified, the incoming NetCL header is stored, and the NetCL runtime is invoked. When it returns, the differences between the old and new NetCL header are used to forward the NetCL message according to §IV. For standard, non-NetCL communication the base program uses basic link-layer forwarding. This simple mechanism was enough to test the performance and correctness of the generated programs in our evaluation.

VII. Evaluation

Following [36], [37], we evaluate our system on three major stateful applications from the INC literature, and, additionally, a small stateless one, shown in Table III. AGG is an implementation of the streaming aggregation protocol of SwitchML [13]. It is almost identical to Figure 7, with the addition of finding a maximum exponent for quantization [13]. Due to current limitations (discussed in §VIII), we only aggregate 32 values per packet.

CACHE is an implementation of NetCache [16]. It extends Figure 4 with `PUT` and `DEL` operations and a validity bit to implement the write-back policy. Similar to [16], it supports 8-byte keys and up to 128-byte values. Accessing a cache line is now a two-step operation where a MAT matches the key to an index that is then used to locate the cache line. We also implement the cache line sharing mechanism of [16], using a

```
_at(LEADER)    _net_ uint32_t Instance;
_at(LEARNERS)  _net_ uint8_t VoteHistory[65536];
_at(ACCEPTORS) _net_ uint16_t VRound[65536];
_at(ACCEPTORS, LEARNERS) _net_ uint16_t Round[65536];
_at(ACCEPTORS, LEARNERS) _net_ uint32_t Value[8][65536];

_at(LEADER) _kernel(1) void leader(uint8_t &type, uint32_t &instance,
    uint16_t round, uint16_t &vround, uint8_t &vote, uint32_t v[8]) {
    ...
}
_at(LEARNERS)  _kernel(1) void learner(...) { ... }
_at(ACCEPTORS) _kernel(1) void acceptor(...) { ... }
```

Fig. 11: P4xos [20] kernels and memory at three locations.

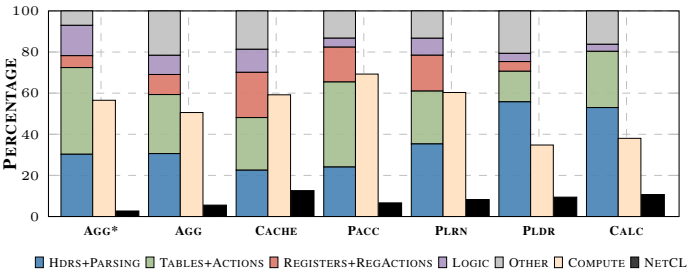

Fig. 12: Breakdown of P4 code in our test applications.

APPLICATION	NETCL	P4*	P4	REDUCTION
SWITCHML [13] (AGG)	38	1139	686	29.97 | 18.05
NETCACHE [16]† (CACHE)	91	692	723	7.60 | 7.94
P4XOS [20]‡	74	381	901	5.14 | 12.17
ACCEPTOR (PACC)	38	230	573	6.05 | 15.08
LEARNER (PLRN)	33	241	436	7.30 | 13.21
LEADER (PLDR)	26	214	276	8.23 | 10.61
CALCULATOR [78]‡ (CALC)	25	139	234	5.56 | 9.36
			GEOMEAN:	8.14 | 11.93

TABLE III: Lines of code in test applications. † = public P4 code written in $P4_{14}$, ‡ = public P4 code only targets the `v1model`.

		AGG	CACHE	PACC	PLRN	PLDR	CALC	EMPTY
P4	bf-p4c	8.82	5.50	3.35	2.30	1.88	1.39	–
NETCL	ncc	0.72	0.73	0.69	0.71	0.69	0.69	0.69
	bf-p4c	10.79	5.42	4.05	4.01	3.47	3.16	2.53
	Total	11.50	6.15	4.75	4.71	4.16	3.86	3.22

TABLE IV: Compilation times (seconds).

bitmap to track which 4-byte words of the cache line belong to a given key. Finally, hits are counted and misses go through a count-min sketch and then a bloom filter to decide whether the key should be reported as hot. Unlike [16] which issues new messages to a dedicated controller, we use an extra header field to mark misses as hot before forwarding them to the KVS server. Since the CACHE implementation is about 90 lines of code, it has been omitted from the paper, and can be found in the paper's artifact material [77]

The third application is the in-network Paxos algorithm of [20]. It requires three kernels, at three locations, shown in Figure 11. Since their implementation is almost identical to the pseudocode in [20] we have omitted it for space efficiency. Finally, CALC is a simple calculator from the P4 tutorials [78].

Language design. Table III shows the lines of code (LoC) required to program our test applications in NetCL, versus P4. The P4* column refers to the publicly available P4 code for each application. Since NetCL does not currently support all the features they do, we isolate only the relevant lines (e.g., remove RDMA-related code from AGG and so on). Moreover, we note that the published SWITCHML code [79] failed to compile, the NETCACHE code [53] is written in $P4_{14}$ and is incomplete, and the P4XOS code [80] is written for the `v1model`. To homogenize LoC measurements and have usable code for the other experiments, we implemented all applications in $P4_{16}$ ourselves, faithfully replicating the functionality of P4*. The LoC of those implementations are shown in column P4.

Overall, NetCL requires $O(10)$ LoC, whereas P4 requires $O(100)$, for the same functionality. We observe an average LoC reduction of 8x and 12x compared to P4* and P4, respectively. The difference between P4 and P4* is mainly because translating P4XOS to Tofino P4 introduces a lot of Tofino-specific code (e.g. `RegisterActionss`), not present in `v1model` P4. To better understand the sources and effectiveness of these reductions, we measured the distribution of code across different P4 constructs. Figure 12 shows the results.

On average, over 65% of P4 code is spent on packet-processing constructs like headers, parsers, and MATs, with about 30% spent on header definitions and parsing alone. The (somewhat) more familiar `RegisterAction` objects account for 13% of the stateful applications' code and in most cases require MATs to invoke them. Considering that only 10% of the code amounts to control logic, most imperative code is embedded in MATs and `RegisterAction` objects that together account for over 40%. Moreover, only 52% of the P4 code is spent on compute-related functionality, meaning that the programmer spends half of their time on network plumbing. NetCL code is only a fraction of the required P4 code (less than 13% in the worst case) and contains only compute logic.

Compilation time. We measure compilation times of NetCL using our compiler (`ncc`), and P4 using Intel's Tofino SDE 9.13.0 [81] (`bf-p4c`) on a standard desktop setup with an AMD Ryzen 7950x and 32GB of memory. Table IV shows the results. An average of five runs is reported. NetCL programs, on average, take 1.7x more time to compile compared to handwritten P4. However, most of the time, over 98%, is spent on P4 compilation. Our compiler introduces insignificant overhead, always finishing in less than one second.

Resources. The most important metric of P4 code quality when targeting Tofino is resource consumption. Table V summarizes the resource consumption of the P4 code generated by our compiler, and hand-written P4. We also include the empty program as an indication of the contribution of the base P4 program we generate code into. We show the number of stages each program requires, and, following [82] we also show the usage of various pipeline resources, as percentages. We report the total consumption across the pipeline on top, and the worst case stage-local consumption on bottom.

All applications were able to fit a Tofino pipe of 12 stages. For CACHE, the generated P4 code requires 3 extra stages. This is mainly due to the min calculation in count-min-sketch

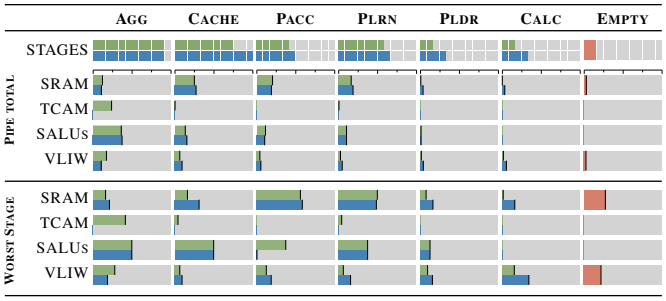

TABLE V: Tofino resource utilization of handwritten (■) and generated (■) P4. The empty program (■) contains no generated P4 code.

		AGG	CACHE	PACC	PLRN	PLDR	CALC	EMPTY
P4	LOCAL VARS	0	504	33	40	0	96	N/A
	HEADERS	1160	240	384	384	384	128	N/A
	METADATA	96	0	9	9	9	0	N/A
	WORST CASE PHV	35.8%	25.2%	23.4%	23.55%	23.2%	8.05%	N/A
NETCL	LOCAL VARS	96	208	0	8b	8	8	0
	IR ALLOCAS	32	176	0	8b	8	8	0
	P4 LOCAL VARS	128	325	48	72	40	40	0
	P4 HEADERS	1224	304	424	424	424	128	64
	P4 METADATA	60	60	60	60	60	60	60
	WORST CASE PHV	36.8%	27.4%	24.2%	24.75%	24.5%	20.5%	19.7%

TABLE VI: Sources of local memory for handwritten P4 vs. NetCL, in bits, and worst-case PHV occupancy. Only non-networking headers and metadata are reported. NetCL rows include the runtime's and base program's headers and metadata.

(lines 8-9 of Figure4). Our compiler generates this as a chain of subtractions and MSB checks. It is possible to implement it more efficiently with a MAT, however our compiler does not currently perform this optimization.

Overall, we observe a modest resource usage of the generated P4 that is in line with handwritten P4. In some cases, we even observe some improvements. For instance, the generated AGG code does not use TCAM. The reason is that for `atomic_cond_add_new` and `atomic_cond_dec` the compiler was able to generate code that evaluates the condition inside the `RegisterAction` using SRAM. Meanwhile, the handwritten P4 code, following [13], uses MATs with ternary lookups that do use TCAM. This is overall advantageous as TCAM is scarce and primarily needed for LPM-based L3 forwarding.

NetCL can increase PHV pressure in two ways. First, the compiler may generate additional local variables (e.g. due to CFG structurization §VI). Second, the shim NetCL header (Figure 10) adds extra bytes that need to be carried across the pipe, compared to handwritten P4 that works directly over an L4 protocol. Table VI shows the local memory requirements and their effect on PHV for NetCL and handwritten P4. We observe the biggest increase (12%) in CALC. This is expected as CALC is a very small program and PHV usage is dominated by the base program. CALC code generation only increases the latter by 1%. In all other cases, worst case PHV occupancy of the generated code is within 2% of the handwritten code, and up to 17% more than the base program.

Performance. Figure 13 shows the worst-case (no egress bypass) per-packet latency derived from the exact cycle costs reported by the Tofino compiler for each program. We observe that NetCL is within 9% of handwritten P4, on average. We note that those differences are in the order of 10s of cycles, and in all cases, total latency is well below 1μs. Moreover, due to Tofino's design, line-rate is guaranteed for any program that successfully compiles. For CACHE, which could potentially be latency-critical, we observe no difference.

Given Tofino's line rate guarantees, our compiler "succeeds" if it generates a P4 program that fits. In other words, end-to-end application performance cannot be degraded by the device program, especially when the objective is throughput, and largely depends on the host-side networking performance. By extension, given equal communication methods (e.g., UDP sockets), host-side implementations, and hardware, we should

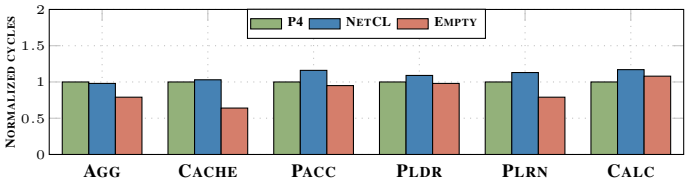

Fig. 13: Device packet-processing latency.

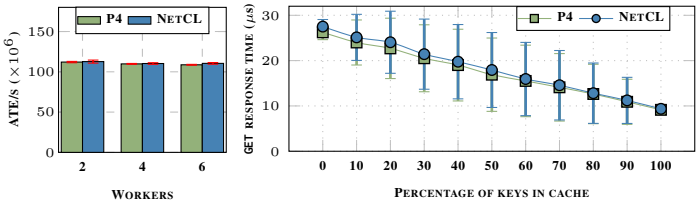

Fig. 14: End-to-end AGG throughput and CACHE response time

expect equivalent performance between NetCL and handwritten P4. To validate those claims we perform two end-to-end experiments where we fix the host program for both versions. One for the throughput-oriented AGG and one for the latency-oriented CACHE. P4XOS requires either three switches or co-locating multiple kernels to one. Since we did not have enough switches and a proper deployment system capable of co-locating kernels is left for future work, we have omitted it. We used a cluster of six servers, each equipped with a 24-core AMD EPYC 7443P processor, 256GB of memory, and a Mellanox Connectx-6 100G NIC, running Ubuntu Server 22.04.4 LTS. Servers are connected to a Netberg Aurora 710 switch [83], equipped with a 3.2 Tbps, dual-pipe Tofino ASIC.

Following [13] we measure AGG throughput in Aggregated Tensor Elements (ATE) per second, per worker. Figure 14 (left) shows the results for 2, 4, and 6 workers. We observe no difference between NetCL and handwritten P4. Similar to [13] we see that adding more workers does not degrade per-worker aggregation throughput.

Figure 14 (right) shows the mean CACHE response time for different amounts of cached keys. When all requests miss the cache, we observe 26 and 27 μs mean response time for P4 and NetCL, respectively, and 9.1 and 9.4 μs when all requests hit the cache. Given the equivalent device latencies reported in Figure 13, we attribute the differences to host-side packet processing costs.

VIII. Discussion and future work

At first sight, NetCL seems like a restricted version of C/C++. However, this only reflects the constraints of the hardware it targets. NetCL's APIs aim to be the lowest common denominator among the capabilities of programmable switching hardware, which by nature is heavily restricted to sustain Tb/s switching rates. Processing at line rate requires the offloaded functionality to be simple, with as little state management as possible [84]. Arbitrarily complex code is simply not compatible with existing devices [85].

While this work focused on P4 and Intel's Tofino, other architectures like Juniper's Trio [4] may offer opportunities to expand our model (e.g., run-to-completion processing, deeper memory hierarchies, and so on). Other PDP architectures like FPGAs [86] or Taurus [85], which are capable of more compute per packet, are interesting future targets. The advantage of using a C/C++ API is that we can introduce new features on the fly by simply lifting restrictions on a per-target basis.

Our system currently comes with certain limitations. It assumes a single communication backend for each computation. However, there may be a need for more. For instance, the original NetCache uses UDP for `GET` requests and TCP for `PUT` and `DEL`. Additionally, the computation specifications (§V) force all nodes to process the same packet layouts, which can result in transferring more data than needed. For instance, in Figure 6, the client sends zeros as a placeholder for the value in the response. In [16], clients only send the key, and the value is appended by the switch. NetCL could be extended with a message *tail* abstraction to support this flexibility.

Tofino 1 can write two 32-bit values in a `Register`, but only read one [13]. This is something that the original SwitchML takes advantage of to aggregate 64 values per pipe (with recirculation). In addition, they use a second recirculation path, across all 4 pipes, to support 256 values per packet. NetCL does not currently support either. The write-two-read-one behavior can be exposed as Tofino-specific intrinsics that, combined with the `repeat()`, can achieve 64 values per pipe. On the other hand, cross-pipe recirculation likely requires a new abstraction, to be handled at the language level. An alternative would be to write Agg as multiple kernels for 4 devices and explicitly chain them in kernel code. This is something that can already be done (for 32-values per device). However, this only pushes the problem to the deployment system and the added complexity of the solution is unclear.

Even without the aforementioned complexities, a deployment system like the one we propose is far from trivial. It is at least as hard of a problem as server scheduling and deployment, with the addition of network devices. For instance, we need to find switches with enough available resources in the base program to fit the NetCL code, while respecting the intended abstract topology. Those switches will also require additional configuration for the NetCL runtime and the tables processing the NetCL header. Future work could investigate existing literature [36], [87] towards that direction.

Finaly, our compiler, while capable of generating good enough P4 code, is in a sense the bare minimum required to translate C/C++ to P4. It is based on heuristics and does not explicitly model low-level Tofino details (mostly due to their unavailability) that could potentially improve resource utilization of the generated code. Future work could investigate incorporating ideas from the literature that focus on RMT code-generation, e.g., [56], [88], [89].

Interface	Syntax	Exposed Networking	PDP Features	CP	Multi-device
Domino [89]	C	●	◐	✗	✗
Snap [34]	Other	●	●	✓	✓
Lyra [32]	New	●	●	✗	✓
Lucid [33]	New	●	●	✗	✗
O4 [52]	P4+	●	●	✗	✗
P4All [35]	P4+	●	●	✗	✗
μP4 [30]	P4+	●	●	✗	✗
P4RROT [90]	New	◐	◐	✗	✗
sPIN [91]	C/C++	◐	◐	✓	✗
NetRPC [37]	RPC	○	◐	~	✗
ClickINC [36]	New	○	◐	✗	✓
NetCL	C/C++	○	◐	✓	✓

TABLE VII: Comparison with related PDP interfaces.

IX. Related Work

Several works in the literature have proposed PDP APIs alternative to P4 (or NPL). Table VII summarizes them and compares them against our work in 5 domains: syntax, amount of exposed networking constructs, amount of PDP features used, control plane integration, and support for multi-device programming. Most works do not focus on INC and thus expose a great deal of networking to the programmer [30], [32], [33], [35], [89], hardly making any difference for INC. Works that do focus on INC adopt a bottom-up approach, with APIs tailored around existing applications, and either offer limited expressiveness or require learning a new language [36], [37]. In short, no existing work fully addresses the requirements we have identified in §III.

X. Conclusion

In this paper, we have presented a compute-centric programming model for INC, based on familiar constructs like kernels, device functions, and memory, steering away from the networking abstractions exposed by contemporary PDP languages. We have implemented our system as C/C++ extensions and libraries, and developed an LLVM-based compiler targeting Intel's Tofino ASIC. Our evaluation using representative applications from the INC literature has shown that NetCL requires an order of magnitude fewer lines of code than handwritten P4 while matching its performance and resource consumption.

XI. Acknowledgements

This research was supported by the Dutch Research Council (NWO) grant OCENW.KLEIN.209 and, in part, by Google Research and Intel's contribution of Tofino hardware through the ICRP Fast Forward Initiative (FFI'22).

REFERENCES

[1] O. Michel, R. Bifulco, G. Retvari, and S. Schmid, "The programmable data plane: Abstractions, architectures, algorithms, and applications," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–36, 2021.

[2] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A survey on data plane programming with p4: Fundamentals, advances, and applied research," *Journal of Network and Computer Applications*, vol. 212, p. 103561, 2023.

[3] Intel, "Intel® intelligent fabric processors," https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors.html, 2023.

[4] M. Yang, A. Baban, V. Kugel, J. Libby, S. Mackie, S. S. R. Kananda, C.-H. Wu, and M. Ghobadi, "Using trio: juniper networks' programmable chipset-for emerging in-network applications," in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022, pp. 633–648.

[5] L. Tan, W. Su, W. Zhang, J. Lv, Z. Zhang, J. Miao, X. Liu, and N. Li, "In-band network telemetry: A survey," *Computer Networks*, vol. 186, p. 107763, 2021.

[6] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh *et al.*, "Hpcc: High precision congestion control," in *Proceedings of the ACM special interest group on data communication*, 2019, pp. 44–58.

[7] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 15–28.

[8] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. V. Ramos, and A. Madeira, "Flowlens: Enabling efficient flow classification for ml-based network security applications," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[9] Z. Liu, H. Namkung, G. Nikolaidis, J. Lee, C. Kim, X. Jin, V. Braverman, M. Yu, and V. Sekar, "Jaqen: A {High-Performance}{Switch-Native} approach for detecting and mitigating volumetric {DDoS} attacks with programmable switches," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3829–3846.

[10] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford, "Beaucoup: Answering many network traffic queries, one memory update at a time," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 226–239.

[11] X. Chen, "Implementing aes encryption on programmable switches via scrambled lookup tables," in *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, 2020, pp. 8–14.

[12] S. Kianpisheh and T. Taleb, "A survey on in-network computing: Programmable data plane and technology specific applications," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 1, pp. 701–761, 2022.

[13] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik, "Scaling distributed machine learning with In-Network aggregation," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021, pp. 785–808. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/sapio

[14] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift, "{ATP}: In-network aggregation for multi-tenant learning," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 741–761.

[15] D. De Sensi, S. Di Girolamo, S. Ashkboos, S. Li, and T. Hoefler, "Flare: Flexible in-network allreduce," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–16.

[16] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 121–136. [Online]. Available: https://doi.org/10.1145/3132747.3132764

[17] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica, "Distcache: Provable load balancing for large-scale storage systems with distributed caching," in *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, A. Merchant and H. Weatherspoon, Eds. USENIX Association, 2019, pp. 143–157. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/liu

[18] Z. Yu, Y. Zhang, V. Braverman, M. Chowdhury, and X. Jin, "Netlock: Fast, centralized lock management using programmable switches," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 126–138.

[19] M. Kogias and E. Bugnion, "Hovercraft: achieving scalability and fault-tolerance for microsecond-scale datacenter services," in *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds. ACM, 2020, pp. 25:1–25:17. [Online]. Available: https://doi.org/10.1145/3342195.3387545

[20] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé, "P4xos: Consensus as a network service," *IEEE/ACM Transactions on Networking*, vol. 28, no. 4, pp. 1726–1738, 2020.

[21] J. Li, E. Michael, and D. R. K. Ports, "Eris: Coordination-free consistent transactions using in-network concurrency control," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 104–120. [Online]. Available: https://doi.org/10.1145/3132747.3132751

[22] G. Sun, M. Jiang, X. Z. Khooi, Y. Li, and J. Li, "Neobft: Accelerating byzantine fault tolerance using authenticated in-network ordering," in *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, H. Schulzrinne, V. Misra, E. Kohler, and D. A. Maltz, Eds. ACM, 2023, pp. 239–254. [Online]. Available: https://doi.org/10.1145/3603269.3604874

[23] M. Tirmazi, R. Ben Basat, J. Gao, and M. Yu, "Cheetah: Accelerating database queries with switch pruning," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2407–2422.

[24] Z. Xiong and N. Zilberman, "Do switches dream of machine learning?: Toward in-network classification," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks, HotNets 2019, Princeton, NJ, USA, November 13-15, 2019*. ACM, 2019, pp. 25–33. [Online]. Available: https://doi.org/10.1145/3365609.3365864

[25] K. Razavi, G. Karlos, V. Nigade, M. Mühlhäuser, and L. Wang, "Distributed DNN serving in the network data plane," in *Proceedings of the 5th International Workshop on P4 in Europe, EuroP4 2022, Rome, Italy, 9 December 2022*, M. Chiesa and S. L. Feibish, Eds. ACM, 2022, pp. 67–70. [Online]. Available: https://doi.org/10.1145/3565475.3569079

[26] T. A. Benson, "In-network compute: Considered armed and dangerous," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2019, pp. 216–224.

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

[28] nplang, "Npl - open, high-level language for developing feature-rich solutions for programmable networking platforms," https://nplang.org/, 2023.

[29] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM computer communication review*, vol. 38, no. 2, pp. 69–74, 2008.

[30] H. Soni, M. Rifai, P. Kumar, R. Doenges, and N. Foster, "Composing dataplane programs with μp4," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 329–343.

[31] P. Zheng, T. Benson, and C. Hu, "P4visor: Lightweight virtualization and composition primitives for building and testing modular programs," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, 2018, pp. 98–111.

[32] J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, and M. Yu, "Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 435–450. [Online]. Available: https://doi.org/10.1145/3387514.3405879

[33] J. Sonchack, D. Loehr, J. Rexford, and D. Walker, "Lucid: A language for control in the data plane," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 731–747.

[34] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "Snap: Stateful network-wide abstractions for packet processing," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 29–43.

[35] M. Hogan, S. Landau-Feibish, M. T. Arashloo, J. Rexford, and D. Walker, "Modular switch programming under resource constraints," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 193–207.

[36] W. Xu, Z. Zhang, Y. Feng, H. Song, Z. Chen, W. Wu, G. Liu, Y. Zhang, S. Liu, Z. Tian *et al.*, "Clickinc: In-network computing as a service in heterogeneous programmable data-center networks," in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023, pp. 798–815.

[37] B. Zhao, W. Wu, and W. Xu, "{NetRPC}: Enabling {In-Network} computation in remote procedure calls," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 199–217.

[38] Nvidia, "Cuda toolkit documentation," https://docs.nvidia.com/cuda/, 2023.

[39] J. Krüger and R. Westermann, "Linear algebra operators for gpu implementation of numerical algorithms," in *ACM SIGGRAPH 2005 Courses*, 2005, pp. 234–es.

[40] G. Karlos, H. Bal, and L. Wang, "Don't you worry 'bout a packet: Unified programming for in-network computing," in *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, ser. HotNets '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 99–107. [Online]. Available: https://doi.org/10.1145/3484266.3487395

[41] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[42] I. Ashkenazi, "Pbt-on-demand on mellanox p4-capable hybrid switch," https://opennetworking.org/wp-content/uploads/2020/04/Itzik-Ashkenazi-Slide-Deck.pdf, 2020.

[43] Cisco, "Silicon one Q100 and Q100L processors data sheet," https://www.cisco.com/c/en/us/solutions/collateral/silicon-one/datasheet-c78-744214.html, 2020.

[44] AMD, "Vitis networking p4 user guide," https://docs.amd.com/r/en-US/ug1308-vitis-p4-user-guide/Introduction, 2023.

[45] ——, "Amd pensando™ infrastructure accelerators," https://www.amd.com/en/accelerators/pensando, 2024.

[46] E. Peer, "Mapping P4 to smartnics," https://opennetworking.org/wp-content/uploads/2020/12/p4_d2_2017_nfp_architecture.pdf, 2017.

[47] T. P. A. W. Group, "P4runtime specification," 2020-12-01.

[48] T. P. L. Consortium, "$P4_{16}$ language specification," https://staging.p4.org/p4-spec/docs/P4-16-v1.2.4.html, 2023.

[49] Intel, "$P4_{16}$ intel® tofino™ native architecture - public version," https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf, 2021.

[50] Broadcom, "Trident4 / bcm56880 series," https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series, 2024.

[51] 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," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[52] A. G. Alcoz, C. Busse-Grawitz, E. Marty, and L. Vanbever, "Reducing p4 language's voluminosity using higher-level constructs," in *Proceedings of the 5th International Workshop on P4 in Europe*, 2022, pp. 19–25.

[53] netx repo, "netcache-p4," https://github.com/netx-repo/netcache-p4, 2018.

[54] M. D. Ernst, G. J. Badros, and D. Notkin, "An empirical analysis of c preprocessor use," *IEEE Transactions on Software Engineering*, vol. 28, no. 12, pp. 1146–1170, 2002.

[55] AMD, "Vitis networking p4 user guide - supported p416 language features," https://docs.xilinx.com/r/en-US/ug1308-vitis-p4-user-guide/Supported-P416-Language-Features, 2023.

[56] Y. Li, J. Gao, E. Zhai, M. Liu, K. Liu, and H. H. Liu, "Cetus: Releasing p4 programmers from the chore of trial and error compiling," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 371–385.

[57] Intel, "Tna `RegisterAction` declaration," https://github.com/barefootnetworks/Open-Tofino/blob/master/share/p4c/p4include/tofino1_base.p4#L675-L701, 2024.

[58] O. N. Foundation, "fabric.p4," https://github.com/opennetworkinglab/onos/blob/master/pipelines/fabric/impl/src/main/resources/fabric.p4, 2021.

[59] D. Project, "Data plane development kit," https://www.dpdk.org/, 2024.

[60] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[61] p4lang, "Behavioral model targets," https://github.com/p4lang/behavioral-model/tree/main/targets, 2023.

[62] cppreference, "Fundamental types," https://en.cppreference.com/w/cpp/language/types, 2023.

[63] ——, "Array declaration," https://en.cppreference.com/w/cpp/language/array, 2023.

[64] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th international conference on emerging networking experiments and technologies*, 2018, pp. 54–66.

[65] L. Zeno, D. R. Ports, J. Nelson, D. Kim, S. Landau-Feibish, I. Keidar, A. Rinberg, A. Rashelbach, I. De-Paula, and M. Silberstein, "{SwiSh}: Distributed shared state abstractions for programmable switches," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 171–191.

[66] Clang, "Clang: a C language family frontend for LLVM," https://clang.llvm.org/, 2023.

[67] nvidia, "The cuda compilation trajectory," https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#the-cuda-compilation-trajectory, 2023.

[68] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt, "gpucc: an open-source gpgpu compiler," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, 2016, pp. 105–116.

[69] T. C. Team, "Offloading design & internals," https://clang.llvm.org/docs/OffloadingDesign.html, 2023.

[70] LLVM, "Llvm's analysis and transform passes," https://llvm.org/docs/Passes.html, 2023.

[71] T. K. Dangeti, R. Upadrasta *et al.*, "P4llvm: An llvm based p4 compiler," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 2018, pp. 424–429.

[72] Y.-H. Lai, E. Ustun, S. Xiang, Z. Fang, H. Rong, and Z. Zhang, "Programming and synthesis for software-defined fpga acceleration: status and future prospects," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 14, no. 4, pp. 1–39, 2021.

[73] LLVM, "Llvm language reference manual," https://llvm.org/docs/LangRef.html, 2023.

[74] H. Wu, G. Diamos, S. Li, and S. Yalamanchili, "Characterization and transformation of unstructured control flow in gpu applications," in *1st international workshop on characterizing applications for heterogeneous exascale systems*, 2011.

[75] N. Reissmann, T. L. Falch, B. A. Bjørnseth, H. Bahmann, J. C. Meyer, and M. Jahre, "Efficient control flow restructuring for gpus," in *2016 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2016, pp. 48–57.

[76] F. Rastello and F. B. Tichadou, *SSA-based Compiler Design*. Springer Nature, 2022.

[77] G. Karlos, H. Bal, and L. Wang, "netcl-paper-artifact," Aug. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.13328729

[78] p4lang, "P4 tutorial," https://github.com/p4lang/tutorials, 2023.

[79] ——, "p4app-switchml," https://github.com/p4lang/p4app-switchML, 2021.

[80] usi systems, "p4xos-public," https://github.com/usi-systems/p4xos-public, 2018.

[81] Intel, "Intel connectivity research program," https://www.intel.com/content/www/us/en/products/docs/network-io/intelligent-fabric-processors/connectivity-education-hub/research-program.html, 2023.

[82] Y. Yuan, O. Alama, J. Fei, J. Nelson, D. R. Ports, A. Sapio, M. Canini, and N. S. Kim, "Unlocking the power of inline {Floating-Point} operations on programmable switches," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 683–700.

[83] Netberg, "Aurora 710," https://netbergtw.com/products/aurora-710, 2024.
[84] D. R. Ports and J. Nelson, "When should the network be the computer?" in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2019, pp. 209–215.
[85] T. Swamy, A. Rucker, M. Shahbaz, I. Gaur, and K. Olukotun, "Taurus: a data plane architecture for per-packet ml," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 1099–1114.
[86] P. Bressana, N. Zilberman, D. Vucinic, and R. Soulé, "Trading latency for compute in the network," in *Proceedings of the Workshop on Network Application Integration/CoDesign*, 2020, pp. 35–40.
[87] M. Blöcher, L. Wang, P. Eugster, and M. Schmidt, "Switches for hire: Resource scheduling for data center in-network computing," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 268–285.
[88] X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. K. Varma, P. G. Kannan, A. Sivaraman, S. Narayana, and A. Gupta, "Switch code generation using program synthesis," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 44–61.
[89] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 15–28.
[90] C. Györgyi, S. Laki, and S. Schmid, "P4rrot: Generating p4 code for the application layer," *ACM SIGCOMM Computer Communication Review*, vol. 53, no. 1, pp. 30–37, 2023.
[91] T. Hoefler, S. D. Girolamo, K. Taranov, R. E. Grant, and R. Brightwell, "sPIN: High-performance streaming Processing in the Network," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC17)*, 2017.