Don't You Worry 'Bout a Packet: Unified Programming for In-Network Computing

George Karlos
Vrije Universiteit Amsterdam

Henri Bal
Vrije Universiteit Amsterdam

Lin Wang
Vrije Universiteit Amsterdam

ABSTRACT

In-network computing is gaining momentum as programmable switches are increasingly employed for compute acceleration. Designed for packet processing, data plane programming languages force developers to express *compute* in *networking* terms, resulting in a complex, error-prone practice. We envision the unification of switch and host programming and propose the Net Compute Language (NCL), a C/C++ extension for expressing computational kernels for switches to execute. NCL implements Compute Centric Communication (C3), our proposed programming model for INC under which, point-to-point primitives are augmented to carry out computations. We motivate our approach with real-world use cases and discuss the technical challenges for its realization.

ACM Reference Format:
George Karlos, Henri Bal, and Lin Wang. 2021. Don't You Worry 'Bout a Packet: Unified Programming for In-Network Computing. In *The Twentieth ACM Workshop on Hot Topics in Networks (HotNets '21), November 10–12, 2021, Virtual Event, United Kingdom.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3484266.3487395

1 INTRODUCTION

The fast evolution of software-defined networking (SDN) [14] has led to network switches capable of Tb/s processing while offering increasingly programmable data plane functionality [2, 8, 10, 19, 35, 37]. This development has allowed for unprecedented innovation in networking [3, 40, 57], and given rise to a new paradigm: in-network computing (INC) [44, 55].

Under INC, application-specific computations occur *inside* the network, improving overall throughput, latency and even energy efficiency [55]. Prior work has realized the potential of programmable switches on a variety of distributed services such as data aggregation [47], caching [23, 29], stream processing [21], query processing [28, 54], agreement [12, 22, 60], and ML training [17, 26, 48]. Offloading heavy-duty tasks like (de)compression [56] and ML inference [46, 52, 59], or even simple data transformations [25], to on-path switches has shown potential for substantial performance gains.

To aid data plane customization, a healthy number of languages have been proposed [5, 7, 49, 50], with P4 [5] and NPL [7] arguably the most popular. Bearing API differences, data plane languages share two fundamental properties. First, they are designed around network functionality and thus expose verbose packet processing. Second, modern switching fabrics rely on application-specific integrated circuits (ASICs) to maintain high speeds. These are not akin to general purpose programming, so data plane languages are necessarily confined to a programming model close to the hardware.

The above characteristics translate to constructs like packet parsers and match-action tables that, while crucial to packet processing, fall short for expressing compute. Programmers are thus forced to encode application logic in unfamiliar terms, often employing clever tricks to realize simple functionality. INC applications are encoded as L4/L5 protocols, which also complicates host side code with packet crafting concerns. Such hurdles make INC programming difficult and error-prone, inhibiting the realization of its full potential.

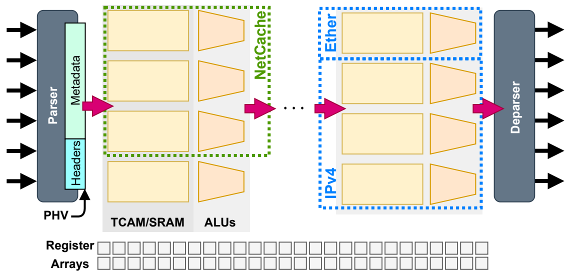
Driven by numerous INC successes, we believe it is time to view the network as yet another accelerator. But, to achieve this, a fitting programming model is required. One that existing data plane languages do not offer. To that end, we introduce the Compute Centric Communication (C3) model for INC. In C3, hosts exchange data arrays in user-defined chunks, by communication primitives programmed to also perform computations on them. We propose the Net Compute Language (NCL) to realize C3 and unify switch/host programming by letting programmers express such computations in C/C++. Its compiler targets both switches and hosts, and a runtime transparently handles network plumbing. Our system relieves programmers from packet processing concerns, letting them focus on application logic. In the remaining of this paper, we present and motivate our vision.

2 BACKGROUND AND MOTIVATION

Over the years, P4 [5] has become the de facto data plane programming standard, supported by switches [2, 10, 19, 37],

HotNets '21, November 10–12, 2021, Virtual Event, United Kingdom

ACM ISBN 978-1-4503-9087-3/21/11.
https://doi.org/10.1145/3484266.3487395



(a) Protocol-independent switch architecture (PISA)

```
action CacheHit(idx) { meta.hit = true; meta.idx = idx; }
action ReadValid() { meta.valid = Valid.read(meta.idx); }
table CacheLookup { key = {headers.cache.key: exact }
                    actions = {CacheHit} }
table CacheValid { actions = {ReadValid} }

CacheLookup.apply();
if ( meta.hit && headers.cache.op == GET)
  CacheValid.apply();
  if ( meta.valid)
    Read0.apply(); Read1.apply(); ...
ipv4.apply(); ether.apply();
```

Networking
Compute

(b) In-network KVS cache (GET) based on NetCache [23]

Figure 1: P4 INC application and its mapping on PISA

SmartNICs [34, 43, 58] and even DPUs [37]. P4 implements the protocol-independent switch architecture (PISA) shown in Fig. 1a, a generalization of RMT [6] and dRMT [9].

Packet processing starts with a programmable parser that extracts headers into the packet header vector (PHV), together with user-defined and architecture-defined metadata. The PHV is processed by a pipeline in VLIW fashion. At each stage it is matched against match-action tables (MATs), where match-rules (stored in TCAM/SRAM) determine actions for the stage's ALUs. Actions are programmable and can modify the PHV and persistent register arrays. Finally, a deparser programmatically reconstructs the packet.

Considerable efforts have been made to simplify data plane programming [15, 16, 49]. Yet, existing solutions are unsuitable for INC as they all fundamentally revolve around packet processing. In particular, we identify the following obstacles:

Complex programming semantics. INC programming in P4 requires deep understanding of packet processing with PISA. Resulting code is typically long, even when expressing simple logic. Fig. 1b sketches the `GET` operation of NetCache [23], an in-network KVS cache. A MAT is applied to look up a key. On hit, a flag and the index of the value in a register array are written to metadata (PHV). The flag is checked and if set, the validity of the value is checked by applying another table to read from the `Valid` register. If valid, multiple tables are applied (`Read0`, `Read1`) to retrieve the value, each reading a portion and writing it to the PHV. Such indirections result in obnoxious control flow and structure that often resembles assembly code, suggesting that a compiler could handle it with better correctness guarantees.

Tedious network plumbing. INC programmers need to deal with normal network operations such as IPv4 routing and Ethernet forwarding, as well as the protocol encoding their application. This complicates both switch and host programming. The code of Fig. 1b only executes if the parser (not shown in the figure) has recognized the NetCache protocol. To do that, it must be programmed to parse the entire header stack, including L2 and L3. In addition, the routing/forwarding behavior of the underlying protocols must be incorporated into the program, by defining and applying the appropriate tables. Finally, on the host side, packets that follow the INC application protocol must be crafted. Such plumbing requires programmers to have profound knowledge in networking, which inevitably raises the bar for INC.

Inflexible development and deployment. A disjoint development process for such heterogeneous systems can lead to subtle compatibility bugs (e.g., type system and endianness differences) that are hard to catch and fix. This increases development and maintenance costs. Since P4 stays at the single device level, an application spanning multiple switches has to be manually partitioned into separate P4 programs and separately deployed. This requires good knowledge of the target platform to be available beforehand.

3 A PROGRAMMING METHOD FOR INC

To open up INC programming to non-networking experts, we introduce Compute Centric Communication (C3), a programming model that treats the network as an accelerator, and propose a complete programming system based on it.

3.1 The C3 programming model

C3 is an array-based model. Hosts exchange data arrays through point-to-point communication primitives that also perform computations on them, on network devices between the source and destination. Hence, INC is initiated on-demand by the sender. Unlike traditional *send/recv*, C3 communication primitives are applied to multiple arrays simultaneously.

Central to C3 is the *window* abstraction that is used to hide the details of packet-based communication. Arrays are transported one window at a time, and a one-to-one correspondence with packets is not necessary (§4.2). Windows associate elements across arrays, in a user-controlled way, in order to form a basic unit of processing.

Programmers express compute in *network kernels*. These are window-processing functions that receivers of windows (switches and hosts) execute on receipt. They can modify window data and device state, and make small forwarding decisions (§4.1). C3 does not specify a transport mechanism, thus a network kernel *defines* a protocol-agnostic, window-based communication primitive.

Fig. 2 shows an INC example in C3. Host-A is directly connected to a programmable switch, and Host-B also has

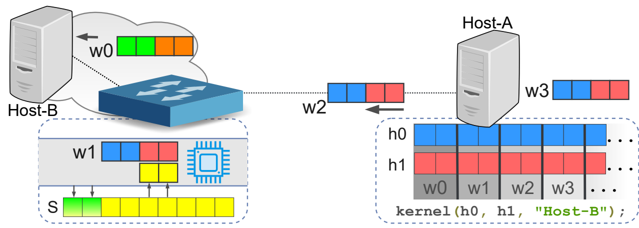


Figure 2: In-network computation under C3

connectivity to it. Host-A initiates the in-network computation by invoking a kernel to send its arrays h0, h1 to Host-B.

Windows are constructed (w0, w1, ...) and sent out one by one. In this case, the arrays are "split" evenly in windows of length two. Host-A has sent the first three windows and is about to send w3. Window w2 is currently on the wire. Window w1 is on the switch and the kernel is executing *on* it, using both data from w1 and a switch array S. Window w0 has already been processed by the switch. According to the kernel's programming, w0 has been modified and forwarded towards Host-B, where it will be handled by another kernel.

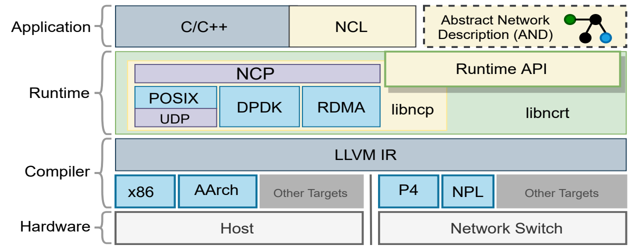
3.2 A programming system for C3

Our proposed programming system consists of a domain-specific language (DSL) for programming network kernels, its compiler and supporting libraries. Fig. 3a shows an overview.

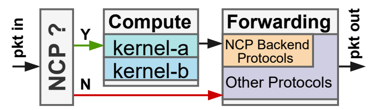
The Net Compute Language (NCL) extends C/C++ with the ability to describe network kernels, and network device resources - similar in fashion to the CUDA [36] / OpenCL [24] extensions for GPU/FPGA/DSP acceleration. Its standard library includes kernel and resource handling APIs as well as a collection of switch-side data-structures. For instance, fast MAT lookups can be exposed as Maps or bloom-filters.

NCL exposes no networking concerns, with the exception of a small, declarative API for influencing window forwarding. Such a mechanism is required when different switches or hosts have different roles [12] and/or window processing location depends on runtime conditions [23]. For this reason, programmers can provide an Abstract Network Description (AND), defining an overlay network configuration of the functional components of their application. Window forwarding inside a kernel is parameterized by the location labels specified in the AND. NCL also allows programmers to place different resources and/or different versions of a kernel on different devices, again using AND location labels.

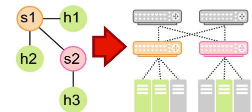
The runtime component of NCL, libncrt, has a multi-faceted role. First, it includes definitions for built-in types, constants, and function overlays required by the NCL extensions. NCL kernels are written for the data plane, but may involve the control plane under the hood. For instance, host code is allowed to update variables that are read-only by switch code. Transparent control-plane interaction is also part of the runtime. Finally, it implements the windowing mechanism completely transparently to the user. This means



(a) The NCL software stack



(b) Switch behavior



(c) Mapping to real net

Figure 3: A complete programming system for C3

that when a kernel is invoked, windows are determined from a window specification provided by the programmer, and from them packets are constructed and sent out.

Window-based communication is carried out by the Net Compute Protocol (NCP). Besides being a transport protocol for windows, NCP also encodes kernel execution context. For instance, it can carry information about which kernel to execute, the offsets of different array chunks, or bytes in a packet that the NCL compiler decided to use as scratch memory. Our goal for NCP is to be implemented over multiple transport backends, like POSIX sockets over UDP [32], DPDK [45] and RDMA/RoCE [33]. NCP is part of the runtime and, except for selecting a backend, the programmer does not interact with it directly. A switch executes a kernel only when the NCP protocol has been recognized, and will forward NCP packets based on the selected backend (Fig. 3b). As such, a portion of the runtime resides on the switch.

The NCL compiler is based on LLVM [27] and targets data-plane languages. It takes an NCL C/C++ program and an AND file and outputs a host binary, and a program for every switch in the AND file. Application deployment is out of scope. In general, a mechanism that maps the overlay network of the AND file into a physical network and allocates network resources accordingly [4] is assumed to be in place. Such mapping is shown in Fig. 3c. This mechanism places application components to physical devices and ensures connectivity by populating routing tables appropriately.

4 INC PROGRAMMING WITH NCL

In this section we discuss in detail the NCL's main components, namely, network (or NCL) kernels and windows, and sketch two example INC applications in NCL.

4.1 Network kernels

From the perspective of the application programmer, traditional communication primitives, like *send()* and *recv()*, have straightforward functions: (a) put data on the wire and (b) take data from the wire and deliver it to the application. Details of the mechanism depend on the underlying established protocol(s) and are mostly hidden. NCL kernels establish communication similarly, but they also perform computations to the data according to the kernel's programming.

NCL kernels run on devices that understand NCP. These are both network devices like switches, and end hosts. For this reason there are two kinds of network kernels, namely, *outgoing* and *incoming*. Outgoing kernels execute on switches and incoming kernels execute on hosts to handle the receipt of windows. A function is declared as a network kernel by the `_net_` declaration specifier, and incoming/outgoing kernels are distinguished by a second declaration specifier. The `_net_ _out_` combination denotes an outgoing kernel and the `_net_ _in_` combination, an incoming kernel.

Outgoing kernels. Outgoing kernels resemble *send()* in that their invocation implies data is sent from the invoking host to another. It also implies that the data is processed by on-path switches, one window at a time and according to the kernel's programming. Unlike *send()*, an outgoing kernel can have multiple array inputs through its arguments.

To accommodate different kinds of applications, we envision two APIs for outgoing kernel invocations. The first one is data-centric and operates on entire arrays. That is, a kernel invocation completes only when all its input arrays have been consumed, resembling more a *send()* in a loop. The second one gives finer control to the programmer, letting them send individual windows. Such mechanism could become a building block for richer interfaces [1, 53].

Programmers can optionally supply the `_at_(label)` declaration specifier to restrict a kernel to a specific location. This allows to write multiple versions of the kernel for different switches with different roles. The label here must be a valid label in the AND file. Location-less kernels run on all switches in SPMD fashion. For this reason, a builtin `location` struct provides information about the current location such that divergent behavior can still be expressed.

Outgoing kernels run on network switches when a window arrives and have single-thread execution semantics. They have access to window data as well as switch memory, statically allocated by the programmer for stateful operations. Window data is accessed through the kernel's arguments and a builtin `window` struct provides information about the current window, including bits provided by the user (§4.2).

Switch memory is only accessible in kernel code and is declared through global variables prefixed with the `_net_` declaration specifier. Optionally, a location may also be supplied, using the `_at_(label)` declaration specifier with a valid AND label. Location-less switch memory exists on all switches, however, modifications to it are local. That is, NCL makes no consistency guarantees, as distributed shared state in the data plane is still an open problem [61].

NCL also exposes control variables. These reside on switches, but are read-only from kernel code and written only by host code. Control variables are declared by the `_net_ _ctrl_ _at_(label)` combination of declaration specifiers, i.e. location is required. Again, NCL makes no consistency guarantees and out-of-band mechanisms, potentially involving the network controller (e.g., ONOS [41]), are required.

Finally, outgoing kernels can make simple forwarding decisions for a window. They can return the window to the previous hop (`_reflect()`), pass it on (`_pass()`, default behavior), broadcast it (`_bcast()`), or drop it (`_drop()`). Their behavior depends on the AND file. For instance, `_bcast()` sends a window to all devices, one hop away - in the overlay - from the current location, and `_pass()` can also accept a valid label from the AND as a parameter.

Incoming Kernels. Incoming kernels resemble *recv()*. They are invoked when a window is expected by the host, and execute when it arrives. They have read/write access to window data, and, unlike outgoing kernels, can also access host memory. A location is meaningless for incoming kernels because they exist on all hosts.

An incoming kernel is "paired" with an outgoing kernel and must match its parameter list so that window data is accessed in the same manner. Host memory is accessed through global variables or by extending the incoming kernel's parameter list and passing additional host pointers. Extra parameters must be marked as `_ext_`. In its simplest form, an incoming kernel can just copy window data to host memory for subsequent processing.

4.2 Data windows

A window is NCL's abstraction over packets and the basic unit of processing for kernels. Windows are transparently constructed and encoded into packets by the runtime. While sharing similarities, windows are not packets. In fact, our aim is to disassociate the two: a packet can carry one or more windows, and a window can span multiple packets.

Constructing windows means associating values across arrays into chunks, to be processed together according to the application's needs. A declarative API gives the programmer agency over this process. For instance, they can specify a mask with the number of elements from each array. As an example, Fig. 2 uses a `{2,2,2}` mask to associate two elements from each array. A mask is associated with kernel invocations (Fig. 4 `main`), but its length must always match the number of pointers in an `_out_` kernel's signature.

NCL defines a builtin `window` struct, that is only accessible in kernel code and contains various metadata about the current window (e.g., sequence number, sender etc.). This struct can be extended by the programmer to include additional information that might be useful to the kernel. For

```
1. _net_ _at_("s1") int accum[DATA_LEN] = {0};
2. _net_ _at_("s1") unsigned count[DATA_LEN/WIN_LEN] = {0};
3. _net_ _at_("s1") _ctrl_ unsigned nworkers;
4.
5. _net_ _out_ void allreduce(int *data) {
6.   unsigned base = window.seq * window.len;
7.   for (unsigned i = 0; i < window.len; ++i)
8.     accum[base + i] += data[i];
9.   if (++count[window.seq] == nworkers) {
10.    memcpy(data, &accum[base], window.len * 4);
11.    count[window.seq] = 0; _bcast();
12.  } else { _drop(); }
13. }
14.
15. _net_ _in_ void result(int *data, _ext_ int *hdata,
16.                                   _ext_ bool *done){...}
17. int main() {
18.   ncl::ctrl_wr(&nworkers, 16);
19.   ncl::out(allreduce, {data}, wnd, mask);
20.   while (!done)ncl::in(result, {data, &done}, wnd, mask);}
```

Figure 4: A synchronous AllReduce operation in NCL

instance, for a uniform split like the one of Fig. 2, the programmer could attach a length field, or the entire mask itself. Extended `window` structs are associated with kernel definitions, but different instances can be attached to different kernel invocations.

4.3 Use Cases

We illustrate NCL by sketching the implementations of two common INC applications.

AllReduce. AllReduce is a collective aggregation operation. It is fundamental to distributed data-parallel ML training [42] and has been the subject of a good amount of INC literature [17, 26, 48]. Each worker i holds an array A_i and the goal is to compute an array B with $B[j] = \sum_i A_i[j]$. For instance, for arrays $\{1, 1, 1\}, \{2, 2, 2\}, \{3, 3, 3\}$ the result is $\{6, 6, 6\}$. Fig. 4 sketches an in-network AllReduce in NCL.

Workers are connected to a ToR switch, labelled *s1*, that aggregates their data and broadcasts the result. They invoke an `_out_` kernel (line 19) to send data to the switch and then iteratively invoke an `_in_` kernel (line 20) to handle incoming windows with aggregation results. An extended parameter list (line 15) allows to copy results (e.g., to update the model) and set a flag that controls the loop.

The switch code uses the `accum` array to accumulate values. Windows allow creating (implicit) aggregation slots in the `accum` array, sized by a window's length. The `count` array tracks the number of windows accumulated at each slot. On line 6, the kernel first computes an index for the first element of the window's slot in `accum`. Here, the `seq` field of the `window` struct is builtin, but the `len` field is user-provided. In the next step, the kernel iterates over window data, accumulating each value. Then, the slot's counter is incremented and compared against the `_ctrl_` variable `nworkers` to determine if the slot is finished. If equal, the values of that slot are copied to the window, the counter is reset, and the window is broadcasted. Otherwise, the window is dropped.

KVS Cache. An in-network cache sits between clients and storage servers. It serves `GET` queries for hot items directly and forwards the rest to a storage server. Fig. 5 sketches an implementation in NCL. To simplify the example we omitted hot item detection and the `DELETE` operation. We also used a single storage server. For key-partitioned storage clusters, the kernel is extended to `_pass()` windows accordingly.

The cache stores 256 items of 8-byte keys and 128-byte values. It is implemented as a combination of a `Map` from NCL's standard library (implicitly `_ctrl_`) for keys, and an array for values. The storage server controls the map and associates keys with indices to the `Cache` array that stores the values. This design resembles NetCache [23] and is needed because the map is implemented as a MAT under the hood, which (at the time of writing) is only managed by the control plane. The `Valid` array is used to track item validity.

On a `PUT` query (line 6), if the item is in the cache, it gets invalidated. A `PUT` query is always forwarded to the storage server, indicated by the absence of a forwarding decision in either path. The storage server uses the same kernel to do an update (line 12) that writes the new value, sets it valid, and drops the window. Note that the same code is used to insert a new value in the cache, with the exception that the storage server must first insert an entry to the `Idx` map.

On a `GET` query (line 8) the key is looked up. If found and valid, the value is written to the window and sent back to the client with a `_reflect()` call. In any other case the window is forwarded to the storage server; implicit `_pass()`. Finally, the kernel does nothing on a `GET` response from the server (line 15), i.e. the window is forwarded to the client.

Although not shown in Fig. 5, for a cache eviction, the storage server just removes an item from the `Idx` map.

5 THE NCL COMPILER ARCHITECTURE

The `nclc` compiler is based on LLVM [27]. It takes as input an NCL C/C++ program and an AND file, and targets LLVM supported architectures (e.g., x86, AArch) for host code and PISA architectures (e.g., PSA [18], TNA [20]) for switch code. As is typical with accelerator-targetting compilers [31, 39], `nclc` employs a dual compilation pipeline, shown in Fig. 6.

The frontend extends Clang [30] with minor rewriting and source level checks, and outputs two LLVM IR files: one for host and one for switch code. The host pipeline (Fig. 6 left) consists of typical C/C++ compilation steps with some minor instrumentation for the runtime. The device pipeline lowers switch side LLVM IR to P4 and "links" it with a template switch configuration. This is done in four stages:

Conformance checking. Not all LLVM IR maps to PISA. For instance, loops must have provably constant trip counts, or recursive calls are disallowed. Conservative dataflow analysis can catch these and reject the program. This stage does

```
_net_ _at_("s1") ncl::Map<uint64_t, uint8_t, 256> Idx;
_net_ _at_("s1") char Cache[256][128] = {{0}};
_net_ _at_("s1") bool Valid[256] = {false};

_net_ _out_ query(uint64_t key, char *val, bool update) {
  if (window.from != SERVER && update) {        // client PUT
    if (auto *idx = Idx[key]) Valid[*idx] = false;
  } else if (window.from != SERVER) {            // client GET
    if (auto *idx = Idx[key]) {                         // hit
      if (Valid[*idx]) {
        memcpy(val, Cache[*idx], 128); _reflect(); } }
  } else if (update) {                          // server update
    auto *idx = Idx[key]; memcpy(Cache[*idx], val, 128);
    Valid[idx] = true; _drop();
  } else { }                            // server GET response
}
```

Figure 5: An In-Network KVS cache (GET, PUT) in NCL

various IR level checks for e.g., location conflicts between kernels and switch memory or invalid window masks.

IR versioning. This stage uses location info from kernel signatures and the AND to create multiple IR modules, containing each location's kernels and `location` struct implementation. It may also attempt to split location-less kernels by inspecting top-level branching on `location` struct fields. Subsequent stages examine all IR modules this stage outputs.

Analysis and optimization. This stage analyzes and transforms the IR in preparation for code-generation. First, loops are unrolled, and typical early SSA optimizations are applied, like const. folding/propagation, GVN/CSE, DCE etc.

Next, we have generic PISA transformations. An idealized PISA target, with a single, arbitrary-length, match-action pipeline is assumed, and the CFG is transformed to a table graph (and actions) for it. Control flow is mapped to branching statements or MAT lookups. Window data is accessed through the packet part of the PHV, and intermediate values become metadata by a (kind of a) reverse SROA pass, mapping SSA registers to a metadata struct. The compiler may also make some high level decisions here, like moving part of the CFG (e.g., small trip count loops) to the packet parser.

Finally, we have arch-specific transformations. The CFG is mapped to the programmable blocks defined by a specific PISA architecture. Based on this, it is also decided if recirculation is required. Given chip-specific information, this stage may reject a program. For instance, the PHV size depends on the VLIW length, which may be too small for a given kernel.

Code generation. This stage transforms LLVM IR to P4. Dangeti et al. proposed a P4 to LLVM compiler to achieve better optimizations for P4 programs [13]. We plan to use this as a starting point for code generation and work backwards. In the final step, the generated P4 code is merged with a template switch configuration. For this part we aim to build on prior work [51, 62] that has shown the feasibility of such modularity. The final P4 program is given to a P4 backend to eventually accept/reject it. This is needed for two reasons: (a) chip constraints are not publicly available and (b) switch ISAs

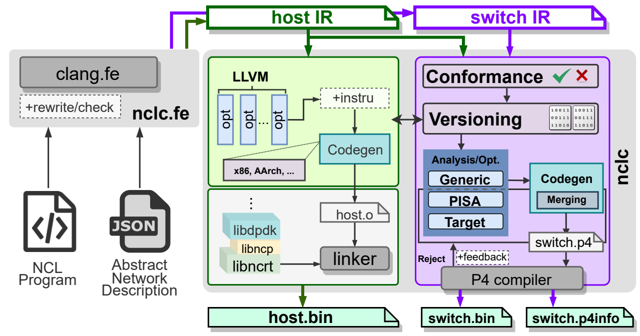


Figure 6: The NCL compilation trajectory

and driver specifications are also proprietary, preventing `nclc` from generating binaries directly.

6 CONCLUSION AND FUTURE WORK

While fundamentally addressing INC programming, we also identify some open problem and places for future work.

NCL requires a concrete concurrency and memory model. That is, memory access semantics in the presence of inter/intra-window parallelism. It is apparent that barrier-like operations do not fit this model, as ordering is hardware enforced. Thus, focus should be put on atomics and their extent. This requires deeper study of P4 semantics [11, §17.4.1] and the details of target platforms. The latter may not be as straightforward, given that currently, this information is not public.

Reliance on a P4 backend limits portability and leads to a potentially lengthy trial-and-error process until an NCL program is accepted. It also requires programmatically receiving feedback and addressing it, a challenging task on its own. For the long-term success of our system, and INC adoption in general, an *open* middle ground is required. An example is the NVVM IR [38]. A valid NVVM IR module is guaranteed to be compilable by NVIDIA's proprietary backend, allowing high-level languages to easily target GPU devices. Something similar might also be possible for P4 backends.

For an early prototype we aim for a smaller set of features: Sockets/UDP backend, one kernel-invoking API, and windows that fit a packet. We note that multi-packet windows pose significant challenges. Storing multiple packets may not yet be practical due to limited switch memory, or pipeline stages may be too few to access enough memory locations.

Future work could extend NCL to more platforms. For instance, ASIC limitations could be lifted by bump-in-the-wire accelerators [17], and incoming kernels could be offloaded to host-side accelerators (e.g., SmartNICs or DPUs). Windows could also be extended to handle more complex associations or multidimensional arrays. Finally, NCL would greatly benefit from external tools for network mapping, deployment, debugging, and testing of programs.

Acknowledgement. This research was supported by Dutch Research Council (NWO) grant OCENW.KLEIN.209.

REFERENCES

[1] Gustavo Alonso, Carsten Binnig, Ippokratis Pandis, Kenneth Salem, Jan Skrzypczak, Ryan Stutsman, Lasse Thostrup, Tianzheng Wang, Zeke Wang, and Tobias Ziegler. 2019. DPI: the data processing interface for modern networks. *CIDR 2019 Online Proceedings* (2019).

[2] Arista. 2021. 7170 Series - High Performance Multi-function Programmable Platforms. (2021). https://www.arista.com/en/products/7170-series

[3] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic In-Band Network Telemetry. 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 (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 662–680. https://doi.org/10.1145/3387514.3405894

[4] Marcel Blöcher, Lin Wang, Patrick Eugster, and Max Schmidt. 2021. 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 (ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 268–285. https://doi.org/10.1145/3445814.3446760

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[6] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 99–110. https://doi.org/10.1145/2486001.2486011

[7] Broadcom. 2021. NPL - Network Programming Language. (2021). https://nplang.org

[8] Broadcom. 2021. Trident4. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series. (2021).

[9] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. 2017. DRMT: Disaggregated Programmable Switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3098822.3098823

[10] Cisco. 2021. Cisco Silicon One. (2021). https://www.cisco.com/c/en/us/solutions/silicon-one.html

[11] The P4 Language Consortium. 2021. $P4_{16}$ Language Specification. (2021). https://p4.org/p4-spec/docs/P4-16-v1.2.2.html

[12] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. 2020. P4xos: Consensus as a Network Service. *IEEE/ACM Trans. Netw.* 28, 4 (Aug. 2020), 1726–1738. https://doi.org/10.1109/TNET.2020.2992106

[13] Tharun Kumar Dangeti, Venkata Keerthy S., and Ramakrishna Upadrasta. 2018. P4LLVM: An LLVM Based P4 Compiler. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. 424–429. https://doi.org/10.1109/ICNP.2018.00059

[14] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2014. The Road to SDN: An Intellectual History of Programmable Networks. *SIGCOMM Comput. Commun. Rev.* 44, 2 (April 2014), 87–98. https://doi.org/10.1145/2602204.2602219

[15] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. 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 (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 435–450. https://doi.org/10.1145/3387514.3405879

[16] Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. 2019. Autogenerating Fast Packet-Processing Code Using Program Synthesis. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets '19)*. Association for Computing Machinery, New York, NY, USA, 150–160. https://doi.org/10.1145/3365609.3365858

[17] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. 2021. In-network Aggregation for Shared Machine Learning Clusters. In *Proceedings of Machine Learning and Systems*, Vol. 3. 829–844.

[18] The P4.org Architecture Working Group. 2021. $P4_{16}$ Portable Switch Architecture (PSA). (2021). https://p4.org/p4-spec/docs/PSA.html

[19] Intel. 2021. Intel Tofino Series. (2021). https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html

[20] Intel. 2021. $P4_{16}$ Intel Tofilo Native Architecture - Public Version. (2021). https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch-Document.pdf

[21] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. 2018. Life in the Fast Lane: A Line-Rate Linear Road. In *Proceedings of the Symposium on SDN Research (SOSR '18)*. Association for Computing Machinery, New York, NY, USA, Article 10, 7 pages. https://doi.org/10.1145/3185467.3185494

[22] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. Netchain: Scale-Free Sub-RTT Coordination. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI'18)*. USENIX Association, USA, 35–49.

[23] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 121–136. https://doi.org/10.1145/3132747.3132764

[24] Khronos Group. 2021. OpenCL - Open Standard for Parallel Programming of Heterogeneous Systems. https://www.khronos.org/opencl/. (2021).

[25] Ike Kunze, René Glebke, Jan Scheiper, Matthias Bodenbenner, Robert H Schmitt, and Klaus Wehrle. 2021. Investigating the Applicability of In-Network Computing to Industrial Scenarios. *IEEE ICPS* (2021).

[26] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 741–761.

[27] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization Feedback-Directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, USA, 75.

[28] Alberto Lerner, Rana Hussein, and P. Cudré-Mauroux. 2019. The Case for Network Accelerated Query Processing. In *CIDR*.

[29] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, USA, 143–157.

[30] LLVM. 2021. Clang: a C language family frontend for LLVM. https://clang.llvm.org. (2021).

[31] LLVM. 2021. Compiling CUDA with clang. https://llvm.org/docs/CompileCudaWithLLVM.html. (2021).

[32] man7.org. 2021. udp(7) - Linux manual page. https://man7.org/linux/man-pages/man7/udp.7.html. (2021).

[33] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting Network Support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 313–326. https://doi.org/10.1145/3230543.3230557

[34] Netronome. 2021. Agilio CX SmartNICs. https://www.netronome.com/products/agilio-cx/. (2021).

[35] NoviFlow. 2021. NoviSwitch. (2021). https://noviflow.com/noviswitch

[36] NVIDIA. 2021. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. (2021).

[37] NVIDIA. 2021. NVIDIA Ethernet P4. (2021). https://developer.nvidia.com/networking/ethernet-p4

[38] NVIDIA. 2021. NVVM IR Specification 1.7. https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html. (2021).

[39] NVIDIA. 2021. The CUDA Compilation Trajectory. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#cuda-compilation-trajectory. (2021).

[40] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. 2018. Stateless Datacenter Load-Balancing with Beamer. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI'18)*. USENIX Association, USA, 125–139.

[41] Open Networking Foundation. 2021. ONOS - Open Network Operating System. https://opennetworking.org/onos/. (2021).

[42] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 16–29. https://doi.org/10.1145/3341301.3359642

[43] Pensando. 2020. Pensando DSC-25 Distributed Services Card. https://pensando.io/wp-content/uploads/2020/03/Pensando-DSC-25-Product-Brief.pdf. (2020).

[44] Dan R. K. Ports and Jacob Nelson. 2019. When Should The Network Be The Computer?. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 209–215. https://doi.org/10.1145/3317550.3321439

[45] DPDK Project. 2021. Data Plane Development Kit. https://www.dpdk.org/. (2021).

[46] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. 2018. Can the Network Be the AI Accelerator?. In *Proceedings of the 2018 Morning Workshop on In-Network Computing (NetCompute '18)*. Association for Computing Machinery, New York, NY, USA, 20–25. https://doi.org/10.1145/3229591.3229594

[47] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets-XVI)*. Association for Computing Machinery, New York, NY, USA, 150–156. https://doi.org/10.1145/3152434.3152461

[48] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 785–808.

[49] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/2934872.2934900

[50] Haoyu Song. 2013. Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*. Association for Computing Machinery, New York, NY, USA, 127–132. https://doi.org/10.1145/2491185.2491190

[51] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. 2020. Composing Dataplane Programs with uP4. 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 (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 329–343. https://doi.org/10.1145/3387514.3405872

[52] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, and Kunle Olukotun. 2020. Taurus: An intelligent data plane. *arXiv preprint arXiv:2002.08987* (2020).

[53] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. 2021. DFI: The Data Flow Interface for High-Speed Networks. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 1825–1837. https://doi.org/10.1145/3448016.3452816

[54] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. 2020. Cheetah: Accelerating Database Queries with Switch Pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2407–2422. https://doi.org/10.1145/3318464.3389698

[55] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. 2019. The Case For In-Network Computing On Demand. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 21, 16 pages. https://doi.org/10.1145/3302424.3303979

[56] Sébastien Vaucher, Niloofar Yazdani, Pascal Felber, Daniel E. Lucani, and Valerio Schiavoni. 2020. ZipLine: In-Network Compression at Line Speed. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '20)*. Association for Computing Machinery, New York, NY, USA, 399–405. https://doi.org/10.1145/3386367.3431302

[57] Dingming Wu, Ang Chen, T. S. Eugene Ng, Guohui Wang, and Haiyong Wang. 2019. Accelerated Service Chaining on a Single Switch ASIC. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets '19)*. Association for Computing Machinery, New York, NY, USA, 141–149. https://doi.org/10.1145/3365609.3365849

[58] Xilinx. 2021. Alveo U25 SmartNIC Accelerator Card. (2021). https://www.xilinx.com/products/boards-and-kits/alveo/u25.html

[59] Zhaoqi Xiong and Noa Zilberman. 2019. Do Switches Dream of Machine Learning? Toward In-Network Classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets '19)*. Association for Computing Machinery, New York, NY, USA, 25–33.

https://doi.org/10.1145/3365609.3365864

[60] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. 2020. 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 (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 126–138. https://doi.org/10.1145/3387514.3405857

[61] Lior Zeno, Dan R. K. Ports, Jacob Nelson, and Mark Silberstein. 2020. SwiShmem: Distributed Shared State Abstractions for Programmable Switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets '20)*. Association for Computing Machinery, New York, NY, USA, 160–167. https://doi.org/10.1145/3422604.3425946

[62] Peng Zheng, Theophilus Benson, and Chengchen Hu. 2018. 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 (CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 98–111. https://doi.org/10.1145/3281411.3281436