# Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs

Jiaqi Gao[†§], Ennan Zhai[†], Hongqiang Harry Liu[†], Rui Miao[†], Yu Zhou[†◇], Bingchuan Tian[†⋆], Chen Sun[†]
Dennis Cai[†], Ming Zhang[†], Minlan Yu[§]

[†]*Alibaba Group* [§]*Harvard University* [◇]*Tsinghua University* [⋆]*Nanjing University*

## ABSTRACT

Programmable data plane has been moving towards deployments in data centers as mainstream vendors of switching ASICs enable programmability in their newly launched products, such as Broadcom's Trident-4, Intel/Barefoot's Tofino, and Cisco's Silicon One. However, current data plane programs are written in low-level, chip-specific languages (*e.g.*, P4 and NPL) and thus tightly coupled to the chip-specific architecture. As a result, it is arduous and error-prone to develop, maintain, and composite data plane programs in production networks. This paper presents Lyra, the first cross-platform, high-level language & compiler system that aids the programmers in programming data planes efficiently. Lyra offers a *one-big-pipeline* abstraction that allows programmers to use simple statements to express their intent, without laboriously taking care of the details in hardware; Lyra also proposes a set of synthesis and optimization techniques to automatically compile this "big-pipeline" program into multiple pieces of runnable chip-specific code that can be launched directly on the individual programmable switches of the target network. We built and evaluated Lyra. Lyra not only generates runnable real-world programs (in both P4 and NPL), but also uses up to 87.5% fewer hardware resources and up to 78% fewer lines of code than human-written programs.

## CCS CONCEPTS

• **Networks** → **Programmable networks**; **Programming interfaces**; • **Theory of computation** → **Abstraction**;

## KEYWORDS

Programmable switching ASIC; Programmable Networks; Programming Language; Compiler; P4 Synthesis

**ACM Reference Format:**

Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, Minlan Yu . 2020. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, NY, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3387514.3405879


*SIGCOMM '20, August 10–14, 2020, Virtual Event, NY, USA*

ACM ISBN 978-1-4503-7955-7/20/08…$15.00
https://doi.org/10.1145/3387514.3405879

## 1 INTRODUCTION

> "*C language lets you get close to the machine, without getting tied up in the machine.*"
>
> — *Dr. Brian Kernighan*

Programmable network devices have gained significant traction in the networking community, as a result of their powerful capability allowing network programmers to customize the algorithms directly in the data plane and thus operate packets at the line rate. People have shown the tremendous benefits brought by the flexibility of programmable network devices [38], *e.g.*, load balancing [12, 27, 32], network monitoring [2, 22, 35], consistency algorithms [24, 28], in-network caching [25] and congestion control [29]. Currently, a growing number of programmable switching ASICs (application-specific integrated circuits) are being commercialized by mainstream chip vendors. For example, Broadcom launched Trident-4 and Jericho-2 which are programmable by NPL [1], whereas Intel/Barefoot's Tofino [6] and Cisco's Silicon One [4] support P4 programming [15].

Despite the bloom of programmable network devices and programming languages, the foundation of network programming on data plane is still at an early stage—network programmers are still using chip-specific languages and manually take care of numerous details with hardware features, hardware capacities, and network environments when developing data plane algorithms, comparably similar to the era when software engineers use assembly languages to write software on CPUs (central processing units). As a result, the manageability of data plane programs is still unready for large scale deployments and operations.

Specifically, there are three major problems faced by network programmers nowadays with chip-specific languages.

**Portability.** First of all, current data plane programs have poor portability because they are tightly coupled with specific ASIC models from specific vendors. For instance, even for the same vendor, a program running on Barefoot Tofino 32Q does not necessarily run automatically on Tofino 64Q due to the varying numbers of march-action units and different memory resources; not to mention the migration from Barefoot Tofino to Broadcom Trident-4 which has totally different pipeline design and chip-specific language. Therefore, network programmers are required to be not only proficient in all the languages involved, but also knowledgeable about the various pipeline architectures and resource constraints of the different programmable ASICs.

**Extensibility.** Second, low-level languages focus on programming individual ASICs, while there are data plane programs that require to execute on multiple ASICs in a distributed way. For example, INT

(in-band network telemetry) [2] has different roles for ingress, transit, and egress switches; middle-boxes, *e.g.*, load balancer (LB) [32], can also collectively use table resources in multiple switches for accommodating large-scale workloads. However, nowadays, network programmers have to individually program each switch's data plane with its own low-level chip language because a high-level, network-wide abstraction for the data plane programming does not currently exist yet.

**Composition.** Last but not least, a practical deployment of programmable data plane must have multiple programs enabled. For instance, a data center network might want both INT, LB, and scheduler co-existed in the data plane. One particular combination of programs can lead to a complete restructure of each individual program and their deployment arrangements because of the considerations on the details of switch capability, network topology, and so forth. The whole process is arduous and error-prone.

We believe the fundamental reason for the above problems in the state-of-the-art data plane programming is the lack of a high-level language. In this paper, we present Lyra—a language & compiler system for programmable data center networks (DCNs)—that facilitates data plane programs to achieve portability, extensibility, and composition *simultaneously*.[1] Lyra language offers a *one-big-pipeline* abstraction to network programmers, and the latter can flexibly express the logic of their programs in a chip-neutral and target-agnostic way; Lyra compiler compiles the Lyra program into multiple pieces of runnable chip-specific code that can be launched directly on the programmable switches of the target network, eliminating the need for engineer proficiency in any chip-specific architectures and languages involved.

**Lyra language.** Different from the existing high-level abstractions on control plane programming that focuses on packet forwarding [13, 14, 19, 31, 34], Lyra's goal is to provide a high-level abstraction for data plane programming to express packet processing logics, such as packet header write and arithmetic operations.

Lyra language offers a *one-big-pipeline* programming abstraction that is simple and expressive for directly describing how packets with different characters will be processed along a chain of algorithms. Each algorithm is a tree-like procedure that defines the packet processing logics with *if-else* statements and simple read, write and arithmetic operations to packets. With this language, network programmers can directly express packet process procedures without worrying about how the underneath switches realize the logics, *e.g.*, using multiple tables to implement an *if-else* statement or using one table to implement multiple *if-else* statements.

Lyra language also offers a critical ability to specify an *algorithm scope* that explicitly defines the scope of candidate switches an algorithm to be deployed into. For example, network programmers may wish to deploy a stateful load balancer merely on ToR (top-of-rack) switches. This feature provides an essential ability for programmers to guide the final compilation and deployments with high-level intents.

**Lyra compiler.** The core task of the Lyra compiler is to combine the high-level Lyra program, algorithm scopes, network topology, and the low-level details of ASICs to generate correct and runnable chip-specific code in the target network.

Different from prior works [14, 26, 41] that focus on resource allocations with integer linear programming (ILP), Lyra faces more complex scenarios due to conditional feature constraints, which cannot be encoded with ILP, under the heterogeneity of ASICs. For instance, if the address resolution protocol learning function is deployed on an NPL/Trident-4 switch, we only need one table for lookup, but the P4/Tofino switch requires more than two tables.

The key methodology of the Lyra compiler is to encode all logics and constraints into an SMT (satisfiability modulo theories) problem and use an SMT solver to find the best implementation and deployment strategy of a given Lyra program in the target network. Lyra takes three steps to achieve this goal. First, Lyra translates the Lyra program into a context-aware intermediate representation (or context-aware IR), with important context information such as instruction dependency and deployment constraints. Second, Lyra synthesizes *conditional language-specific implementations* for each algorithm based on its context-aware IR. Lyra puts the synthesized conditional language-specific implementations into the corresponding switches, and uses a logical formula to restrict that there will be only one implementation exist of each algorithm in the final solution. We design effective algorithms to solve the major challenge at this step, which is the generation of language-specific tables and their actions based on the dependencies of statements written in Lyra language (§5.2 and §5.3). Finally, Lyra constructs an SMT formula that encodes all resource and placement constraints to decide the chip-specific implementation and placement of all algorithms simultaneously. If an algorithm cannot be placed into a single switch due to a lack of enough resources, Lyra can split it into smaller ones and put them into multiple switches. The major challenge here is to understand the resource allocation behaviors of different ASICs and encode them into the SMT formula (§5.4-§5.5).

**Evaluation.** We have built Lyra and evaluated its effectiveness on a variety of real-world programs. Lyra not only generated runnable real-world programs (in both P4 and NPL), but also used up to 87.5% fewer hardware resources and up to 78% fewer lines of code than human-written programs.

## 2 OVERVIEW

As the major switch vendors, *e.g.*, Broadcom, Cisco, Intel/Barefoot, *etc.*, embrace programmable data plane with their new mainstream ASIC products for DCNs, the revolution towards programmable DCNs has already started.

However, despite that programmability on data plane offers programmers tremendous opportunities to customize network features or offload computations to networks, one crucial requirement to deploy and operate a programmable DCN is how to maintain the manageability at least on the same level as current DCNs. Without meeting this need, the adoption of programmable DCN would significantly slow down or even never happen in the worst case.

As one of the largest global service providers, Alibaba is already focusing on the challenge to develop, maintain, and composite data plane programs in realistic DCNs with heterogeneous ASICs. In fact, DCNs are always heterogeneous in switch vendors and ASIC types for two reasons. First, network operators need to prevent the "vendor lock-in" problem [30, 36], so they intentionally use

[1] We focus on programmable DCNs in this work, but we believe Lyra is easily extendable to more scenarios such as programmable WANs.

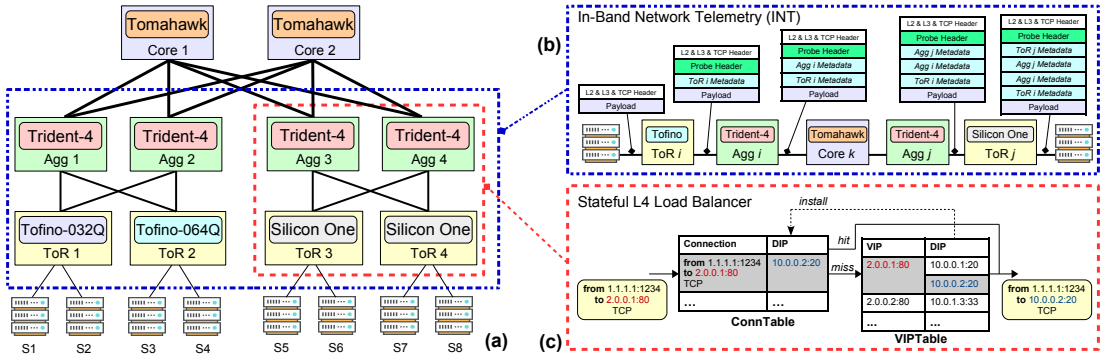


**Figure 1: Motivating example. The network programmers deploy INT across the entire network, and stateful load balancer on Agg 3, Agg 4, ToR 3 and ToR 4.**

different vendors in their networks and require the equipment from these vendors to be replaced transparently to their management plane and applications. Second, different ASICs have distinctive trade-offs among programmability, throughput, buffer size, and cost, due to the physical limitations in chip manufacture. Different layers of DCNs, therefore, adopt different types of ASICs. For example, ToR switches may use high-programmability ASICs (*e.g.*, Barefoot Tofino and Broadcom Trident-4) for near-server computation offloading, while core switches employ high-throughput but less programmable ASICs, *e.g.*, Broadcom Tomahawk.

## 2.1 Motivation

Similar to control plane software, data plane programs also need to be continuously upgraded for bugs fixing or introductions of new features; Different data plane programs still have to co-exist inside one DCN, and each program should be added or deleted as well. Nonetheless, the current practice of data plane programming with chip-specific languages can hardly achieve the above requirements, especially under the heterogeneity of ASICs. Concretely, there are critical problems resulting from low-level programming languages, as we will explain with a simplified but realistic example.

Figure 1 shows an example for a programmable DCN that has five types of ASICs (ToR and Agg layers are fully programmable) and two data plane programs.

*(i) INT [2]:* INT was originally proposed to collect and report network state by inserting the critical metadata in the packet header. As shown in Figure 1(b), given a packet $p$, each programmable switch $k$ on the path inserts a metadata to $p$'s header by computing $eg_{k(p)} - ing_{k(p)}$, where $ing_{k(p)}$ and $eg_{k(p)}$ denote the ingress time stamp and egress time stamp of $p$ on $k$, respectively. In particular, INT contains three algorithms: ingress INT, transit INT, and egress INT. Ingress INT identifies the packets of interest, and inserts a probe header and the metadata (see $\text{ToR}_i$ in Figure 1(b)). Transit INT only inserts the metadata. Egress INT inserts the metadata, and mirrors the received packet for post analyzing. In our example, network programmers are required to deploy ingress and egress INT on ToR switches, and deploy transit INT on aggregation switches.

*(ii) Stateful L4 load balancer (LB) [32]:* The L4 LB maps the packets destined to a service with a virtual IP (or VIP), to a set of servers holding the service with multiple destination IPs (or DIPs). In Figure 1 example, network programmers are required to deploy the LB in the scope {Agg 3, Agg 4, ToR 3, and ToR 4} to balance the traffic from core switches to servers S5-S8. A stateful L4 LB has two tables, VIPTable and ConnTable, as shown in Figure 1(c). For a given connection's packet $c$, if $c$'s VIP hits one of the items in ConnTable, $c$ is directly forwarded to the corresponding DIP; otherwise, the LB identifies the DIP pool based on $c$'s VIP in the VIPTable, and installs this ⟨VIP, DIP⟩ pair to ConnTable. For example, in Figure 1(c), all the subsequent packets of the connection matching ⟨1.1.1.1:1234, 2.0.0.1:80, TCP⟩ in ConnTable get forwarded to 10.0.0.2:20.

```
action enable_int() {
    enable_int = 1;
}

table check_src_ip {
    reads {ipv4.src_ip:exact;}
    actions {enable_int;}
    size: 1024;
}

table check_dst_ip {
    reads {ipv4.dst_ip:exact;}
    actions {enable_int;}
    size: 1024;
}

control int {
    apply(check_src_ip);
    apply(check_dst_ip);
} // P4
```

```
logical_table check_ip {
    table_type:hash;
    min_size:1024;
    max_size:1024;
    keys { bit[32] ip; }
    key_construct() {
        if(_LOOKUP0)
            ip = ipv4.src_ip;
        if(_LOOKUP1)
            ip = ipv4.dst_ip;
    }
    fields_assign() {
        enable_int = 1;
    }
}

program int {
    check_ip.lookup(0);
    check_ip.lookup(1);
} // NPL
```

**Figure 2: Flow filter: P4 V.S. NPL.**

If network programmers develop, deploy and maintain the above two data plane programs with P4 (on Tofino and Silicon One) and NPL (on Trident-4), three problems stem from the complexity in both the languages and ASIC architecture.

**Problem 1: Portability.** It is hard to migrate a low-level program from one ASIC to another. In Figure 1, initially, the network programmers develop ingress and egress INT programs in P4 on ToR switches. Despite that all ToR switches support P4, network programmers have to develop INT programs for each ASIC because Tofino-032Q, Tofino-064Q, and Silicon One have quite different pipeline architectures and resource constraints. For example, Tofino-064Q and Tofino-032Q have 12 and 24 match-action units (MAUs) [7] and different memory sizes respectively, causing Tofino-032Q's INT program in P4 that uses 18 MAUs compiles unsuccessfully on Tofino-064Q, let alone on Cisco's Silicon One; so, per model tuning is a must. Even worse, in the aggregation layer, the programmers rewrite the programs in NPL, because P4

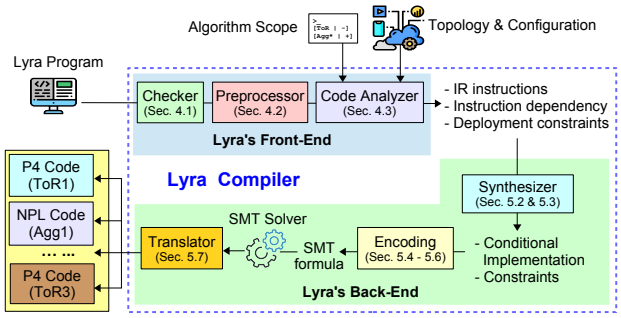


**Figure 3: Lyra workflow overview.**

and NPL have different language features and ASIC architectures. For example, Figure 2 shows the clear difference between the two languages in implementing the flow filter function of INT—P4 has to use two tables for matching both source and destination IPs, while NPL uses one table with two lookups.

**Problem 2: Extensibility.** Low-level languages focus on how to program individual ASICs, but a program is usually required to run on top of multiple ASICs in a distributed setting. In Figure 1 example, the programmers now need to deploy the stateful LB program on Agg 3, Agg 4, ToR 3, and ToR 4. At the beginning, they only need to write an NPL program implementing ConnTable, $T_c$, and VIPTable, $T_v$, on both Agg 3 and Agg 4. As the number of traffic connections increases, the programmers expand the size of $T_c$ by modifying the NPL program. However, the new NPL program compiles unsuccessfully because the total size of $T_v$ and the expanded ConnTable $T_c'$ exceeds the resource constraints of Trident-4 ASIC. The programmers, therefore, decide to move the VIPTable from Agg 3 and Agg 4 to ToR 3 and ToR 4 by writing another P4 program for $T_v$. It takes many hours for the programmers to make sure: (1) the P4 program compiles well on Silicon One ASICs, and (2) ConnTable and VIPTable can work together across switches. As the number of connections continues to grow, the programmers expand $T_c'$ again to get a bigger ConnTable, $T_c''$, making $T_c''$ no longer fit in a Trident-4 ASIC. In this tough case, the programmers have to *carefully* split $T_c''$ into $T_{c1}''$ and $T_{c2}''$, and make sure $T_{c1}''$ and $T_{c2}''+T_v$ compilable on the corresponding ASICs, while coordinating correctly. Obviously, the programmers spend a lot of effort and time in the above depressing process.

**Problem 3: Composition.** It is non-trivial to make multiple low-level programs co-exist well in a DCN. For example, in Figure 1, any particular combination of INT and LB programs may result in a complete restructure of each program and its deployment arrangements. For example, once the programmers move too many entries of ConnTable to the ToR switches, the program may not compile successfully because of not only VIPTable but also ingress and egress INT programs. As the number of deployed programs increases, it would be much harder to find a "fittable" deployment.

**Summary.** The problems of portability, extensibility, and composition fundamentally undermine the manageability of programmable DCNs, since network programmers will get trapped into endless program reconstructions and numerous hardware details in daily operations. The root cause of this dilemma is the direct use of low-level, chip-specific programming languages, since a high-level, cross-platform, network-wide programming language is missing at present. This is our fundamental motivation to build Lyra.

```
>HEADER:
  header_type int_probe_hdr_t { // Define header type
      bit[8] hop_count;
      ...
  }
  packet in_pkt { fields { ... } }

>PIPELINES:
  pipeline[INT]{int_in -> int_transit -> int_out};
  pipeline[LB]{loadbalancer};

  algorithm loadbalancer { // Define load balancer
      load_balancing();
  }
  algorithm int_in { // Ingress INT
      global bit[32][1024] packet_counter; // global variable
      int_filtering();
      if (int_enable) {
          add_int_probe_header();
          add_int_md_hdr();
      }
  }
  algorithm int_transit { ... } // Transit INT
  algorithm int_out { ... } // Engress INT

>FUNCTIONS:
  func add_int_md_hdr() {
      extern dict<bit[8] msg_type, bit[30] switch_id>[128]
          ↪ add_int_md_hdr_filter;
      if (int_probe_hdr.msg_type in add_int_md_hdr_filter) {
          add_header(int_md_hdr);
          int_md.queue_len = get_queue_len();
          int_info(int_info);
          ...
      }
  }
  func load_balancing() {
      extern dict<bit[32] hash,bit[32] ip>[1024] conn_table;
      extern dict<bit[32] vip,bit[8] group>[1024] vip_table;
      hash = crc32_hash(ipv4.srcAddr, ipv4.dstAddr, ipv4.
          ↪ protocol, tcp.srcPort, tcp.dstPort);
      if (hash in conn_table) {
          ipv4.dstAddr = conn_table[hash];
      }
      ...
  }
  ...
```

**Figure 4: Lyra program for our motivating example.**

## 2.2 Overview of Lyra Program & Workflow

Lyra enables programmers to efficiently program data plane. For example, the network programmers can write a Lyra program shown in Figure 4 for the case in Figure 1. By taking in this program, Lyra compiler generates eight pieces of chip-specific code that compile successfully on Agg 1-4 and ToR 1-4, while meeting the functional correctness specified by the input Lyra program.

**Lyra's workflow.** Figure 3 presents Lyra's workflow. First, Lyra takes as input: (1) a high-level Lyra program, (2) an algorithm scope describing each algorithm's placement, and (3) DCN topology and configurations. Then, Lyra's front-end generates a context-aware intermediate representation (or context-aware IR), with important information such as instruction dependency and deployment constraints. Finally, Lyra's back-end uses the context-aware IR to synthesize conditional implementations for different languages (*e.g.*, P4 and NPL), and encodes various constraints in the form of SMT formula. We solve the formula to get a solution that can be translated into multiple pieces of chip-specific code.

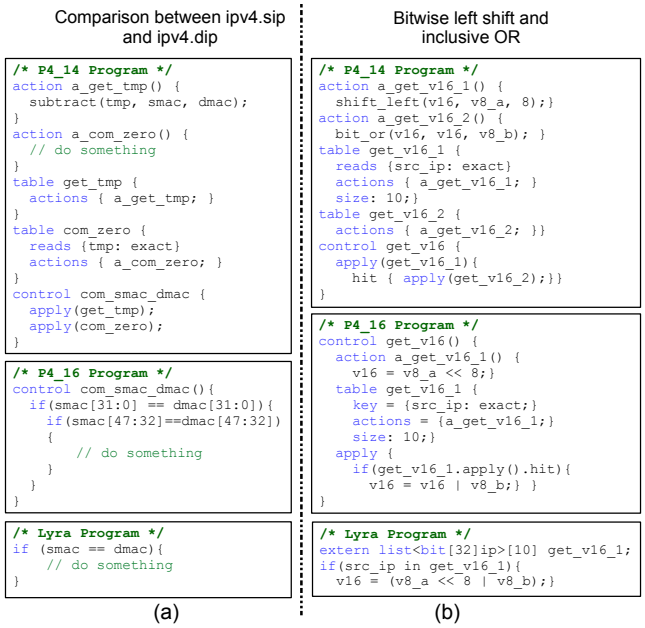


**Figure 5: Lyra program V.S. P4, chip-specific program.**

## 3 LYRA LANGUAGE

Lyra introduces a high-level abstraction for the network programmers to express their algorithms without the hassle of low-level details. Figure 6 shows the grammar.

### 3.1 Why Lyra is High-Level?

Compared with current chip-specific languages (*e.g.*, P4 and NPL), Lyra's abstraction is easier to use for the following two reasons. First, Lyra programming only relies on simple semantics (*e.g.*, if-else) to express packet processing logic rather than "mandatory" built-in data structures such as tables and registers in P4 and NPL. In other words, using Lyra, the programmer does not need to take into account how many tables they need to create, what functions should be put in which tables, or how to assign registers to different stateful variables. Second, Lyra is an architecture-independent language, which allows the programmer to program without considering chip-specific resource limitation (*e.g.*, how many bits each stage can support) or architecture constraints (*e.g.*, how many shared register between stages and can the same stages be accessed multiple times or just once). In some sense, the relationship between Lyra and chip-specific languages can be compared to the relationship between C language and processor-specific assembly languages.

Figure 5 shows two examples to illustrate the difference between Lyra and P4. In Figure 5(a), the programmer wants to check whether the source MAC address, `smac`, is equal to the destination MAC address, `dmac`. While P4 language itself does not limit the maximum bit width in such a comparison, some of programmable ASICs, say ASIC-X, cannot support the comparison of longer-than-44-bit variables.[2] In $P4_{14}$, the programmer has to address this restriction by creating two additional tables: one for `subtract(tmp, smac, dmac)` and another one for checking `tmp` is zero or not; in $P4_{16}$, the programmer needs to reduce the original 48-bit variable comparison to two 32-bit variable comparisons; on the contrary, the programmer can use Lyra to directly write `if(smac == dmac)` rather than handling the above low-level details in person, and then Lyra compiler can automatically generate P4 code according to the underlying ASICs' restrictions. In Figure 5(b), the programmer implements a set of simple bitwise operations by introducing multiple actions and tables in both $P4_{14}$ and $P4_{16}$ for assignment, bitwise left shift and bitwise inclusive OR, due to the sequential read-write dependency of ASIC-X. Using Lyra, on the other hand, the programmer only needs to write: `v16 = (v8_a « 8 | v8_b)`.

The above examples are similar to the memory allocation situations in C and assembly languages. If using assembly languages (*e.g.*, ARM and x86), we pay attention not only to the low-level instructions (*e.g.*, MOV and PUSH), but also to the usage of register and memory; on the contrary, we can use higher-level C language to express memory allocation by just writing a `malloc`.

### 3.2 Lyra's Programmable Model

Lyra introduces a new programming model named *one big pipeline*, or OBP. This programming model treats each data plane program involving multiple algorithms as a single pipeline covering these algorithms. OBP aims to avoid low-level details such as table-oriented grammar (like P4). In Figure 1, INT is an OBP consisting of three algorithms: ingress INT, transit INT, and egress INT, and stateful L4 LB is another OBP. We can implement these two OBPs in a Lyra program that *must* consist of three parts: (1) pipeline specification, and (2) function, and (3) header definition.

**Pipelines & algorithm definition (Line 8 in Figure 4).** The OBP allows the programmers to treat what they want to deploy as a single pipeline that contains one or more algorithms. We use `pipeline` to define an OBP, and use `algorithm` to specify each algorithm in the OBP. In our motivating example, as an OBP, INT has three algorithms: (1) `int_in` (defined in Lines 16-22 in Figure 4), (2) `int_transit` (Line 23), and (3) `int_out` (Line 24), corresponding to ingress, transit, and egress INT, respectively. On the other hand, the stateful LB is another OBP which only has one algorithm, defined in Lines 13-15 in Figure 4. In Lyra, we recursively specify all the algorithms in an OBP. In Figure 4 example, we define these two OBPs in Lines 9-10, respectively. Using the OBP abstraction, the programmers only need to focus on what algorithms should be involved in an OBP.

**Function definition.** An algorithm (*e.g.*, `int_in`) may contain multiple functions. In Lyra, the definition of each function is similar to the C language. In Figure 4, Lines 36-43 define the only function for the LB algorithm. Lyra also offers many predefined library-function calls that commonly exist in the state-of-the-art chip-specific languages. For example, both NPL and P4 have functions that extract the queue length, so that Lyra offers a predefined library-function call `get_queue_len()`, as shown in Line 31 in Figure 4.

**Header definition.** The programmers should specify the packet header and parser for each deployed algorithm. This part is similar to the header and parser definitions in P4. In Lyra, we use **header_type** and **parser_node** to define the header type and parser, respectively.

[2] We use examples from real ASICs, but omit the ASIC names.

| | | | |
|---|---|---|---|
| *prog* | ::= | *declaration** | Lyra program |
| *declaration* | ::= | *header* | Header type |
| | \| | *bigPipe* | One big pipeline |
| | \| | *alg* | Algorithm |
| | \| | *func* | Function |
| | \| | ... | |
| *cmd* | ::= | *if* \| *func* \| ... | Commands |
| *str* | ::= | *string* | String |
| *header* | ::= | **header_type** *str* {*hBody*} | Header type |
| *hBody* | ::= | **fields** {*hFields*} | Header body |
| *hFields* | ::= | *type str* | Header field |
| ... | | ... | |
| *bigPipe* | ::= | **pipeline** [*str*] {*bpB*} ; | One big pipeline |
| *bpB* | ::= | *str* ( -> *str* )* | A list of algorithms |
| *alg* | ::= | **algorithm** *str* {*pBody*} | Network algorithm |
| *pBody* | ::= | *cmd** | Functions |
| *func* | ::= | **func** *str*(*para* ) {*funBody*} | Define a function |
| *para* | ::= | *type str* ( , *type str* )* | Parameters |
| *funBody* | ::= | *item** | Table items |
| | \| | *cmd** | Commands |
| *item* | ::= | **extern** *tableType* | External variables |
| ... | | ... | |

**Figure 6: Lyra language grammar.**

## 3.3 Specifying Algorithm Scope

Lyra allows the programmers to specify the fine-grained scope for each algorithm in a given pipeline. The algorithm scope is designed for extensibility and composition. Note that specifying such a scope should be the main job of network operators rather than programmers, so that Lyra allows either programmers or operators (or both) to define each algorithm's scope. Due to different business needs and deployments, we should use the scope to "tailor" the underlying data plane in a specific way for each of DCNs.

The algorithm scope can be specified as:

```
algorithm_name: [ region | deploy | direct ]
```

**Region.** For an algorithm $A$, we use `region` to specify a set of switches for $A$'s potential placement, *e.g.*, all ToR switches or a single switch Agg 3.

**Deploy.** The programmers may want to deploy the copies of $A$ on multiple switches. In Figure 1 example, the copies of `int_in` are deployed on the four ToR switches, respectively. On the other hand, the programmers may use multiple switches to realize one single algorithm. For example, `loadbalancer` in Figure 1 is deployed on four switches. The programmers can distinguish the above two cases by specifying `deploy` field in the algorithm scope specification. The value of the `deploy` field is either PER-SW or MULTI-SW. PER-SW means copying the algorithm on each of the specified switches, and MULTI-SW means realizing the algorithm across the specified switches. In Figure 7, we use PER-SW for INT's three algorithms. ToR* and Agg* denote all ToR and Agg switches.

**Direct.** When an algorithm is deployed on a set of switches, we need to specify the direction of the packet flow via the `direct` field. As shown in Figure 7, because the algorithm `loadbalancer` specifies MULTI-SW, we should define the packet flow direction via `direct`; thus, `direct` is `(Agg3,Agg4->ToR3,ToR4)`, which means the load balancer algorithm needs to handle the packet flow entering Agg 3 and Agg 4 and leaving from ToR 3 and ToR 4. This information is critical for the compiler because it restricts the possible paths the packet could take, so that the compiler can decide where to deploy the program. For example, in Figure 7, a packet traverses the load balancer could never take a path from ToR 4 to Agg 4.

```
int_in: [ ToR* | PER-SW | - ]
int_transit: [ Agg* | PER-SW | - ]
int_out: [ ToR* | PER-SW | - ]
loadbalancer:
 [ ToR3,ToR4,Agg3,Agg4 | MULTI-SW | (Agg3,Agg4->ToR3,ToR4) ]
```

**Figure 7: An example for algorithm scope example.**

## 3.4 Variables in Lyra Programs

Lyra defines three types of variables: internal variable, global variable, and external variable.

**Internal variable.** The internal variable is straightforward. It is created when a packet comes in the pipeline and destroyed when the packet leaves. Internal variable is fixed-width and single-element. For example, `bit[8]` in Line 3 in Figure 4 is an internal variable.

**Global variable.** The global variable provides an index-based array interface. Different from internal variables, global variables keep the information across packets. They are created when the Lyra program is burnt into the programmable switching ASIC, and last until the switch is down or the program is replaced by another. For example, `global bit[32][1024] pkt_counter` in Line 11 in Figure 4 defines a global variable, `pkt_counter`, which has 1024 elements, and each element is 32 bit wide. The global variable supports read and write on the data plane.

**External variable.** The external variable exposes an "table interface" bridging the data plane and control plane. To define an external variable, we need to define its type, input type, output type, and element number, such as `extern list<bit[32] ip>[1024] known_ip`. The external variable also allows the value (both input and output) to be a tuple:

```
extern dict<bit[32] src,bit[32] dst>,bit[8] p>[1024] route
```

We discuss how to translate the external variables into the tables exposed to the control plane in §5.8.

# 4 LYRA'S FRONT-END

The front-end of Lyra takes in a Lyra program and outputs a context-aware intermediate representation (or context-aware IR). Lyra's front-end, as shown in Figure 3, has three key modules: (1) given a Lyra program $\mathcal{P}$, **checker** checks the syntax and semantics of $\mathcal{P}$ (§4.1); (2) **Preprocessor** enriches and optimizes $\mathcal{P}$ to generate an IR (§4.2); and (3) **Code analyzer** (§4.3) analyzes IR's context information (*e.g.*, instruction dependency and deployment constraints) to form a context-aware IR for post synthesizing.

## 4.1 Checker

Similar to any compiler, Lyra uses a checker for the syntax and semantic correctness checking. Suppose a Lyra program, $\mathcal{P}$, is input. We check $\mathcal{P}$ with grammar defined in §3.

## 4.2 Preprocessor

The preprocessor translates $\mathcal{P}$ (*e.g.*, Figure 8(a)) into an IR (*e.g.*, Figure 8(c)). IR is crucial for post synthesizing, because high-level Lyra program $\mathcal{P}$ hides too many details, which makes it hard to directly synthesize chip-specific code from $\mathcal{P}$. In this section, we use Figure 8 to illustrate how preprocessor generates the IR.

**Step 1: Function inlining.** We iterate all the algorithms in the Lyra program. In each algorithm, we inline all the functions with their function bodies. For example, we expand `int_info(int_info)`

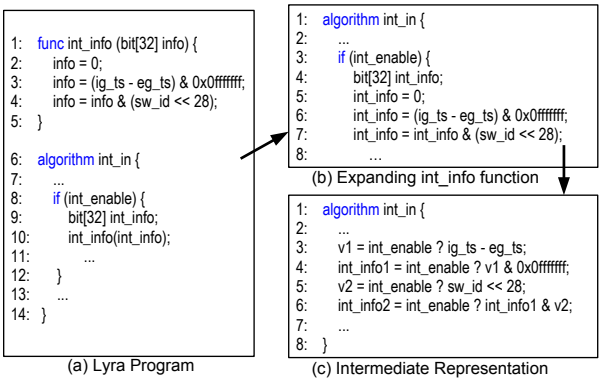


**Figure 8: Example. (a) is a Lyra program, (b) is the result of function expansion, and (c) is the generated IR.**

(Line 10 in Figure 8(a)) with its body (Line 2-4 in Figure 8(a)), obtaining Line 5-7 shown in Figure 8(b).

**Step 2: Branch removal.** A Lyra program may contain conditional statements, *e.g.*, Lines 8-12 in Figure 8(a), which complicate dependency analysis [37]. We, therefore, convert each **if-else** condition into a predicate, and then apply this predicate to all the instructions in the condition body. For example, the if condition `int_enable`, in Line 3 in Figure 8(b), is converted into a predicate, `int_enable ? ...`, and is applied to all instructions in this condition body, thus getting Lines 3-6 in Figure 8(c). Once all the branches are removed, the body of the algorithm becomes a straight-line code block.

**Step 3: Single operator tuning.** We expand the instructions that have more than one operator. For example, Line 6 in Figure 8(b) is flattened into Lines 3 and 4 in Figure 8(c).

**Step 4: Static single assignment (SSA) form conversion.** SSA assigns each variable a *version* field. When the variable is assigned to a new value, the version increases accordingly. SSA guarantees no versioned variable is assigned twice and removes the Write-After-Read and Write-After-Write dependencies. After this step, only Read-After-Write dependency remains. The `int_info1` and `int_info2` (*i.e.*, Lines 4 and 6 in Figure 8(c)), for example, are assigned to different versions.

**Step 5: Variable type inference.** The width of program variables is inferred based on 3 rules: (1) **function call**, such as `crc32_hash` returns a 32-bit variable; (2) **operation**, *and* operation generates a 1-bit variable; and (3) **variable lookup**, the input/output type of the table are defined explicitly. For example, in Figure 8(c), the `v1` is inferred as a 32-bit variable as the `ig_ts` and `eg_ts` are 32 bits.

## 4.3 Code Analyzer

So far, the preprocessor has translated a Lyra program $\mathcal{P}$ to an IR. Nevertheless, this IR is a plain-text IR, which lacks context information (*e.g.*, instruction dependency, and deployment constraints) for chip code synthesizing (§5). We, therefore, build a code analyzer to add "context" to the IR.

**Instruction dependency generation.** We first analyze the dependencies among IR instructions to generate an instruction dependency graph. This is important, because it would determine the execution order and placement of these instructions in the chip-specific code synthesizing. For example, if instruction $b$ relies on another instruction $a$, $b$ should be placed in the stage behind $a$'s stage; if there is no dependency between $a$ and $b$, we can parallelize their executions in different ALUs even in the same stage.

Since IR has been a straight-line code with only read-after-write dependency, it is straightforward to build an IR instruction dependency graph, where each node represents an IR instruction, and a directed edge from node $a$ to node $b$ means the instruction $b$ reads one or more variables written by instruction $a$. For example in Figure 8(c), there are three dependencies: (1) Line 3 → Line 4; (2) Line 4 → Line 6; and (3) Line 5 → Line 6.

**Deployment constraints generation.** Given the network topology information and algorithm scope specification, as shown in Figure 3, we can generate the following data: (1) target network topology with the algorithm-scope tags, such as the Agg 4 in Figure 1 is tagged with algorithms *int_transit* and *loadbalancer*; and (2) potential flow paths in each scope, such as in the Load Balancer scope there are four possible flow paths: *Agg3* → *ToR3*, *Agg3* → *ToR4*, *Agg4* → *ToR3*, and *Agg4* → *ToR4*.

# 5 LYRA'S BACK-END

By taking in the context-aware IR, Lyra's back-end synthesizes chip-specific code. Specifically, this section first models our problem (§5.1). Then, we describe how to synthesize conditional implementations for P4 (§5.2) and NPL (§5.3). Next, we use the public RMT architecture [16, 26] as an example to illustrate how to encode chip-specific constraints for the portability (§5.4). We further present how to encode deployment constraints for the composition (§5.5) and resource extensibility (§5.6). We put all the above encoded constraints in the set of conditional placement constraints, and call an SMT solver to solve the formulas, obtaining a solution which can be translated into chip-specific code (§5.7). Finally, we present the control plane interfaces exposed by Lyra in §5.8.

## 5.1 Problem Modeling

A Lyra program contains a list of algorithms $\mathcal{G}$. For an algorithm $a \in \mathcal{G}$, it has a specified algorithm scope, $\mathcal{S}_a$, *e.g.*, Figure 7, which represents a group of switches. §4 describes how does the front-end transform $a$ to a collection of IR instructions, defined as $I_a$. We use $I_{a(j)}$ to denote the $j$th IR instruction in $I_a$. We define $f_s(I_{a(j)})$ as a boolean function, which indicates whether the IR instruction $I_{a(j)}$ should be deployed on the switch $s \in \mathcal{S}_a$. The goal of this section is to find a feasible combination of $f_s$ that meets the constraints in the target network. Note that an IR instruction can be deployed on multiple switches, as long as the correctness of the program holds.

## 5.2 Conditional P4 Synthesis

This section synthesizes conditional P4 implementation based on the context-aware IR. Intuitively, conditional P4 synthesis aims to map the instructions in IR representation to tables in P4 by analyzing the dependencies among those IR instructions.

Before describing the details of the synthesis algorithm, we need to define several important terminologies.

- *Potentially deployed IR.* We define $\mathcal{R}_s$ as a set containing all IR instructions potentially deployed on switch $s$. We learn $\mathcal{R}_s$ because we know the scope of the algorithm any IR instruction belongs to.
- *Predicate blocks.* A predicate block is a set containing IR instructions, where (1) IR instructions have the same predicate, and (2)

IR instructions have no dependency. For example, in Figure 8(c), Line 4 and Line 5 should be put in the same predicate block because (1) they have the same predicate `int_enable`? and (2) they have no dependency. Based on the above definition, in Figure 8(c), we have three predicate blocks: {Line 3}, {Line 4, Line 5}, and {Line 6}. Predicate blocks are important because they are used to determine tables and the match-action in those tables.

- *Relationships between predicate blocks.* There are three types of relationships between predicate blocks $B_a$ and $B_b$: (1) *Predicate block dependency:* A predicate block $B_a$ depends on another block $B_b$ if there is an instruction in $B_b$ that writes the predicate of $B_a$. Because each predicate is written only once, each predicate block has only one predicate block it depends on. In this case, $B_a$ and $B_b$ would be mapped to two P4 tables. (2) *Mutually exclusive:* $B_a$ and $B_b$ are located in different branches of if-else or different cases in the switch statement (*e.g.*, the NetCache program segment in §7.1). In this case, $B_a$ and $B_b$ should formulate the same P4 table. (3) *No-correlation:* If there is no dependency or mutually exclusive between $B_a$ and $B_b$, then we say $B_a$ and $B_b$ have no correlation. In this case, it is highly possible that $B_a$ and $B_b$ would formulate two P4 tables.

In Figure 8(c) example, three predicate blocks {Line 3}, {Line 4, Line 5}, and {Line 6} formulate three P4 tables. More specifically, in the first table (generated by {Line 3}), the match and action should correspond to `int_enable` and `ig_ts-eg_ts`, respectively. Similarly, {Line 4, Line 5} generate the second P4 table with actions `v1 & 0xffffffff` and `sw_id « 28`. The third P4 table is generated by predicate block {Line 6}, and has actions `int_info1 & v2`.

**Synthesis algorithm.** Algorithm 1 presents the details of the synthesis. Given the algorithm $a$'s scope, for each P4 switch $s$, we extract $\mathcal{R}_s$ (Line 2 in Algorithm 1). Because we have learned the dependencies between IR instructions from the front-end (§4.3), we group $\mathcal{R}_s$ into many predicate blocks, $PB_s$ (Line 3 in Algorithm 1). With predicate blocks in hand, we build a dependency tree, $PBTree_s$, for $PB_s$ based on the above-defined predicate block dependency.

We traverse $PBTree_s$ bottom-up. For each traversed predicate block $pb_j$, we check if it is mutually exclusive with other predicate blocks (say $pb_k$). If not, we append $pb_i$ to its parent node's table list; if yes, we merge $pb_i$ and $pb_k$, and append the merged result to their parent node's children predicate block list.

Finally, we traverse $PBTree_s$ top-down to compute whether the one predicate block should be compiled into a new P4 table or merge with the existing P4 tables. For each traversed predicate block $pb_j$, we scan its child predicate block list. For each predicate block $m$ in the *ChildPBList*, we check if $m$'s predicate is only reading $pb_j$'s table output. If so, we translate $m$ into its parent predicate block $pb_j$'s action $a_m$, and the read variable is translated into action $a_m$'s parameters (Line 12 in Algorithm 1). Otherwise, we create a new table for $m$, append it to the table list $\mathcal{L}_s$, and add the instructions in $m$ into instruction identify list $\mathcal{I}_s$. In principle, $\mathcal{L}_s$ contains tables that are potentially deployed on switch $s$, and $\mathcal{L}_s$ is used to encode resource constraints (§5.4). $\mathcal{I}_s$ contains all instructions that decide whether each predicate block in $\mathcal{L}_s$ would eventually be a table, so that we use $\mathcal{I}_s$ to encode table validity constraint below.

**Table constraints encoding.** The set $\mathcal{I}$ is one of the most important parts for the set of conditional placement constraints, so that we encode P4 table constraints based on $\mathcal{I}$. We first encode the validity of each table. Because each instruction is conditionally deployed on switch $s$, we encode the validity as: $\bigvee_{i \in \mathcal{I}} f_s(i)$. Based on $\mathcal{I}$, we can also encode (1) the dependency between two predicate blocks, (2) the match field width constraints, (3) the number of actions, and (4) the number of entries. The above-encoded constraints are put in the set of conditional placement constraints.

**Algorithm 1:** Conditional P4 implementation synthesis.

```
Input: I_a: Context-aware IR for algorithm a.
Output: L: The potential table group for all P4 switches.
Output: I: The instructions identify L.
1  foreach P4 switch s ∈ S_a do
2      R_s ← Extract(I_a)
3      PB_s ← predict_block_gen(R_s)
4      PBTree_s ← dependency_tree(PB_s)
5      foreach node pb_j ∈ PBTree_s by bottom-up do
6          if pb_j is mutually exclusive with predicate block pb_k then
7              pb_j ← merge(pb_j, pb_k)
8          append pb_j to pb_j.parent.ChildPBList
9      foreach node pb_j ∈ PBTree_s by top-down do
10         foreach m ∈ pb_j.ChildPBList do
11             if m.predicate.reads(pb_j.extern_output) then
12                 pb_j.add_action(m)
13             else
14                 L_s.append(m)
15                 I_s.append(m.table_condition)
16 return L and I
```

## 5.3 Conditional NPL Synthesis

Network programming language (NPL) is a data-plane programming language used by Broadcom [3]. It has been used to program Broadcom's ASICs, such as Trident-4 and Jerrico-2 [1]. Given the fact that Broadcom's switching ASICs account for the largest market share, we believe NPL would become increasingly more common. Similar to P4, a typical NPL program contains at least five elements: (1) header and parser, (2) logical table, (3) logical register, (4) functions, and (5) logical bus. Compared NPL with P4, NPL is more similar to C++ language. The logical table and function in NPL are, in principle, similar to virtual function and function instance in C++. This feature enables Lyra to compile the IR into the conditional NPL implementation easier than P4.

NPL synthesis takes the same inputs as P4 synthesis in §5.2. We briefly describe the synthesis approach as follows:

**Packet header and function synthesis.** Because the grammar of packet header and function in our IR is similar to NPL, synthesizing packet header and function is straightforward.

**Logical bus usage synthesis.** The logical bus in NPL handles local variables; thus, we collect all local variables in $\mathcal{R}_s$ (defined earlier), getting $\mathcal{V}_s$. We define $\mathcal{I}_{Bus}$ as a set that contains instructions reading or writing any element in $\mathcal{V}_s$. Whether $i \in \mathcal{I}_{Bus}$ should be deployed on switch $s$ can be encoded as $\bigvee_{i \in \mathcal{I}_{Bus}} f_s(i)$.

**Logical table synthesis.** NPL has a unique feature that allows multiple lookups on the same logical table; thus, we traverse all the instructions in $\mathcal{R}_s$, and merge the instructions that read the same external variables into one logical table. All the logical tables for $s$ are put in $\mathcal{L}_s$.

**Logical register.** NPL only supports name-based indexing, *e.g.*, *register_r.field_a*, so that we translate the global variables that have only one element into logical tables. For other global variables, *i.e.*, arrays containing more than one element, we distribute them across target switches.

## 5.4 Encoding Chip-Specific Constraints

We have presented how to synthesize the conditional implementation for two representative languages. We now describe how to encode chip-specific constraints. **Chip-specific constraints encoding is the key effort for the portability.** We choose reconfigurable match tables (RMT) architecture [16, 26] as an example to show how do we encode constraints for ASICs. Lyra can also encode other ASICs' constraints, *e.g.*, Tofino and Trident-4.

**RMT architecture.** RMT is a reconfigurable pipeline-based architecture for switching ASICs. The RMT architecture has an ingress and an egress pipeline. Each pipeline consists of a parser, multiple match-action stages, and a deparser. Each match-action stage has several SRAM and TCAM memory blocks, and several action units.

**Encoding chip-specific constraints.** To check whether the synthesized tables meet the underlying resource constraints, we model the architecture resources of RMT. For each table $t$ in the synthesized table group $\mathcal{L}$ (obtained from §5.2 and §5.3), we define: $M_t$ as the match field length, $E_t$ as the total number of entries, $A_t$ as the total number of actions, and $V_t$ as the validity of table $t$. Because one table can be split into multiple stages, for $t$, we define $\xi_{t,start}$ and $\xi_{t,end}$ as the start and end stages of the table $t$, respectively. If $\xi_{t,start} = \xi_{t,end}$, then $t$ should be deployed on only one stage.

We define $E_{t,s}$ as the total number of entries that table $t$ deploys on stage $s$. The stage constraints are encoded as:

$$\sum_{j<\xi_{t,start}} E_{t,j} = 0, \sum_{j>\xi_{t,end}} E_{t,j} = 0, \sum_{\xi_{t,start}\le j\le\xi_{t,end}} E_{t,j} \ge E_t \tag{1}$$

We also encode the RAM memory constraints based on [26]. Suppose each stage in the RMT switch has $N_{memory}$ RAM blocks with $h$ entries and $w$ bit-width. For each stage $j$:

$$\sum_{t\in\mathcal{L}} \lceil \frac{\lceil \frac{E_{t,j}}{h} \rceil \cdot M_t}{w} \rceil * Valid(t) \le N_{memory} \tag{2}$$

where *Valid(t)* represents the validity of table $t$, and its value is either 1 or 0. In similar ways, we can also encode other constraints such as the maximum number of stages, the maximum number of tables per stage, the maximum number of entries in the parser TCAM table, PHV allocation, predefined library-function call related resources, packet transactions [37]. Please see Appendix A for more details of chip constraint encoding. All the encoded constraints are put in the set of conditional placement constraints.

## 5.5 Encoding Deployment Constraints

Besides the constraints related to the conditional implementation and different switching ASICs, we also need to encode constraints like scope, flow path, and instruction dependencies. This section describes how to encode these constraints, which is **important for the composition**. Note that the constraints in this section cannot be encoded by integer linear programming (ILP), since ILP cannot encode "if-else" and dependency.

**Algorithm 2:** Extensible resource encoding.

**Input:** $I_a$: Context-aware IR for algorithm $a$, target switch $s$.
**Output:** $\mathcal{L}$: The variables in the resource and the existence condition.

```
S ← DownStreamSwitches(s);
foreach local variable v ∈ I_a do
    I_w ← WriteInstruction(v)
    I_r ← ReadInstructions(v)
    foreach switch s_d ∈ S do
        V_w.append(f_{s_d}(I_w))
        foreach read instruction I_r ∈ I_r do
            F_r.append(f_{s_d}(I_r))
        V_r.append(⋁ F_r)
    L[v] = (⋁ V_w) ⊕ (⋁ V_r)
return L
```

**Encoding topology constraints.** As shown in §4.3, topology constraints in context-aware IR contain two parts: algorithm scope and flow paths in the specified scope.

*Scope constraints.* For each IR instruction in $I_a$, it can only be deployed in the specified scope: $\bigvee_{I\in I_a, s\notin\mathcal{S}_a} f_s(I) = False$.

*Flow path constraints.* For each possible flow path $p$ within the scope, an instruction $I$ must be deployed on only one of switches, $s$, on each path. $\sum_{I\in I_a, s\in p} If(f_s(I), 1, 0) = 1$

**Encoding instruction dependencies.** We now encode the instruction dependencies in the context-aware IR. If an instruction $I'$ is deployed on one switch $s$ on the path $p$, then (1) for each instruction $I$ the instruction $I'$ depends on, $I$ cannot be deployed on the switches behind $s$; (2) for another instruction $I''$ depended by $I$, $I$ cannot be deployed on switches in front of $s$. Thus, we have:

$$\begin{aligned} I' \text{ depends on } I &\Rightarrow \bigvee_{s'\in prev(s,\,p), I'\in succ(I)} f_{s'}(I') = False \\ I \text{ depends on } I'' &\Rightarrow \bigvee_{s''\in next(s,\,p), I''\in pred(I)} f_{s''}(I'') = False \end{aligned} \tag{3}$$

where $prev(s, p)$ means all the switches in front of $s$ on the path $p$, $next(s, p)$ represents all the switches behind $s$, $pred(I)$ denotes all the predecessor instructions in the instruction dependency graph, and $succ(I)$ means all the successor instructions.

**Encoding external and global variables.** See Appendix B.

## 5.6 Encoding Resource Extensibility

To support extensibility, Lyra is able to handle the algorithm even though it is distributed across multiple switches, *e.g.*, splitting ConnTable on ToRs and Aggs in Figure 1. Because other constraints such as the instruction dependencies and global variable constraints are already encoded, the data can only flow from upstream to downstream (*e.g.*, from Agg to ToR in a DCN). It is impossible that the upstream switch (*e.g.*, Agg) requires a result generated by downstream switches (*e.g.*, ToR); thus, in order to encode the resource extensibility, we only consider what is the information downstream switches require from the upstream and pass this information through.

Specifically, in a Lyra program, there are two types of information required by the downstream: value in the local variable and the result of a predicate. Lyra passes this information via pushing it in the packet header, enabling the downstream switches to get it from the parser. We call such information as *extensible resources*. These resources "bridge" the upstream and downstream switches, and keep their "correlations". For example in Figure 1 LB program, suppose the ConnTable and VIPTable are deployed on the Agg and

| | $P4_{14}$ | | | | Lyra | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Synthesized $P4_{14}$ | | | | Synthesized NPL | | | |
| Program | LoC / Logic LoC | Tables | Actions | Registers | Lyra LoC / Logic LoC | Lyra Compile Time | Tables | Actions | Registers | Lyra Compile Time | Tables | Registers | Longest Code Path |
| Ingress INT | 308/99 | 9 | 8 | 0 | 207/62 | 0.987s | 8 | 7 | 0 | 0.78s | 4 | 0 | 9 |
| Transit INT | 275/66 | 6 | 6 | 0 | 193/46 | 0.914s | 5 | 5 | 0 | 0.72s | 2 | 0 | 4 |
| Egress INT | 282/73 | 7 | 7 | 0 | 197/47 | 0.897s | 6 | 6 | 0 | 0.73s | 2 | 0 | 4 |
| Speedlight | 453/351 | 21 | 23 | 6 | 194/97 | 1.352s | 16 | 20 | 6 | 0.95s | 9 | 6 | 18 |
| NetCache | 1137/937 | 96 | 96 | 40 | 372/153 | 1.909s | 12 | 14 | 40 | 1.17s | 3 | 40 | 20 |
| NetChain | 319/211 | 16 | 16 | 2 | 177/73 | 1.530s | 13 | 16 | 2 | 0.85s | 6 | 2 | 18 |
| NetPaxos | 241/140 | 6 | 11 | 5 | 150/69 | 1.158s | 6 | 11 | 5 | 0.84s | 3 | 5 | 4 |
| flowlet_switching | 195/130 | 8 | 7 | 2 | 113/43 | 0.91s | 8 | 7 | 2 | 0.70s | 4 | 2 | 12 |
| simple_router | 101/66 | 4 | 4 | 0 | 72/31 | 0.852s | 4 | 4 | 0 | 0.67s | 3 | 0 | 10 |
| switch | 4924/3876 | 131 | 363 | 0 | 4151/2563 | 33.6s | 131 | 363 | 0 | 19.4s | 125 | 0 | 53 |

**Figure 9: Experimental results conducted on a workstation with Intel Core i7 3.7GHz 6-core CPU and 16GiB RAM.**

ToR switches respectively. Given the fact that the VIPTable needs the ConnTable's table hit/miss information, Lyra needs to ask the Agg switch to pack that information in the packet header, so that the ToR switch can apply or skip the VIPTable based on the result. Similarly, if the ConnTable is split across two switches, then Lyra adds the first ConnTable's entry hit/miss information to the header.

The extensible resource encoding algorithm is shown in Algorithm 2. Because the program could be split at any position, the content in the extensible resources is also dynamic. In a nutshell, the extensible resource contains all the local variables that are not written but read by the downstream switches. Lyra checks each local variable and collects the instruction that writes or reads the variable. After the SSA form conversion (step 4 in §4.2), there should be only one write instruction $F_w$ and a list of read instructions $\mathcal{F}_r$. Next, we can calculate whether the variable is read $\mathcal{V}_r$ or written $\mathcal{V}_w$ by the downstream via the deployment boolean function. Finally, we can compute the existence condition by comparing whether two flags $\mathcal{V}_r$ and $\mathcal{V}_w$ are different.

## 5.7 Translator

With a set of conditional placement constraints, *i.e.*, the constraints encoded in §5.2-§5.6, we call an SMT solver to solve them, obtaining a solution that presents a concrete placement plan for tables, instructions, and variables. We equip the back-end with different chip language (*e.g.*, P4 and NPL) templates; thus, we can easily translate the solved plan into multiple pieces of chip-specific code. The current Lyra prototype supports $P4_{14}$, $P4_{16}$ and NPL generation.

## 5.8 Control Plane Interfaces

Lyra does not synthesize the control plane programs/functions such as installing flow table entries and configuration policies. Instead, Lyra allows the programmers to explicitly specify the tables as external variables (defined in §3.4) in Lyra program without worrying about how these tables are allocated or distributed. In other words, the connection between the control plane and data plane supported by Lyra is abstracted to the variables in OBP representation, so programmers only need to fill in the control plane tables, but do not need to know exactly how each table is mapped to target devices.

For example, Lines 36-43 in Figure 4, Lyra defines two control plane variables, `extern dict<bit[32] hash, bit[32] real_ip> [1024] conn_table` and `extern dict<bit[32] vip, bit[8] group> [1024] vip_table` via the keyword `extern` (also explained in §3.4). After this, Lyra compiler compiles the program into multiple pieces of chip code for distributing across the underlying switches. Thus, the programmers do not need to focus on the details such as hardware and resource constraints of these tables across the target switches. Lyra can also generate a set of "empty" control-plane programs for each table. For example, Lyra also generates "empty" Python functions (*e.g.*, `conn_table_entry_set(key, value)` and `conn_table_entry_get(key)`) for the programmers to easily add the code manipulating table entries. In other words, Lyra generates P4 or NPL tables according to what the Lyra program specifies, and these tables play a role as the "interfaces" between control plane code and the synthesized data plane code.

# 6 OPTIMIZATION

We further propose a collection of optimization techniques to improve the efficiency and resource usage, such as reducing the number of generated P4 tables and optimizing the results via diverse metrics. Due to limited space, please see Appendix C for details.

# 7 EVALUATION

We have built Lyra with 7,000 lines of Python code. Lyra relies on Z3 [18] for SMT solving. Lyra compiler can compile $P4_{14}$ and $P4_{16}$ for Tofino, and NPL for Trident-4.

Our evaluation aims to answer whether Lyra can successfully offer portability (§7.1), extensibility (§7.2), and composition (§7.3). The target network for our evaluation is a fat-tree data-center testbed consisting of eight servers and ten programmable switches: four ToR switches (Tofino), four Agg switches (Trident-4), and two Core switches (Tofino). All compilations were conducted on a desktop with Intel Core i7 3.7GHz 6-core CPU and 16GiB RAM.

## 7.1 Portability

To evaluate the portability, we wrote Lyra programs for the state-of-the-art network algorithms (*e.g.*, NetCache [25], and NetChain [24] and INT [2]), and then evaluated whether these Lyra programs can generate P4 and NPL code runnable on Tofino and Trident-4. The Tofino constraint is encoded according to the RMT architecture.

Figure 9 shows the comparison between our compiled chip-specific code with the manually-written $P4_{14}$ code. We evaluated Lyra in two aspects. (1) Lines of Codes: in Figure 9, LoC is the total LoC and Logic LoC is the code ignoring the header and parser because this is a better metric to show the labor on writing a program. (2) Resource usage: for P4, we compare the total number of tables, actions, and registers used. For NPL, we show the number of logical

tables and logical registers, and the length of the longest code path. All our generated code can compile on the corresponding ASICs.

First, as shown in Figure 9, Lyra can dramatically reduce the total line of codes to implement a program. It, for example, takes only 22% of LoC to implement the logic component of NetCache [25]. This shows Lyra language can describe the program more concisely.

Second, by comparing with programs written by researchers and engineers (*e.g.*, NetCache, Speedlight, and INT), Lyra can reduce the total number of tables and actions. This means we can reduce the resource occupation in the switch. For example, we observed that manually-written NetCache and SpeedLight programs have more tables than the Lyra-generated ones, because the manually-written version kept many independent tables for modularity, but Lyra merged these independent tables into a single table.

```
if (nc_hdr.op == NC_READ_REQUEST) {
    apply (check_cache_valid);
} else if (nc_hdr.op == NC_UPDATE_REPLY)
    apply (set_cache_valid);
```

In the above code in the NetCache program, `check_cache_valid` and `set_cache_valid` have no match field and only one action. Lyra merged the tables with match field `nc_hdr.op`. Note that NetCache is the only program for which Lyra can save 87.5% hardware resources. For the rest of the programs, *e.g.*, Speedlight and INT, Lyra can save 10% – 23% resources. For the programs posted on the p4c [5] project, *e.g.*, `switch.p4`, Lyra generates an equal P4 code.

In our experience, whether the manually-written ASIC code (*e.g.*, P4 or NPL) is optimal or not totally depends on the expertise and knowledge of the programmers. If the program is simple or the programmer is knowledgeable enough, it is highly possible the written code is optimal, *i.e.*, no more resource saved by Lyra. If the program is already optimized, Lyra can perform the same. However, in order to write an optimized code, even the most knowledgable programmer may need to spend tons of time and effort; on the contrary, Lyra can reduce these efforts and burdens, so that the programmer only needs to focus on implementing the logic itself.

## 7.2 Extensibility

To evaluate the extensibility, we conducted a real-world case study similar to LB example in §2.1. Initially, we set the size of both ConnTable and VIPTable to one million entries, so that these two tables can be put in the same aggregation switch. Both Tofino and Trident-4 ASICs can hold about three million entries at most. After we increased the size of ConnTable to 2.5 million and 4 million entries, respectively, Lyra can intelligently generate the response solutions. For example, if we set ConnTable's size to 4 million entries, Lyra generates an NPL-version ConnTable program with 2.5 million entries on each aggregation switch, and a P4-version ConnTable and VIPTable programs holding 1.5 and 1 million entries, respectively, on each ToR switch. Lyra also generates a function that passes the entry hit information between switches to lookup the ConnTable on ToRs, if the ConnTable on aggregation switches misses. This ensures the generated distributed programs work correctly. Compared with manually-written programs, Lyra compiles the programs for the above two updates less than 10 seconds, which only needed the programmer to change the size of the external variable ConnTable in the Lyra program; on the contrary, our well-trained programmer needed about 1.5 days to write these programs manually.

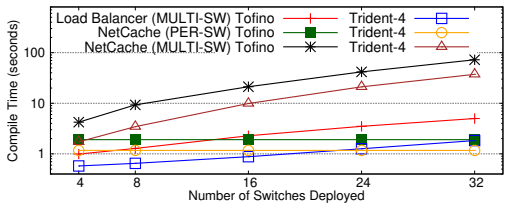


**Figure 10: The scalability for extensibility.**

**Scalability.** In general, the complexity of Lyra is related to the size of the topology, the length, and resources used by the Lyra program. To evaluate the scalability, we deployed NetCache and stateful LB on a *pod* of a simulated fat tree DCN. For the LB, we set all the switches in the pod as the scope, which means they serve together as a single LB. For NetCache, we deployed it in two modes, one in `PER-SW` mode, which means each switch has its own copy of the program; one in `MULTI-SW` mode, which is the same as the LB. We evaluated two ASICs: Tofino with the P4 and Trident-4 with the NPL, and changed the topology size by varying $k = 4$ to $k = 32$, where $k$ is the number of ports per switch and also equals to the total number of switches deployed. Figure 10 shows the result. As the topology size increases, we observe that both the `MULTI-SW` algorithms compilation time increases, but Lyra is still able to find a solution in less than 100 seconds, even in the largest topology. For `PER-SW` mode NetCache, the compilation time stays the same, because all the switches have the same program and Lyra can generate the program for each switch in parallel. By comparing two `MULTI-SW` mode algorithms, we see that the complexity of language and ASIC matters a lot: Lyra generates NPL/Trident-4 programs 2× faster than P4/Tofino programs, as NPL synthesizing needs no predicate block construction process and has shorter SMT formulas due to language complexity.

## 7.3 Composition

To evaluate composition, we attempted to deploy multiple algorithms into our testbed by changing the scope from eight switches to only one switch. For the scope, smaller is more challenging, since it evaluates whether Lyra can handle resource constraints in the code composition. First, we wrote a Lyra program including a classifier, firewall, gateway, LB, and scheduler, which is similar to Dejavu [40]. Then, we compiled the Lyra program by gradually changing the scope from the entire network to a single switch. For either case (the entire network and single switch), Lyra spent less than five seconds to generate $P4_{14}$ program that successfully compiles on the Tofino ASIC. We also asked the programmers to manually write a program for this goal manually. It took them about two days (10000× more time than using Lyra) to compress these programs into a single ASIC.

Lyra independently generates chip-specific code for each algorithm. For example, all the generated variables and tables for algorithm *firewall* are assigned the same prefix-name *firewall*. Thus, there is no shared program-level resource between generated code.

# 8 DISCUSSION

This section discusses Lyra's implementation details and limitations. More discussions can be found in Appendix D.

**Does the synthesized code always compile?** It is possible that the synthesized chip code unsuccessfully compiles on the target

ASICs, if some of the constraints are missing from the Lyra compiler. For example, egress timestamp must be collected in the egress pipeline; otherwise, the synthesized P4 code cannot compile on Tofino ASIC or is meaningless. In Lyra, we manually check and encode ASIC's resource constraints based on the ASIC specifications provided by the chip vendors. By far, while we are not aware of any "constraint missing" cases, we cannot exclude such a possibility. Thus, we offer an encoding template for the programmers to encode the missed constraints as a plug-in patch for Lyra.

**Recirculation.** P4 supports recirculation and resubmission, which allows the packet to go through the pipeline one more time. In Lyra, the programmers need to explicitly define the recirculation, as there is no switch concept in Lyra. Instead, Lyra uses the recirculation as an optimization method to pack a longer program into one switch.

**Copy-to-cpu.** Both P4 and NPL support sending a copy of the current packet to the switch's CPU but in different ways. Lyra provides a uniformed API called `copy_to_cpu()` to enable such a function. This API is translated into the corresponding APIs in different languages. Similar to the control plane interface, the programmer only needs to focus on what to do with the copied packet, rather than taking care of which switch the packet is copied at.

**Multi-pipeline support.** Programmable ASICs use multiple identical pipelines to increase the throughput. For example, the Tofino 64Q model has 4 pipelines. The program deployed on each pipeline thus is typically the same. Lyra allows the programmer to individually specify each pipeline via our OBP abstraction. Furthermore, switches, *e.g.*, RMT, split one above-mentioned pipeline into two: ingress pipeline and egress pipeline. Different from the pipeline mentioned above, these two pipelines have different capabilities: the programmer can only designate the egress port at the ingress pipeline because the egress pipeline is directly connected to the physical port and cannot re-direct the packet; all the queuing information (*e.g.*, queue length and queuing time) must be gathered in the egress pipeline as the queuing buffer sits between the ingress and egress pipeline. Lyra models the two pipelines as two individual switches and connects them via a link. Next, Lyra adds constraints that the pipeline exclusive statements cannot be deployed in the other pipeline. The SMT solver then allocates the statements.

**Synthesizing incremental changes.** Since Lyra relies on an SMT solver to generate a feasible allocation solution, a potential challenge is an incremental change in the Lyra program may result in a significantly different allocation plan. This may cause difficulty in debugging or upgrading the network in practice. Currently, if the changes are small (*e.g.*, few lines of code), our programmers manually modify the chip code, because such modifications may not violate resource constraints; if the changes are significant, we directly re-run Lyra to generate the chip code from scratch.

**Unifying different ASIC libraries.** Given that the programmable ASICs from different vendors offer different chip-specific libraries, in Lyra compiler, we hard-code a collection of functions converting these libraries into common IR. See Appendix D for more details.

## 9 RELATED WORK

**Abstractions for forwarding packets.** Software-defined networking (SDN) allows the network operators to specify the packet forwarding policy via network-wide abstractions such as SNAP [14], NetKAT [13, 31], Magellan [41], NetCore [33], and Frenetic [19]. In terms of the programming model, SNAP's one-big-switch (OBS) abstraction [14] is the most relevant to Lyra; however, the OBS model cannot explicitly specify the fine-grained scope, *e.g.*, a specific set of switches. P4Runtime [10] offers control plane-level APIs for P4 programs, rather than a compiler generating ASIC code. In general, the state-of-the-art programming models in SDN aim to generate the forwarding rules, which have different goals from Lyra.

**Programmable ASIC compilers.** The state-of-the-art efforts in programmable ASIC compilers focus on compilation for individual devices. $\mu$P4 [39] also targets portability and composition problems; different from Lyra, however, $\mu$P4 only supports P4-family programming, and does not target data plane programming across multiple switches. Jose *et al.* [26] compiles P4 programs to architectures such as the RMT and FlexPipe. Domino [37] builds upon the Banzai machine model that supports stateful packet processing, supporting a much wider class of data plane algorithms. Chipmunk [20, 21] leverages slicing, a domain-specific synthesis technique, to optimize Domino in compilation time and resource usage. Different from the state of the arts, Lyra offers a new, chip detail-orthogonal language, generates chip-specific code (like NPL and P4), and supports data plane programming across multiple switches.

**P4 synthesis for programmable NICs.** Programmable NICs (*e.g.*, Netcope [8], Netronome [9] , and Pensando [11]) support P4. Compared with Lyra, there are two differences. First, Lyra takes as input an OBP program and generates chip-specific programs for different ASIC architectures. The P4 compilers for programmable NICs take as input P4 programs and generate binary code. Second, Lyra can generate code across a distributed setting consisting of multiple programmable switches, but P4 NICs do not target such a goal. We believe Lyra is potentially extendable to programmable NICs, but this requires non-trivial extensions such as new NIC-function synthesis algorithm and NIC-specific constraints encoding.

**P4 Virtualization.** P4 virtualization (*e.g.*, Hyper4 [23], HyperV [42], HyperVDP [43], and P4Visor [44]) offers a general-purpose P4 program that can be dynamically configured to adopt behaviors equivalent to other P4 programs. Different from Lyra, P4 virtualization aims to mimic the target P4 program's behavior by configuring table entries for the underlying "hypervisor" program (*e.g.*, `hp4.p4` in Hyper4 [23]), rather than generating chip-specific code like Lyra.

## 10 CONCLUSION

Lyra is the first compiler that allows the network programmers to program data plane while achieving portability, extensibility, and composition. Lyra offers a one big pipeline programming model for the programmers to conveniently express their data plane algorithms, and then generates chip-specific code across multi-vendor switches. Our evaluation results show that Lyra not only generates runnable real-world programs (in both P4 and NPL), but also uses fewer hardware resources than human-written programs.

*This work does not raise any ethical issues.*

## ACKNOWLEDGMENTS

We thank our shepherd, Noa Zilberman, and SIGCOMM reviewers for their insightful comments. Jiaqi Gao and Minlan Yu are supported in part by the NSF grant CNS-1413978.

## REFERENCES


[1] 2019. Broadcom's new Trident 4 and Jericho 2 switch devices offer programmability at scale. https://www.broadcom.com/blog/trident4-and-jericho2-offer-programmability-at-scale.

[2] 2019. In-band Network Telemetry (INT) Dataplane Specification. https://github.com/p4lang/p4-applications/blob/master/docs/INT.pdf.

[3] 2019. NPL 1.3 Specification. https://github.com/nplang/NPL-Spec.

[4] 2019. ONE Silicon, ONE Experience, MULTIPLE Roles. https://blogs.cisco.com/sp/one-silicon-one-experience-multiple-roles.

[5] 2019. p4c, a reference compiler for P4 programming language. https://github.com/p4lang/p4c.

[6] 2020. Barefoot Tofino. https://www.barefootnetworks.com/products/brief-tofino.

[7] 2020. Barefoot Tofino's 32Q-model and 64Q-model. https://www.arista.com/en/products/7170-series.

[8] 2020. Netcope P4 – Flexible FPGA Programming. https://www.netcope.com/en/products/netcopep4.

[9] 2020. Netronome P4. https://www.netronome.com/technology/p4/.

[10] 2020. P4Runtime. https://p4.org/p4-runtime/.

[11] 2020. Pensando Expands What SmartNIC Offloads Can Do. https://pivotnine.com/2020/05/18/pensando-expands-what-smartnic-offloads-can-do/.

[12] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM (SIGCOMM)*.

[13] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic foundations for networks. In *41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[14] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. SNAP: Stateful network-wide abstractions for packet processing. In *ACM SIGCOMM (SIGCOMM)*.

[15] 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. *Computer Communication Review* 44, 3 (2014), 87–95.

[16] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando A. Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM (SIGCOMM)*.

[17] 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. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013).

[18] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *14th Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[19] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: a network programming language. In *16th ACM SIGPLAN international conference on Functional Programming (ICFP)*.

[20] Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. 2019. Autogenerating fast packet-processing code using program synthesis. In *18th ACM Workshop on Hot Topics in Networks (HotNets)*.

[21] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2020. Switch code generation using program synthesis. In *ACM SIGCOMM (SIGCOMM)*.

[22] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: query-driven streaming network telemetry. In *ACM SIGCOMM (SIGCOMM)*.

[23] David Hancock and Jacobus E. van der Merwe. 2016. HyPer4: Using P4 to virtualize the programmable data plane. In *12th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*.

[24] 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 *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[25] 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 *26th Symposium on Operating Systems Principles (SOSP)*.

[26] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. 2015. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[27] Naga Praveen Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. HULA: Scalable load balancing using programmable data planes. In *Symposium on SDN Research (SOSR)*.

[28] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[29] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High precision congestion control. In *ACM SIGCOMM (SIGCOMM)*.

[30] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. 2017. CrystalNet: Faithfully emulating large production networks. In *26th Symposium on Operating Systems Principles (SOSP)*.

[31] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Cerný. 2016. Event-driven network programming. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[32] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *ACM SIGCOMM (SIGCOMM)*.

[33] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. 2012. A compiler and run-time system for network programming languages. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[34] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. 2013. Composing software defined networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[35] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-directed hardware design for network performance monitoring. In *ACM SIGCOMM (SIGCOMM)*.

[36] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network. In *ACM SIGCOMM (SIGCOMM)*.

[37] 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 *ACM SIGCOMM (SIGCOMM)*.

[38] Anirudh Sivaraman, Thomas Mason, Aurojit Panda, Ravi Netravali, and Sai Anirudh Kondaveeti. 2020. Network architecture in the age of programmability. *Computer Communication Review* 50, 1 (2020).

[39] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. 2020. Composing dataplane programs with $\mu$P4. In *ACM SIGCOMM (SIGCOMM)*.

[40] Dingming Wu, Ang Chen, T. S. Eugene Ng, Guohui Wang, and Haiyong Wang. 2019. Accelerated service chaining on a single switch ASIC. In *18th ACM Workshop on Hot Topics in Networks (HotNets)*.

[41] Yang Richard Yang, Kai Gao, Kerim Gokarslan, Dong Guo, and Christopher Leet. 2019. Magellan: Toward high-level programming and analysis of SDN using flow algebra. In *ACM SIGCOMM Workshop on Networking and Programming Languages (NetPL)*.

[42] Cheng Zhang, Jun Bi, Yu Zhou, Adbul Basit Dogar, and Jianping Wu. 2017. HyperV: A high performance hypervisor for virtualization of the programmable data plane. In *26th International Conference on Computer Communication and Networks (ICCCN)*.

[43] Cheng Zhang, Jun Bi, Yu Zhou, and Jianping Wu. 2019. HyperVDP: High-Performance virtualization of the programmable data plane. *IEEE J. Sel. Areas Commun.* 37, 3 (2019), 556–569.

[44] Peng Zheng, Theophilus Benson, and Chengchen Hu. 2018. P4Visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*.


# APPENDIX

Appendices are supporting material that has not been peer-reviewed.

## A RESOURCE CONSTRAINT ENCODING

In this section, we detail more about how to encode the resource constraints in the Reconfigurable Match Tables (RMT) architecture [17]. For other ASIC architectures, *e.g.*, Trident-4 and Silicon One, the constraint encoding principle is the same.

## A.1 Preliminary

Before detailing each type of resource encoding, we first describe important preliminaries for the post usage.

**Predicate block.** Because RMT supports P4 [26], we first analyze the whole Lyra program and synthesize the P4 program with methods introduced in §5.2.

The synthesizing algorithm returns a list of predicate blocks $\mathcal{L}_s$ and the dependency relationship between the predicate blocks. As we stated earlier, each predicate block is potentially deployed as a table in the data plane: if any of the instructions $i$ that belongs to the predicate block $B$ is deployed in the switch $s$, then the predicate block should be in the switch. So we encode the validity $V_B$ of a predicate block $B$ on switch $s$ as:

$$V_B = \bigvee_{i \in \mathcal{I}_B} f_s(i) \tag{4}$$

where $\mathcal{I}_B$ denotes the instructions in predicate block $B$.

**Header usage.** We say a header is unused if no instruction deployed in the switch reads or writes the fields in the header. The header usage affects the resource occupation, thus we want to remove unused headers as much as possible. Similar to the predicate block validity computation, the usage of one header $h$ on switch $s$ is encoded as:

$$u_h = \bigvee_{i \in \mathcal{I}_h} f_s(i) \tag{5}$$

where $\mathcal{I}_h$ denotes the instructions that read or writes header $h$. If a header is unused, it does not necessarily mean the header takes no resource in the switch, because we also need to take into account the header dependency (see below).

**Header validity.** In the configurable parser, RMT doesn't provide any mechanism to skip header bytes. Thus we assume the packet is parsed from the starting bit until the last valid header. This means if a TCP header is parsed, then all the headers before the TCP header (IPv4/IPv6, Ethernet) are also parsed. So we need to compute the header dependency relationship based on the parser definition. For two headers $h_i$ and $h_j$, we say $h_i$ depends on $h_j$ if in the parser graph $h_j$ sits on one of the paths from the root to $h_i$. For example, the TCP header depends on the Ethernet header, but does not depend on the UDP header. Here, $h_i$ depends on $h_j$ means if $h_i$ is parsed, then $h_j$ must also be parsed.

So the validity of a header $V_h$ (whether the header should be parsed) is encoded as:

$$V_h = u_h \vee \bigvee_{h_i \in \mathcal{D}(h)} u_{h_i} \tag{6}$$

where $\mathcal{D}(h)$ denotes all the headers that depends on header $h$.

**Variable validity.** Because there is no dependency relationship between variables in the Lyra program, the variable's validity can be computed via the header usage computation method. The validity of the internal, global and external variable is denoted as $V_i$, $V_g$, and $V_e$ respectively.

## A.2 Parser

In RMT, the parser is implemented as a state machine. Each state in the state machine represents a parser node in the program and the transition in the state machine maps to the transition of the parser nodes. In the hardware, the state machine data is stored in a TCAM and a RAM table. Each entry in the TCAM table matches 32 bits of incoming data and 8 bits of parser state. The RAM table stores the next parser state and where the parsed data is stored. When a packet comes in, the parser checks both the packet header content and its own state to decide the action such as consume how many bits and jump to which next state.

The complexity of the header directly affects the number of entries in the parser's TCAM table. In a nutshell, the more headers or the more transitions between the parser nodes, the more entries in the TCAM table. For each header $h$, Lyra can compute the entries required to parse the header. For example, the TCP header requires an entry that matches the *ethernet.ether_type* field. Because one table entry $e$ may be required by multiple headers, whether it should be in the TCAM is decided by all related header's validity. The above entry is required by both TCP header and UDP header, so as long as one header is valid, the entry should be deployed in the TCAM. As a result, the deployment of an entry $D_e$ is encoded as:

$$D_e = \bigvee_{h \in \mathcal{S}_e} V_h \tag{7}$$

where $\mathcal{S}_e$ denotes the set of headers that requires entry $e$ to be deployed in the TCAM. And we can compute the total number of entries in the TCAM table by summing all $D_e$ up:

$$N_{parser} = \sum_e D_e \tag{8}$$

In RMT [17], the maximum number of entries is 256, so $N_{parser} \leq 256$. In reality, Lyra can skip this encoding when it figures out that the header is not complex enough that even all headers are valid, the encoded entries cannot overflow the TCAM table.

## A.3 Packet Header Vector

Packet header vector (PHV) acts as the bus that transmitting the data between the physical stages. It takes the data generated by the parser and contains the fields used for matching and actions' output. The basic component of PHV is called word. In RMT, there are 64, 96, 64 words with 8, 16, and 32-bit width respectively. In total, the width of the PHV is 4Kb.

How the fields are stored in the PHV is also interesting. Multiple units of small words can be combined together to act as a larger word. For example, to host a 48b source MAC address, RMT can use either six 8-bit words, three 16-bit words, one 32-bit word, and one 16-bit word, and so on.

Given a field $f$ with length $l_f$, we can calculate all packing strategies $C_f$ by dynamic programming. Each strategy $c$ is a three-element tuple $c = (c_0, c_1, c_2)$ that represents the number of words used to hold the field. Because only one strategy is valid, we introduce a Boolean variable $B_{c,f}$ for each strategy and add the constraint that only one of the Boolean variables can be true:

$$\sum_{c \in C_f} (1 \cdot B_{c,f}) = 1 \tag{9}$$

So take the 8-bit word as an example, the total PHV usage can be encoded as:

$$N_{8b} = \sum_{f \in \mathcal{F}} \left( \sum_{c \in C_f} c[0] \cdot B_{c,f} \cdot V_f \right) \tag{10}$$

where $\mathcal{F}$ denotes all the fields including header fields and internal variables, $V_f$ denotes the validity of the field $f$. If field $f$ is an internal variable, then $V_f = V_i$, if it is a header field, then the validity equals the validity of header $h$, $V_f = V_h$. Similarly, we can encode the usage of 16 and 32-bit words. The total usage should not exceed what is available in the RMT architecture.

## A.4 Memory Block

All the table entries, action parameters, and stateful objects take memory space. In RMT, the memory is counted in block: each stage has 106 SRAM blocks of 1K entries with 80b width, 16 TCAM blocks of 2K entries with 40b width.

To improve memory allocation efficiency, RMT introduces an architectural trick called word-packing. The packing allows multiple table entries packed together horizontally. For example, for a 48-bit MAC address, one 80-bit-wide entry in the SRAM block can only fit one entry, if we pack two memory blocks together and form a 160-bit wide, 1K entry packing unit, each entry can fit three 48-bit MAC addresses.

RMT does not mention the maximum block packing capability; thus, Lyra assumes it can pack as many blocks as possible. Suppose a table requires $w$ bits match width and $h$ entries, while each memory block is $w_m$ wide and has $h_m$ entries, the minimum number of memory blocks can be computed as:

$$N_{memory} = \lceil \frac{\lceil \frac{h}{h_m} \rceil \cdot w}{w_m} \rceil \tag{11}$$

Without word-packing, the number of memory blocks is:

$$N_{memory} = \lceil \frac{h}{h_m} \rceil \cdot \lceil \frac{w}{w_m} \rceil \tag{12}$$

For the assignment overhead, such as action parameters and registers, Lyra uses the method introduced in Jose *et al.* [26] to compensate for the overhead. As shown in RMT architecture [17, 26], if a table requires more blocks than a stage is able to provide, it could expand to multiple stages to meet the requirement. The memory block constraint is encoded along with the table constraint introduced in §A.6.

## A.5 Stateful Operations

RMT does not explain how to implement a stateful operation except the counters and meters. So Lyra uses the solution introduced in Domino [37] called atom. In a nutshell, an atom is a collection of hardware circuits that able to perform comparison, arithmetic operations, read and write in a single clock cycle. There are many types of atoms introduced in Domino [37], Lyra uses the most powerful one called *Pairs*.

In a Lyra program, the global variable is the only component that is stateful. When compiling to an RMT switch, Lyra uses the algorithm introduced in Domino [37] to check if the algorithm can fit into the atom. If not, then all the related instructions cannot be deployed in this switch, Lyra has to find other switches to deploy them. If so, replace all related instructions with an *atom call*. Note that this computation is done before the preparation step because it affects predicate block computations.

## A.6 Table and Pipeline

Table is the core component of the RMT architecture and implements the logic of the data plane program. The RMT requires that if one table $T_a$ reads the output of another table $T_b$, $T_a$ must be deployed in a stage after $T_b$. On the other side, RMT has only 32 stages in the ingress and egress pipeline, and according to Jose *et al.* [26], RMT only allows 8 tables per stage. So we have to deploy the tables wisely.

As we stated earlier, each synthesized predicate block is a table if it is deployed in the switch. Thus the tables' dependency relationship is already encoded in the predicate block's dependency graph. For each predicate block, based on the content and each instruction's validity, we can compute the properties of the predicate block, such as the total number of entries $E_B$, the width $W_B$.

The table and pipeline constraint can be encoded with the following steps: Firstly, because each table could be deployed across many stages, for each predicate block $B_i$, Lyra creates two integers $\xi_{B_i,start}$ and $\xi_{B_i,end}$ indicating the start and end stage of the predicate block. If $\xi_{B_i,start} = \xi_{B_i,end}$, the predicate block is only deployed in one stage.

Secondly, let $S$ denotes the number of stages in RMT, for each predicate block $B_i$, Lyra creates $S$ integers, each integer $E_{B_i,s}$ denotes the total number of entries $B_i$ deploys on stage $s$. So we have the following constraints:

$$\sum_{s<\xi_{B_i,start}} E_{B_i,s} = 0, \sum_{s>\xi_{B_i,end}} E_{B_i,s} = 0, \sum_{\xi_{B_i,start} \le s \le \xi_{B_i,end}} E_{B_i,s} \ge E_{B_i} \tag{13}$$

where $E_{B_i}$ denotes the total number of entries in $B_i$.

Thirdly, we encode the dependency relationship between the tables, if predicate block $B_i$ depends on $B_j$, then we have:

$$V_{B_i} \wedge V_{B_j} \Rightarrow S_{B_i,f} > S_{B_j,l} \tag{14}$$

Finally, Lyra encodes the memory constraint. For each stage $s$, there are $M$ memory blocks, so we have:

$$\sum_{B_i \in \mathcal{B}} \lceil \frac{\lceil \frac{E_{B_i,s}}{h_m} \rceil \cdot W_{B_i}}{w_m} \rceil * V_{B_i} \le M \tag{15}$$

## A.7 Other Constraints

Besides the above constraints mentioned in the RMT architecture, Lyra can encode other constraints using similar techniques. Such as total number of atoms per stage, number of atoms per table, number of actions per stage, power consumption limitation, *etc.*

# B EXTERNAL & GLOBAL VARIABLES ENCODING

We have presented how to encode deployment constraints in §5.5. This section further shows how do we encode external variable constraints (§B.1) and global variable constraints (§B.2).

## B.1 Encoding External Variable Constraints

An external variable can be split and deployed on multiple switches due to the memory constraint. Each external variable $e$ should co-exist with all the instructions that read $e$. Also on each possible path $p$ in the scope, there are no duplicated or missing elements of $e$. Let $\mathcal{E}_e$ denotes the total size of $e$, $\mathcal{E}_s$ denotes the number of elements deployed on switch $s$, $I_e$ denotes all the instructions that read $e$, thus we have:

$$\sum_{s \in p} \mathcal{E}_s = \mathcal{E}_e, \bigwedge_{s \in p} If(\bigvee_{i \in I_e} f_s(i), \mathcal{E}_s > 0, \mathcal{E}_s = 0) \tag{16}$$

## B.2 Encoding Global Variable Constraints

The global variable is special, because its value is maintained over the entire data plane, and any packet on different paths should be able to read and write this shared information. We encode a constraint that all the write operations and read operations that occurred earlier than the last write operation with respect to the global variable must co-exist on the same switch.

# C OPTIMIZATIONS

We put more details about our Lyra's optimizations.

## C.1 Reducing P4 Tables

**Reducing the number of P4 tables.** The first optimization aims to reduce the number of tables in the synthesized P4 programs. In one of the P4 features, the metadata in packet header can be set by `set_metadata(dst_v, src_v)` during the packet header parsing; thus, in the conditional implementation synthesis (§5.2), we traverse each extracted predicate block to identify the `set_metadata`. For each `set_metadata`, we backtrack the dependency of its `dst_v` to check whether the `dst_v` was read in somewhere. If not, we move this `set_metadata` to the parser body, preventing the `set_metadata` from being enveloped as a table. In other words, we do the best to reduce the chance of `set_metadata` generating tables. This optimization can yield a 50% reduction to the number of generated tables in our P4 INT program.

## C.2 Optimizing Results via Metrics

Lyra further offers the network programmers with options that allow them to specify a requirement for optimizing the generated chip code. Such a requirement could be maximizing the usage of a certain switch or compacting the Lyra program to minimize the number of programmed switches in each scope. To achieve this optimization, we implement the constraints specific to each option. For example, we can minimize the number of switches hosting the generated tables by minimizing $\sum_{I \in I_a, s \in p} If(f_s(I), 1, 0)$. For another example, we can maximize the number of tables on a specified switch, by assigning a much bigger weight for that specified switch and minimizing the final result.

# D MORE DISCUSSIONS

This section discusses more details about Lyra's implementation, including unifying different ASIC libraries and the expressiveness of Lyra model.

**Unifying different ASIC libraries.** Given that programmable ASICs from different vendors offer different chip-specific libraries, in Lyra compiler, we hard-code a collection of mapping functions that convert chip-specific libraries into common IR representation. In our experience, we met two types of mapping. First, libraries from different ASICs offer the same functions but with different APIs. For example, in P4, a CRC hash calculation is realized through a combination of `field_list`, `field_list_calculation`, and `modify_field_with_hash_based_offset`; while in NPL, the CRC hash calculation is implemented as a special function call. For this type, Lyra compiler provides a predefined library-function, called `crc_hash()`, which hard-codes the CRC hash calculation APIs in different programmable ASICs.

The second type is: some of the chip-specific libraries only exist on their own ASICs. For example, ASIC *A* may support range matching tables, ASIC *B* may only provide TCAM matching tables. In this case, Lyra also hard-codes mapping functions to unify the chip-specific features across different ASICs, such as converting the range matching rules (for ASIC *A*) into multiple TCAM matching rules (for ASIC *B*). For multi-switch case, Lyra would also search a solution across the ASICs defined in the given scope. If there is no mapping function and cannot find an alternative ASIC, Lyra compiler would report a compilation error.

**The expressiveness of Lyra model.** Lyra's constraint solving is built on the SMT solver; thus, our synthesis capability heavily relies on the expressiveness of SMT solver. Although our experience observed that many constraints are hard to encode, we have not seen any of resource constraints that cannot be encoded into SMT model yet. By far, while we are not aware of cases that can not be encoded by Lyra, we cannot exclude such possibility.