Cross-Platform Transpilation of Packet-Processing Programs using Program Synthesis

Xiangyu Gao
xg673@nyu.edu
New York University, USA

Jiaqi Gao
jiaqi.g@alibaba-inc.com
Alibaba Cloud, USA

Karan Kumar Gangadhar
kk5409@nyu.edu
New York University, USA

Ennan Zhai
ennan.zhai@alibaba-inc.com
Alibaba Cloud, USA

Srinivas Narayana
sn624@cs.rutgers.edu
Rutgers University, USA

Anirudh Sivaraman
anirudh@cs.nyu.edu
New York University, USA

ABSTRACT

The proliferation of programmable network devices offers a wide range of device options for developers of packet processing programs. However, there are several differences in programming language usage, hardware resource constraints, and hardware architecture across these devices. Programmers must understand multiple programming languages and hardware designs to write programs for various devices.

We propose an alternative: leveraging program synthesis to build a transpiler, Polyglotter, that outputs programs for target hardware devices from input programs written for source hardware devices. This can reduce the efforts required to write algorithms across platforms. Our evaluation results show that, compared to traditional program rewriting methods, Polyglotter can quickly produce correct results with efficient use of hardware resources. We also outline several directions for future work in such transpilers.

CCS CONCEPTS

• **Networks** → **Packet-switching networks**; **Programmable networks**;

KEYWORDS

Programmable switches; program synthesis; programmable parser; finite state machine; code generation; packet processing pipelines;

ACM Reference Format:
Xiangyu Gao, Jiaqi Gao, Karan Kumar Gangadhar, Ennan Zhai, Srinivas Narayana, and Anirudh Sivaraman. 2024. Cross-Platform Transpilation of Packet-Processing Programs using Program Synthesis. In *The 8th Asia-Pacific Workshop on Networking (APNet 2024), August 3–4, 2024, Sydney, Australia.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3663408.3663419

1 INTRODUCTION

Programmable network devices are becoming increasingly popular because of their flexibility in supporting customized network functions and their ability to handle packet processing workloads with high throughput. Devices including Barefoot Tofino [5], Broadcom Trident 4 [6], the Intel IPU [1], and the Pensando SmartNIC [4] share similar architecture (e.g., RMT pipeline [8]) but have varying hardware resource constraints. Although the emergence of these hardware devices provides more choices, the diversity of their features increases programmers' difficulty in writing code for each specific device.

To alleviate the difficulty of writing programs for multiple hardware devices, we propose automatically generating these programs using a program synthesis-based transpiler called Polyglotter. Given as input a program written for a source device, Polyglotter will take into consideration the hardware constraints of the target device before generating programs that can run on the target device.

Program synthesis tools use an exhaustive search algorithm to find an output program that is semantically equivalent to the input program. Compared with performing transpilation based on numerous rewriting rules, there are several benefits to incorporating program synthesis into the transpiler. First, it is impractical for humans to exhaustively list all rewriting rules without ignoring corner cases. This is why we believe that traditional pattern-matching compilers, consisting of many program rewrite rules, need to be repeatedly updated by adding new rules. Second, even if we could list all possible rewriting rules, the program synthesis-based approach can output more resource-efficient results as it can search through all feasible solutions within the hardware resource constraints. Manually-developed rules may guarantee semantic equivalence but cannot ensure that the outcome is ideal in resource usage. Exceeding available resources, such as pipeline stages in a programmable switch, can lead to transpilation failures.

The workflow of our transpiler is represented in Figure 1. To design our transpiler, we use hardware configuration files to help the transpiler interpret the semantics of the input program and generate programs that follow the target hardware device's resource constraints. This requires the transpiler developers to be familiar with the source and target hardware constraints (§2); and the programming language semantics for the input and output programs (§2). They need to ensure that output programs do not exceed the target devices' resource limit and are semantically equivalent to input programs. All of these should be encoded into our transpiler.

However, building the transpiler using program synthesis requires overcoming a key challenge: long running time due to the large search space of all implementation candidates. Hence, our design divides the whole synthesis problem into several smaller steps, each of which only solves a subproblem (§3.3.3), either state transition or packet field extraction, of the whole transpilation, leading to faster transpilation speed.

We test this idea over P4 programs and NPL programs that exercise the packet parser on the Tofino switch [5] and Trident 4 switch [6]. The preliminary results (§4) show that Polyglotter can quickly generate correct target programs that are semantically equivalent to source programs. Compared with transpilation using

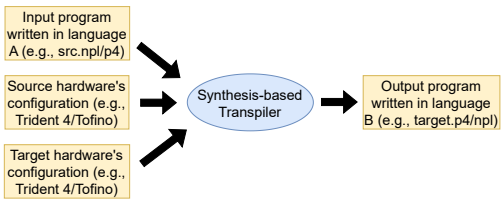

Figure 1: Workflow of the transpiler design where language A and B can be the same or different.

several rewrite rules, our output is more efficient in terms of resource usage. We outline future directions to extend our approach to entire packet processing programs including the parser, pipeline, and stateful packet processing functions across devices.

2 WHERE IS TRANSPILATION USEFUL?

A transpiler should tackle the differences between the source/target programming language designs and source/target hardware constraints. Below, we list several examples of language and hardware differences that necessitate transpilation, with source and target language snippets. When these examples occur concurrently in one program, there could be a combinatorial explosion for the total number of semantically equivalent output options, each with different usage for different types of hardware resources.

2.1 Wide state transition key

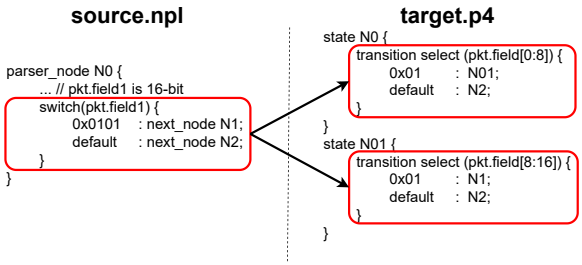

Figure 2: Transpiling a parser node with wide match key (16 bits) into multiple parser nodes with narrower match key (8 bits).

A hardware packet parser identifies headers and extracts packet fields for subsequent processing in the ingress/egress switch pipeline. A parser is commonly modeled as a finite state machine (FSM). Each state is a *parser node*. Each parser node can take a packet field as its state transition key and transit to subsequent states based on the key's value. However, hardware limits the bit width of a state transition key and this limit varies across platforms. We may have to convert one parser node on a particular hardware into multiple parser nodes on a different hardware.

As a concrete example shown in Figure 2, *N0* from the input program on the LHS has a transition key `pkt.field1` of size 16 with 2 switch cases. The language of the output program has a limit on the transition key to be of size ≤ 8. As a result, the transpiler needs to use 2 parser nodes, each of which checks *8 bits* of the transition key. One solution is presented on the RHS of Figure 2. In general, when the size of the transition key is larger than the transition key width limit of the target hardware, multiple parser nodes are required in the transpiled program to express the same semantics.

2.2 Table operations in parser

In a parser, the allowed operations might be different across languages and hardware devices. In NPL programs, the language allows `parse break` and `parse continue` to allow interleaving parser and logical table operations. Specifically, the Trident switch hardware jumps out of the parser after `parse break`, does a series of logical table operations, and jumps back to the parser after `parse continue`. However, such interleaving is not allowed in P4 programs. So the transpiler should express the same behavior in a different manner that is supported by the P4 programming language.

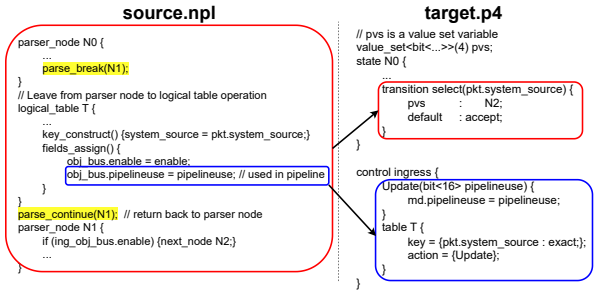

Figure 3: Transpiling parse break/continue into the a value set data structure and table operation in a pipeline.

The logical table in NPL performs match action operations based on rules from the control plane. If we put one logical table between 2 parser nodes, the variables updated in the logical table might influence the behavior of the subsequent parser node. We observe that, depending on the rules programmed by the control plane, a P4 `value_set` may be used to mimic the same semantics of an NPL table that influences the behavior of a downstream parser node. A P4 `value_set` is a data structure that may be used as part of a parser transition to check whether or not the value of a packet field belongs to a set.

The NPL program in Figure 3 has one logical table between 2 parser nodes (parser N0 and parser N1). This table updates a bus field (`obj_bus.enable`) that is used in parser node N1. Even though P4 does not support table operations within its parser, it can implement the same semantic by using `value_set`. In this example, checking the updated value of the bus field is equivalent to checking the execution status of the logical table. Thus, we could know the updated value by checking whether the key of the logical table is in the match set or not using `value_set`.

Additionally, logical tables within the parser in languages such as NPL might update variables that are used in the pipeline. In Figure 3, `obj_bus.pipelineuse` is updated in one logical table of the parser and used in the pipeline later. Given P4 language cannot do temporary variable updates in the parser, the generated transpiled program adds one extra P4 table at the beginning of the

pipeline to guarantee that the rest of the pipeline can witness the latest value of `obj_bus.pipelineuse`. There could be other ways to generate semantically-equivalent transpiled programs as well.

2.3 Multiple lookups per table

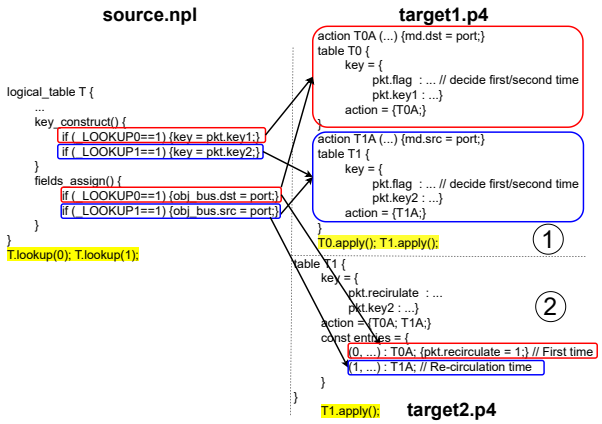

Figure 4: LHS shows an NPL program with 2 lookups for 1 logical table; the RHS shows 2 alternatives for the transpilation results in P4.

We use packet headers and temporary variables to store intermediate information during the parsing process. We use the table data structure to update packet headers or temporary variables based on match-action rules. But there are differences in the # lookups per table across languages (e.g., NPL vs P4).

The program on the LHS of Figure 4 shows one logical table in NPL which can support 2 lookups. Allowing more than one lookup per table can let programmers write more complex functions into one table without using multiple tables. More benefits of this feature in the NPL language can be seen from the NPL spec [2]. However, P4 can only support one lookup for each table. This restriction forces the transpiler to find non-trivial ways to express the same functionality in P4.

We show two methods to implement the multiple lookup functionality in P4. The method labeled ① in Figure 4 defines 2 tables, each containing a copy of the match-action rules in the NPL table. Each table maps to one lookup in the NPL's logical table. We need to add one extra boolean match key `flag` in each P4 table. If its value is 0, it maps to the first lookup from NPL; otherwise, its value is 1, meaning it is the second lookup from NPL.

It is also possible to use packet re-circulation to transpile multiple lookups. ② of Figure 4 realizes such an implementation using only 1 table by adding an extra match key `pkt.recirculate`. The benefit of re-circulation is that it can store useful information in a first pass and continue processing packets in a second pass. But this may reduce the throughput. The choice of which transpilation method to use depends on the programmers' objectives.

2.4 Initialization for temporary variables

The initialization approaches for temporary variables are different in the NPL and P4 languages. Specifically, the NPL initializes all temporary variables (called bus fields in the NPL program) through one initialization function before the parser while P4 does so by extracting bits from the input bit stream at the beginning of the parser. This difference leads to a potential failure in transpilation.

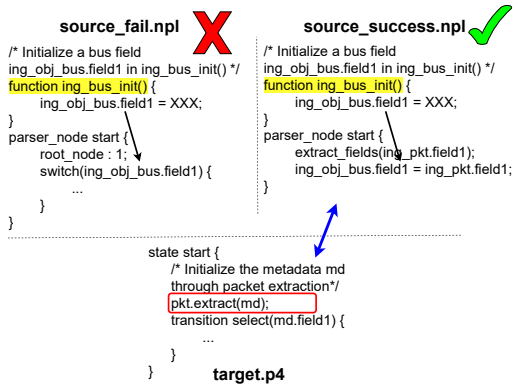

Figure 5: Different ways to initialize temporary vars.

Figure 5 gives a concrete example. `source_fail.npl` initializes `ing_obj_bus.field1` and the initial value decides transition logic in the parser node `start`. We cannot find an equivalent expression in P4 due to its language design constraints. However, `source_success.npl` initializes `ing_obj_bus.field1` but its value is later updated by one extracted packet field. We can express this semantic through packet extraction shown in `target.p4`.

3 WORK: AUTOMATED PARSER TRANSPILATION

We build Polyglotter, a cross-platform transpiler for the parser portion of NPL and P4. This requires us to handle transpilation examples such as those mentioned at §2.1 and §2.2. All other cases described in §2 are beyond the parser and left for Polyglotter's future development. It consists of 3 main steps (Figure 7). In step 1, Polyglotter generates the corresponding intermediate representation (IR) from the input program; step 2 turns the IR format into a semantically equivalent version that follows the target hardware's constraints using program synthesis; in step 3, Polyglotter lifts the synthesized IR format back to the target programming language.

3.1 IR design for parser behavior

Designing a new IR format can simplify complex data structures (e.g., logical table in NPL and value set in P4) and help extend the transpilation process across more hardware platforms and programming languages. Verifying the semantic equivalence at the IR level is much simpler and more generic.

The parser *determines* packet headers from an input bit stream through a sequence of parser nodes. Therefore, the IR design should be capable of reflecting the parser's behavior (e.g., state transition and packet header extraction). Inspired by [17], Figure 6 shows the IR grammar. The general workflow of a parser starts from variable declaration for packet fields, temporary variables, and extraction status bit vectors. Then, the whole parser will be represented by a nested if-then-else (ITE) block.

Parser node transition. We use a sequence of ITE blocks to model the parser behavior. In the nested ITE statement, the predicate consists of at least one clause and each clause checks whether

Literals:	c	::=	NUMBER
Extraction status bit vector:	bv	::=	BIT VECTOR
Bit vector:	I	::=	INPUT BIT VECTOR
Value sets:	S	::=	$S_1 \mid S_2 ... \mid S_{k1}$
Packet fields:	pkt	::=	$f_1 \mid f_2 ... \mid f_{k2}$
Temp variables:	tmp	::=	$t_1 \mid t_2 ... \mid t_{k3}$
Operations:	op	::=	"==" | "in" | "="
Expressions:	e	::=	$c \mid pkt \mid tmp \mid bv[c] \mid S \mid I[c:c] \mid$ e op e
Predicates:	$pred$	::=	e (&& e)* $\mid TRUE$
Statements:	s	::=	$e \mid s\ ; s \mid \epsilon \mid$ if $(pred)\{s\} \mid$
			if$(pred)\{s\}$(elif(pred)$\{$s$\}$)* else$\{s\}$
Packet fields definition:	$pktDef$	::=	bit<c> pkt; $pktDef \mid \epsilon$
Temp variables definition:	$tmpDef$	::=	bit<c> $tmp = c$; $tmpDef \mid \epsilon$
Bit vec definition:	$bvecDef$	::=	bit<c> $bv = \{0, 0, ..., 0\}$; $\mid \epsilon$
$p \in$ program		::=	$pktDef$; $tmpDef$; $bvecDef$; s

Figure 6: The BNF of IR for parser.

the value of one variable is equal to a constant or belongs to one set. We include "True" as a possible predicate for completeness. All these predicates are concatenated by "&&". The "||" logic can be represented by using the *elif* statement it is not in the predicate IR. If the predicate is satisfied, it will enter into the next level ITE block, meaning the transition to the next parser node; the *else* keyword can be considered as the *default* statement in the parser.

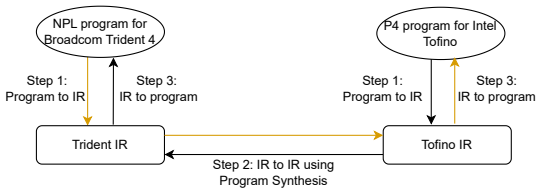

Figure 7: 3 steps in Polyglotter's transpilation.

Packet field extraction. There are 3 types of operators (e.g., ==, in, and =) in the IR design, and the operator "=" is used to define the packet field extraction behavior. To be specific, *bv* is a set of indicator variables, each of which represents whether a corresponding packet field is extracted or not. Setting its value to 1 means this packet field should be extracted in the parser node. Then, we update a packet field's value by setting it to one slice of a input bit stream using *I[c:c]*. Our current IR design is expressive to express parser for both Tofino and Trident 4 programmable switches.

3.2 Step 1: Generating low-level IR

In this IR generation step, developers need to make sure that the generated IR is semantically equivalent to the input program and complies with hardware capabilities. Polyglotter analyzes the input program's semantics and outputs the IR format. This process involves predicate generation and packet extraction generation.

Predicate generation. In our IR design (Figure 6), one predicate might include several clauses and each clause is to compare a variable (e.g., packet field or temporary variable) against either a constant or a set. In hardware such as Tofino, it does not support temporary variable modification in the parser so we should treat packet fields and bus fields differently. Specifically, if the variable

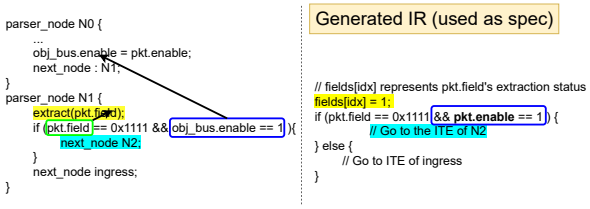

Figure 8: Predicate generation.

is a packet field, we should leave it as it is. However, when the variable is a temporary variable, Polyglotter does a backward dataflow analysis to find all possible packet bits that influence this temporary variable, and then replaces the predicate by one or more conditions over those packet bits.

As a concrete example, the predicate in Figure 8 has 3 clauses "`pkt.field==0x1111 && obj_bus.enable==1`". The predicate replacement pass leaves the packet field (e.g., `pkt.field`) as it is. However, it has to determine the packet bits that affect the value of bus fields. Given the temporary variable `obj_bus.enable` is affected by `pkt.enable`, the clause `obj_bus.enable==1` is replaced by `pkt.enable==1`.

Packet extraction generation. Each parser node might extract a range from the input bit stream and set values to several packet fields. Our IR design regards the whole parser as a nested ITE block. Polygotter uses one *bit vector* to represent each packet field's extraction status. In the output IR, we represent the packet extraction behavior by setting the corresponding indicator variables to 1.

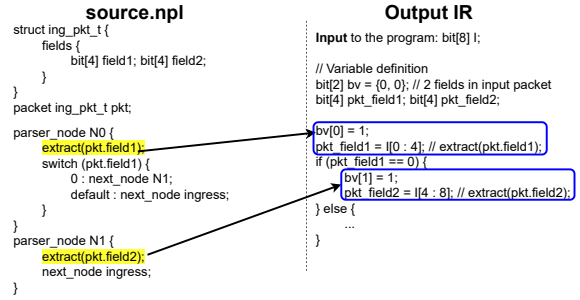

Figure 9: The generated IR for one given input NPL program.

Output IR from input program. After predicate generation and packet extraction generation, we can output the IR format. Figure 9 provides one concrete example of the IR generation process for an NPL program. Specifically, in the output IR, we use a bit vector variable, *bv*, and some other bit vector variables, *field1* and *field2*, to record the extraction status of each packet field. There are 2 fields in the input program so the size of the bit vector is 2 as well. All other bit vectors are used to store the extracted packet fields' value. The whole parser is presented by a large "nested" ITE statement, where each if predicate is one state transition logic, determining the subsequent packet extraction behavior of the parser.

3.3 Step 2: Generating IR for target device

Polyglotter encodes hardware constraints and generates the target IR format through program synthesis. The output program not only guarantees the semantic equivalence but also obeys the resource

constraints. We divide the IR to IR transformation into 2 parts: **synthesis for predicates** and **synthesis for packet extraction**.

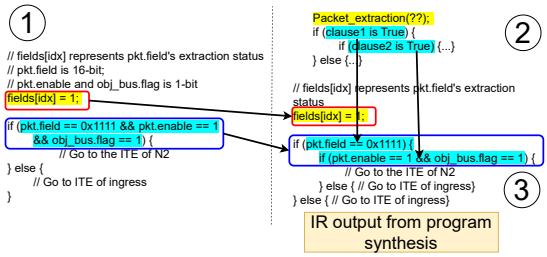

Figure 10: Synthesis-based IR generation for target.

3.3.1 Synthesis for predicates. Polyglotter splits the predicate synthesis into multiple sub-problems, each of which takes all predicates within *one parser node* as the specification. Hardware resource constraints such as the size limit of a state transition key will be encoded. As an example, if the target device limits the transition key to be at most 16 bits, we cannot fit the if-condition (the transition key is 18 bits) in ① of Figure 10 into 1 parser node and need to use 2 in ② of Figure 10. In addition, Polyglotter captures common conditions in all if-predicates and places them before other conditions in the generated IR. Its benefits will be discussed in §3.3.3.

3.3.2 Synthesis for packet extraction. The next step is to decide where in the process of parsing we extract specific packet fields. In this case, the specification is the generated IR after predicate synthesis where we could replace all conditions with one boolean variable; while the implementation is a partial program where we put one packet extraction function in each if-else body. The function will decide what new fields to extract in this node. In ③ of Figure 10, the transpiler represents the packet field extraction behavior by setting the indicator variable `fields[idx]` to 1. Polyglotter ensures that each packet field is extracted at most once in the parser.

3.3.3 Why do we use the program synthesis-based approach?
Synthesis for ITE predicates. Polyglotter generates the implementation of all ITE predicates within a parser node at one time. Usually, the state transition logic within one parser node is not that complex so the synthesis engine can output the result quickly. Besides, compared with a manual approach to creating the result, using synthesis tools can reduce resource usage (e.g., # transitions and # nodes). For example, in Figure 11 the input program does a

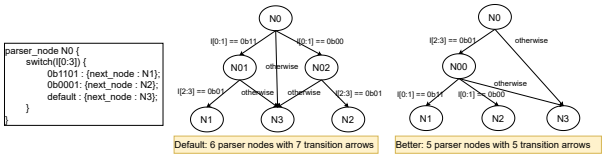

Figure 11: Two semantically equivalent transpilation results with the limit of transition key size to be 2 bits. Each result uses different resources.

parser state transition depending on one 4-bit variable. The target hardware can match at most 2 bits in one parser node. One way is to match the first 2 bits and then the next 2 bits in sequence before deciding the next transition parser node. This example is illustrated as in the first tree-based finite state machine which uses 6 parser nodes and 7 transition arrows. However, the right figure shows an alternative, which only needs 5 parser nodes and 5 transition arrows by matching the last 2 bits before matching the first 2 bits. In our synthesis procedure, we let the synthesis engine find common conditions among all predicates and the transpiler will prioritize putting these common conditions before others.

Synthesis for packet extraction. We let the synthesis solver determine the concrete extraction behavior in each generated parser node. Given there are several paths from the first parser node to the last one, it is both time-consuming and error-prone for humans to manually figure out a solution. However, a synthesis solver can generate the correct result quickly.

Benefits of decoupling predicate synthesis and packet extraction synthesis. Compared to generating the whole parser from one synthesis problem, we need to do implementation configurations exploration. Each configuration is determined by different hyper-parameters (e.g., the number of state transition rules per parser node, the value of each constant value, and the total number of parser nodes). Instead, Polyglotter generates the IR format for state transition predicate and packet extraction in sequence. Synthesizing all ITE predicates can give us the ITE skeleton and the subsequent synthesis is to fill packet extraction behaviors into each ITE's body. Therefore, such decomposition not only makes the transpilation faster but also avoids the skeleton exploration work.

3.3.4 Why do we use different granularity for predicate and extraction synthesis? We synthesize all predicates within *a parser node* because common conditions may exist among all these predicates. There are other alternatives. For instance, we could synthesize predicate by predicate, but this may lose the benefits of finding common conditions from these predicates; if we synthesize all paths within the whole parser together, there may be no conditions shared by them. Synthesizing all packet extraction operations for *the whole parser* is a good balance. First of all, the search space is proportional to # packet fields and # parser nodes, which is in a reasonable size; additionally, this choice can leverage the advantage that some packet extraction behavior can be shared across paths instead of appearing multiple times.

3.4 Step 3: Lifting to switch program

The last step is to turn the IR back to a complete program written in the target language that can run over the target hardware. This step can be considered as the reversed process for step 1. The concrete implementation is just a straightforward rewriting process. For instance, in our IR design, we use indicator variables to determine the extraction status of a packet bit but in a high-level program, we should replace it with corresponding keywords that follow the language syntax. Currently, we manually transform the synthesized output IR to the high-level programs in the target language.

4 EVALUATION

We measure Polyglotter in 2 main parts. **Correctness**: can it correctly generate the output program within a reasonable time period? **Efficiency**: how many resources does the output program consume?

Program	Polyglotter			Rule-based transpilation		
	# nodes	# transition	depth	# nodes	# transition	depth
Wide Key	6	7	3	5	5	3
Parser break/continue	3	3	3	0	1	1
Temporary variable initialization	1	0	1	1	0	1
Move update to pipeline	2	2	2	2	1	2

Table 1: Resource usage comparison.

Setup. We generate benchmarks by extracting portions of one NPL program used in production. All these benchmarks reflect some features mentioned in (§2) because we want to check whether Polyglotter can successfully realize the transpilation for programs with these features. The syntax of the output P4 program is checked by the commercial Tofino compiler and the semantics are verified manually. Currently, Polyglotter focuses on the parser portion and realize the transpilation from a NPL program for the Trident 4 switch to a P4 program for the Tofino switch. We plan to extend it into the pipeline portion and realize multi-directional transpilation across languages in the future.

Results. Preliminary results are in Table 1. The rule-based transpilation implements a series of rewriting rules to get the output program, including dividing a condition with a big transition key into several conditions with small transition keys in order. All these rules might generate locally optimal results but the combination of several rules can be far from the globally optimal outcome. In fact, Polyglotter can successfully generate correct output for 4 representative benchmarks that fit the target hardware's constraints within seconds. In terms of the resource usage, Polyglotter can output results that use fewer number of # nodes, # transition, and the depth of the parser. In addition, it is useful to decompose the whole synthesis problem into 2 subproblems (e.g., predicate and packet extraction) to speed up the transpilation procedure.

5 RELATED WORK

Programming languages for programmable network devices. P4 [3] and NPL [2] are widely used programming languages for engineers to develop programs for high-speed network devices. Many domain-specific languages (DSLs) have emerged in academia to remedy some of the shortcomings of P4 and NPL. Examples include Lyra [10], microP4 [20], Domino [18], Lucid [19], FlightPlan [21], P4All [15] and NetKAT [7]. Different languages serve different design purposes, such as offering convenient data structures [15] and incorporating the target hardware's specific features [10] [11]. Our work, with a focus on cross-platform transpilation of existing DSLs, is complementary to the design work of new DSLs. This paper aims to unify features of different languages and hardware devices by building a cross-platform transpiler. This could avoid forcing programmers to understand several language-specific and hardware-specific features.

Program synthesis for compilers. Program synthesis [14] selects one program from a space of programs such that this program is semantically equivalent to the input specification. This technique was used in compiler design (e.g., Chipmunk [11] and CaT [12]) for programmable network devices. However, their focus is to compile a high-level packet processing program into low-level representation without considering the parser portion of the program. In addition, the output of our proposed transpiler is a high-level program that satisfies the constraints of a target that can have a different architecture from the source.

Expressing and verifying parser behavior. The principles of parser design were discussed by Gibb *et al.* [13]. We can consider a parser as a finite state machine (FSM). The state transition logic can be determined by comparing certain fields against constants and each parser node executes packet extractions to identify packet headers from the input bit stream. Leapfrog [9] proposes a new framework to verify the equivalence of 2 parser structures. However, our goal is to generate a new parser through a synthesis procedure, which is orthogonal to their implementation.

6 FUTURE WORK

In this section, we discuss 3 future directions.

Is the transpiler retargetable to a more complex parser structure? At a high level, a parser identifies headers and extracts packet fields for subsequent processing in the ingress/egress pipeline. Different hardware parsers might support various operations including specific hardware constraints and configuration. As an example, Kangaroo [16] supports a non-streaming parser design that does not limit the window size while Gibb *et al.* [13] designs a streaming packet parser that restricts the window size. So the natural question becomes: could we easily tailor our synthesis-based transpiler to parsers in various hardware devices? We believe this is possible but requires the transpiler to encode new operations and adjust the setting to be consistent with the target hardware's constraints.

Is the transpiler extensible to the packet processing pipeline portion of a network device? The current implementation of the transpiler is over the parser level of a packet processing program but we want to extend to the full program finally. Then, the research question is: can we encode the computation capabilities (e.g., ALU and extern functions) into our transpiler? Can the transpiler encode all hardware constraints automatically through a unified hardware description language? Such an extension can help the transpiler go beyond the parser and realize program-to-program transpilation.

Can we develop a transpiler for software platforms? This paper is about developing a transpiler for various programmable network devices. Can we extend the technique to software targets such as eBPF or DPDK framework? In such cases, the transpiler needs to transform the P4 to C program. It can focus less on the hardware constraints or language features but should pay more attention to the performance of the output program (e.g., runtime).

7 CONCLUSION

The proliferation of programmable network devices motivates engineers to write programs for different targets. This paper proposes to build a program synthesis-based transpiler, Polyglotter, that realizes transpilation for the parser part of P4 and NPL programs over Tofino and Trident 4 programmable switches. Our initial results show the correctness and efficiency of Polyglotter. We hope our work can prompt further research on the transpiler design across more network languages and hardware devices.

ACKNOWLEDGMENTS

We are grateful to the anonymous APNet reviewers for their valuable comments on previous drafts of this paper. We thank Tiancheng Hou, Tao Wang, Fabian Ruffy, and Zhanghan Wang for their help in understanding the parser architecture in detail. This work was supported in part by NSF grants CNS-2008048 and CNS-1910796.

REFERENCES

[1] [n. d.]. Intel® Infrastructure Processing Unit (Intel® IPU). https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html. ([n. d.]).

[2] [n. d.]. NPL Specification. https://github.com/nplang/NPL-Spec. ([n. d.]).

[3] [n. d.]. P4-16 language specification. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html. ([n. d.]).

[4] [n. d.]. Pensando Distributed Services Architecture SmartNIC. http://www.servethehome.com/pensando-distributed-services-architecture-smartnic/. ([n. d.]).

[5] [n. d.]. Product Brief Tofino Page | Barefoot. https://barefootnetworks.com/products/brief-tofino/. ([n. d.]).

[6] [n. d.]. Trident 4 / BCM56690 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56690. ([n. d.]).

[7] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: semantic foundations for networks *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/2535838.2535862

[8] 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. *SIGCOMM Comput. Commun. Rev.* 43, 4 (aug 2013), 99–110. https://doi.org/10.1145/2534169.2486011

[9] Ryan Doenges, Tobias Kappé, John Sarracino, Nate Foster, and Greg Morrisett. 2022. Leapfrog: certified equivalence for protocol parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 950–965. https://doi.org/10.1145/3519939.3523715

[10] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *ACM SIGCOMM*.

[11] 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 *(SIGCOMM '20)*. New York, NY, USA, 44–61. https://doi.org/10.1145/3387514.3405852

[12] Xiangyu Gao, Divya Raghunathan, Ruijie Fang, Tao Wang, Xiaotong Zhu, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2023. CaT: A Solver-Aided Compiler for Packet-Processing Pipelines. In *ACM ASPLOS*. New York, NY, USA, 72–88. https://doi.org/10.1145/3582016.3582036

[13] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. 2013. Design principles for packet parsers. In *Architectures for Networking and Communications Systems*. IEEE, 13–24.

[14] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

[15] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. 2022. Modular Switch Programming Under Resource Constraints. In *USENIX NSDI*.

[16] Christos Kozanitis, John Huber, Sushil Singh, and George Varghese. 2010. Leaping Multiple Headers in a Single Bound: Wire-Speed Parsing Using the Kangaroo System. In *2010 Proceedings IEEE INFOCOM*. 1–9. https://doi.org/10.1109/INFCOM.2010.5462139

[17] Muhammad Shahbaz and Nick Feamster. 2015. The case for an intermediate representation for programmable data planes. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*. Association for Computing Machinery, New York, NY, USA, Article 3, 6 pages. https://doi.org/10.1145/2774993.2775000

[18] 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*.

[19] John Sonchak, Devon Loehr, Jennifer Rexford, and David Walker. 2021. Lucid: A Language for Control in the Data Plane. In *ACM SIGCOMM*.

[20] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. 2020. Composing dataplane programs with μP4. In *ACM SIGCOMM*.

[21] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. 2021. Flightplan: Dataplane disaggregation and placement for p4 programs. In *USENIX NSDI*.