ParserHawk: Hardware-aware parser generator using program synthesis

Xiangyu Gao[w]* Jiaqi Gao[a] Karan Kumar G[n] Muhammad Haseeb[n] Ennan Zhai[a]
Bili Dong[g] Joseph Tassarotti[n] Srinivas Narayana[r] Anirudh Sivaraman[n]
[w]*University of Washington* [n]*New York University* [a]*Alibaba Group* [r]*Rutgers University* [g]*Google*

Abstract

Parser programs are becoming increasingly complex to accommodate intricate network packet formats and advanced protocols. Existing parser compilers incorporate predefined program rewrite rules to output the low-level parser implementation. Yet, these rules are often brittle and sensitive to how the input parser program is written. As a result, generated implementations could consume more hardware resources than necessary. In some cases, these compilers unnecessarily reject valid parser programs that could have fit within the target device parser's resource constraints.

We leverage program-synthesis-based techniques to build a parser compiler, ParserHawk, for 2 network devices: the Intel Infrastructure Processing Unit (IPU) and the Barefoot Tofino programmable switch. Naively formulating code generation as a program synthesis problem can take hours, if not days, to complete. As a result, ParserHawk incorporates several optimization algorithms, which achieve a geometric mean speed-up of 309.44×. Within a compile time on the order of minutes for most benchmarks, ParserHawk can correctly compile parser programs rejected by existing compilers and can generate parser implementations that use fewer hardware resources.

CCS Concepts

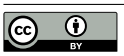
• **Networks → Packet-switching networks**; • **Theory of computation → Formal languages and automata theory**.

Keywords

Programmable Parser; Code Generation; Program Synthesis; Constant Synthesis

ACM Reference Format:
Xiangyu Gao, Jiaqi Gao, Karan Kumar G, Muhammad Haseeb, Ennan Zhai, Bili Dong, Joseph Tassarotti, Srinivas Narayana, Anirudh Sivaraman. 2025. ParserHawk: Hardware-aware parser generator using program synthesis. In *ACM SIGCOMM 2025 Conference (SIGCOMM '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3718958.3750484

*Work was done partially at New York University.



SIGCOMM '25, Coimbra, Portugal

ACM ISBN 979-8-4007-1524-2/25/09
https://doi.org/10.1145/3718958.3750484

1 Introduction

Programmable network devices have become popular in data center networks. We have seen the emergence of several programmable network devices [1, 8, 10, 14, 20–22] from various vendors (e.g., Nvidia, AMD, Intel) over the past several years. A typical programmable network device includes a line-rate programmable parser and a packet-processing pipeline. The parser identifies headers within the packet and stores them in some structured format (e.g., a vector of fixed-length containers) for the downstream packet-processing pipeline to update. More precisely, a network device's packet parser is responsible for turning unstructured bitstreams into a structured collection of packet headers based on a specification of the parser's behavior.

Today's high-end programmable packet parsers run at line rate [8, 21, 25, 33, 35] and flexibly parse diverse header formats—a flexibility that is increasingly essential for parsing many diverse and dynamic protocol headers such as Geneve [7]. Enterprises gradually migrate their business workloads to data centers managed by professional cloud providers, and therefore, network traffic within a data center becomes diversified to support a wide range of services and application-level requirements. Recent partnerships between Google Cloud and CME [3] drive the need for more sophisticated parsing logic to analyze financial traffic and identify packet origins within data centers [38]. These trends suggest that advanced parsers are required to identify packets sent from different sources (e.g., internal servers, premium-level customers) within data centers.

Can existing compilation techniques handle the increasing need to support complex packet formats over diverse parser targets? Surprisingly, there is limited prior work on compilation algorithms for programmable parsers. Gibb et al. [33] propose a dynamic programming algorithm for parser compilation, but it only supports compilation for one parser architecture. Commercial parser compilers incorporate heuristics to improve the quality of generated code. Yet, their compilation process may still fail to find a good implementation (§3.2) under certain conditions, especially when the parsing logic is complex and hardware constraints are tight. Similar to Gibb et al., commercial compilers are specially designed to generate code for one device and cannot target diverse network devices. Based on our conversation with multiple cloud providers, their developers, usually unaware of all hardware constraints, spend excessive time "reshaping" parser programs to pass compilation, significantly reducing production efficiency.

Briefly, a compiler's job is to convert a parser *specification*, represented in a language like P4 or NPL, into an *implementation* based on a finite-state machine, realized using high-speed ternary content-addressable memory (TCAM). The size of the TCAM in programmable parsers is quite limited, so it is important for the

compiler to generate an efficient implementation that fits within the limited resources. The inefficiency of existing compilers stems from the fact that parser compilation is inherently a *combinatorial problem*. The compiler must solve a resource allocation task under multiple resource constraints such as state transition key size limits and TCAM entry limits, as elaborated in §5.1. Existing parser compilation algorithms fail to consistently generate high-quality compilation results across all parser programs and hardware architectures.

Our approach. To generate high-quality parser implementations, this paper presents ParserHawk (§5), a program-synthesis-based compiler for programmable line-rate parsers that leverages a combinatorial search engine to explore better compilation outcomes. Concretely, we first analyze the input parser program to get semantic information—such as header transition logic and field extraction order—and take that as the specification. Next, we generate a parameterized parser skeleton that encodes a finite-state machine structure with symbolic variables representing state transitions and parsing actions within a state. Then, we use the Z3 solver in a counterexample-guided synthesis (CEGIS) [40] loop to synthesize a concrete implementation by assigning values to these symbolic parameters, thereby completing the parser skeleton.

Naively encoding the parser generation process as a synthesis problem fails to work in practice because of the long compilation time. The reason is that the search space can be too large for a solver to quickly find a solution. In response, we propose multiple optimization algorithms (§6) to remedy this problem. One representative algorithm (§6.4) involves learning from the semantics of the input parser programs and subsequently guiding the synthesis solver to find constant values within a reduced search space.

Findings. We generate several new benchmarks starting from both open-source production and self-created synthetic parsers, including a subset of switch.p4's parser program. In practice, these open-source implementations guide parser developers in designing new parsing logic. ParserHawk successfully compiles all these benchmarks (§7) while existing compilers fail to get the compilation output for some of them (11 out of 58). ParserHawk's output is more resource-efficient by using fewer TCAM entries or parser stages. Optimizations of the synthesis process offer a geometric mean of 309.44× in compilation speed-up on average, leading to >80% of the benchmarks completing compilation in one minute.

We evaluated ParserHawk on programs targeting 2 programmable parser platforms, the Barefoot Tofino switch and the Intel Infrastructure Processing Unit (IPU). Although different programmable parsers may vary in their specific architecture and hardware resources, they generally share a common high-level structure resembling a finite-state machine. This similarity suggests that ParserHawk's techniques can be generalized to a broader range of parser architectures (e.g., Pensando DPU, Bluefield DPU).

Our contributions can be summarized as follows:

- Identify parser compilation as a critical problem, including describing scenarios in which existing compilation algorithms will fail to generate good parsers.
- ParserHawk a retargetable, program-synthesis-based compiler that can perform parser compilation across multiple heterogeneous parser architectures.
- Domain-specific optimizations to accelerate program synthesis.
- Evaluation of ParserHawk against state-of-the-art parser compilers, with an analysis of where and why ParserHawk can generate better results.

We have open sourced ParserHawk along with instructions to replicate this paper's results at https://github.com/ParserHawk/ParserHawk. *This work does not raise any ethical issues.*

2 Background

2.1 Parser functionality

The parser sits at the beginning of a programmable network device, responsible for preparing data for downstream processing. It converts the raw input bitstream into a structured format, like a vector of fixed-length containers storing values of packet fields, for later modifications in the packet-processing pipeline. A parser operates as a finite state machine (FSM). Each state of this FSM performs actions such as extracting bits out of the bitstream to deposit into various packet header fields (e.g., the IP source address) and state transitions to move between parsing one header field and another one (e.g., from parsing Ethernet to parsing IP). The state transition logic checks the value of a transition key, which may include both already extracted packet header fields and bits that have not yet been extracted (called lookahead bits).

2.2 Emerging parser programs

With the broader developments around network programmability, several new customized network functions (e.g., malicious attack detection [37], congestion control [23, 28]) have been developed to meet the requirements of cloud providers. These programs usually require customized logic to identify header fields in the parser, leading to various protocols and their associated parsing logic.

In recent developments, cloud providers have begun collaborating with the finance industry. For instance, Google Cloud and Chicago Mercantile Exchange (CME) have recently initiated a partnership [3] to provide cloud-based, ultra-low-latency networking. Similarly, Myers et al. [38] provide a roadmap to build low-latency networks for algorithmic trading systems. These advancements demand new parsing logic capable of analyzing finance-related network packets. Within cloud providers' data centers, it is essential to identify the origin of a network packet (e.g., internal users, financial exchanges) before routing it to the packet-processing pipeline. Such developments lead to larger and more complex parser programs.

2.3 Parser code generation

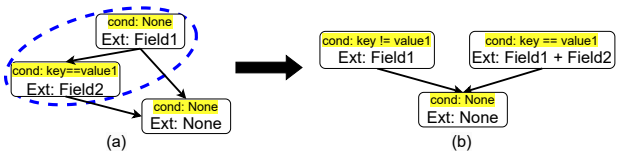


Figure 1: (b) uses one fewer TCAM entry than (b) by clustering 2 states.

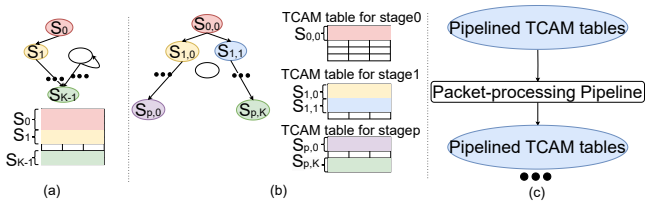


Figure 2: Heterogeneous parsers in programmable network devices: (a) has a single TCAM table. (b) has pipelined TCAM tables. All parser nodes from the same stage (e.g., $S_{1,0}$, $S_{1,1}$) can only access TCAM tables allocated to that particular stage (e.g., TCAM table for stage 1). (c) interleaves pipelined-TCAM-table parser and the packet-processing pipeline.

Current parser compilers are still quite elementary, as they can either falsely reject an input program or produce suboptimal compilation results. To the best of our knowledge, Kangaroo [35] and Gibb et al. [33] are two of the leading works on efficient parser generator design. Gibb et al. consider more hardware constraints (e.g., window size to fit the state transition key) than Kangaroo. Compared to these projects, commercial compilers from vendors incorporate basic heuristics for parser generation, lacking advanced optimization techniques.

Usually, one TCAM entry encodes a single parser state transition. If two adjacent parser states can be merged into a single state, the internal transitions between them no longer require separate TCAM entries, potentially reducing the overall TCAM entry usage. Therefore, Gibb et al. propose a dynamic programming (DP) approach by clustering adjacent parser states to perform code generation for parsers. Each cluster should follow the hardware resource constraints in entries of the TCAM table (e.g., one transition arrow "→" in Figure 1 uses one TCAM entry). For instance, Figure 1(b) uses 1 TCAM entry fewer than Figure 1(a) by clustering 2 adjacent states. This DP algorithm explores a good clustering strategy that minimizes the TCAM entry usage.

Gibb et al.'s parser generator, while effective at the time of its proposal, has become outdated due to advancements in programmable devices. It restricts the transition key selection from each state from its extracted fields and does not support lookahead. Moreover, its target parser architecture is limited to one particular device, making it incompatible with newer devices with different architectures. Additionally, some of its generated outputs result in suboptimal hardware resource utilization. §7 shows detailed results.

3 Parser Hardware Model

3.1 Programmable parser architecture

A parser identifies header fields from an input bitstream. These fields are updated later by the packet-processing pipeline (e.g., IP.TTL = IP.TTL-1). A parser compiler translates the parser specification written by developers into a hardware-compatible format that integrates with the target device's architecture. There are 3 typical parser configurations for line-rate programmable parsers.

Single TCAM table: Devices such as the Tofino switch use ONE TCAM table (Figure 2(a)) to fit the whole parser. Such a design allows one to visit an entry as many times as one wants. For example, suppose that there is a header field (e.g., MPLS) appearing multiple times in a packet and that we want to iterate over all occurrences of that header. In this case, no matter how many instances of the header field, a *SINGLE* entry can advance over one instance of the header field and loop back to itself until we have parsed all instances. But, the next packet cannot enter the parser until the parsing of the previous packet completes.

Pipelined TCAM tables: Devices such as the *Intel IPU* chain a series of TCAM tables in a pipelined structure. There is *ONE TCAM table* per pipeline stage (Figure 2(b)). All parser nodes within a stage can only use TCAM entries allocated to this stage. Because of the pipelined design, it can process a new packet every cycle, improving parsing throughput. However, due to the inability to "loop around" like the previous architecture, we cannot fit a long-depth parser that consumes more than the available stages of the target device.

Interleaving between parser and pipeline: Devices such as the *Broadcom Trident series* support jumping out of the parser to modify packet fields in the packet-processing pipeline and returning to the parser (Figure 2(c)). This can be regarded as an interleaving between subparser components and packet processing pipeline. In Trident, each subparser consists of a sequence of TCAM tables arranged in a pipeline. These devices allow more expressive parsing behavior because the pipeline can execute complex packet field updates to affect the parsing behavior afterwards.

3.2 Motivating example

To run a parser in the target network device, we need to rely on the parser compiler to do low-level code generation (i.e., fill in various TCAM entries). Existing compilers often generate code that inefficiently utilizes hardware resources. We demonstrate the suboptimality through 2 cases.

```
state N0: transition select(tranKey): // tranKey is 4-bit
            0b1111, 0b1011, 0b0111, 0b0011 : N1;
            0b1110 : N2;
            0b0010 : N3;
            default: accept;
```

Figure 3: Parser specification program used for code generation in Figure 4.

3.2.1 Suboptimality from parser generation algorithm. Figure 4 shows 2 ways (V1 and V2) to generate a parser implementation for the input parser program provided in Figure 3. Briefly, this parser program has multiple state transition rules: if the 4-bit state transition key has a value in {15, 11, 7, 3}, the parser transits to state N1. If the value is 14, it goes to state N2; if the value is 2, the next state is N3. For all other values, the parser accepts the bitstream and exits. There are 2 target devices (A and B) with different constraints for the state transition key. Device A can at most fit a 2-bit field in the state transition entry of the TCAM table while device B can fit at most a 4-bit field.

The code generation process consists of 2 steps. Step1 merges multiple state transition rules using the mask (m) and value (v) as long as $key\&m == m\&v$. Step2 splits the key into subfields to fit into the hardware's constraints if needed.

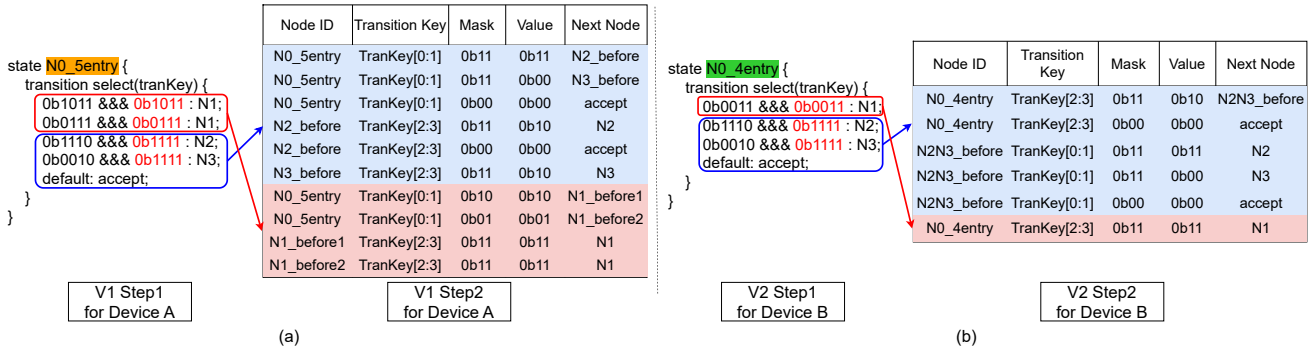


Figure 4: A motivating example demonstrating how different parser generation algorithms (V1 and V2) can produce outputs with varying hardware resource consumption. Such difference originates from suboptimal entry merging (Step 1) and transition key splitting (Step 2) strategies.

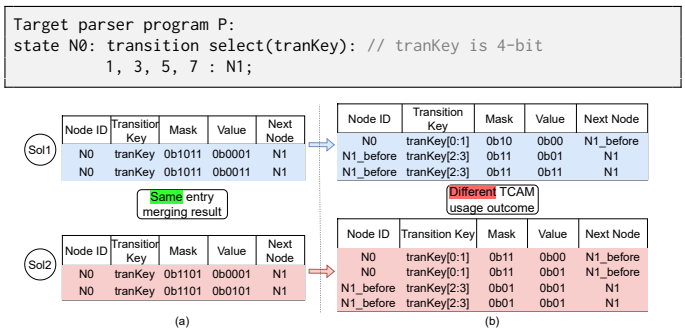


Figure 5: For parser program P, Sol1 and Sol2 generate merging results with the same number of entries in (a) but lead to different TCAM usage in (b) for devices that can only start key+value matching from the current extraction cursor.

There are multiple ways in step1 to choose the mask + value combination, each of which might consume different # entries. Some existing approaches [33] develop rule-merging algorithms. Unfortunately, they might lead to a suboptimal result (V1 Step1 in Figure 4(a)), while we can find a better merging solution that only needs 4 entries (V2 Step1 in Figure 4(a)). Due to differing bit-width constraints on the state transition keys of devices A and B, the output of step1 is sufficient to fit into device B. However, if we want to generate a parser for device A whose state transition keys can only be ≤2-bit, we need to do key splitting in step2. There are multiple ways (blue entries in Figure 4) to do key splitting for the same state transition rules (code in blue rectangle in Figure 4). Each way uses a different number of TCAM entries.

Two main reasons lead to this result. Firstly, redundant mask + value combinations (in the red rectangle) require more TCAM entries from the hardware device. Secondly, even for the same mask + value combination (in the blue rectangle), different orders to check values of the state transition key (e.g., check tranKey[0:1] before or after tranKey[2:3] in Figure 4) may lead to different TCAM entry usage. These reasons cause 6 vs 10 TCAM entry usage from 2 parser generation solutions.

3.2.2 Suboptimality from decoupling compilation into separate phases. To make things even more complicated, even if two entry merging algorithms produce the same number of mask-value pairs to cover the state transition rules (Figure 5(a)), their resulting TCAM usage can be different (Figure 5(b)). These issues could be mitigated by adding more compiler rewrite rules, but similar cases may still arise where different ways of expressing the same semantics lead to varying hardware resource usage. The fundamental reason for suboptimality is that the generation algorithms depend on the specific written style of the input parser program. As a solution to such brittleness, we claim that ***parser generation is a combinatorial problem and that a combinatorial search engine should be used to solve this problem.***

3.3 Program-Synthesis-based approach

This inefficiency of existing parser generators may be tolerable when hardware resources for the parser are relatively abundant. Unfortunately, this is not the case. Most programmable network devices have severely constrained hardware resources (e.g., # TCAM entries in Tofino, # parser stages in IPU) to support the high-speed and high-throughput performance. These resource limitations, combined with the inefficiencies of existing compilers, often result in compilation failures for parsers with complex parsing logic and multiple parser states. Such parsers, commonly developed by cloud providers, are designed to analyze network packets from diverse sources (e.g., financial exchanges, universities, internal servers).

We believe a program-synthesis-based solution is a good fit to remedy these drawbacks because it takes into consideration *ONLY* the input parser's semantics rather than the code's written style. Accordingly, we turn the compilation into a combinatorial search problem and utilize state-of-the-art solvers [26] to generate a semantically equivalent parser implementation by solving this search problem.

4 Problem Statement

ParserHawk produces a parser implementation for a specification written in a high-level language. A parser takes a bitstream as input and identifies header fields through a series of parsing steps. The

final output of a parser is a dictionary that maps packet fields to corresponding values.

```
def Impl(I):
  // starting table (TID) and state (SID)
  TID = 0, SID = 0, OD = {}
  // each iteration is one state transition
  // there are at most K iterations
  for i in range(K):
    // find all entries
    for EID ∈ GetEntry(TID, SID)
      // when we first match a condition
      if Condition(TID, SID, EID)
        // extract relevant bits into output dictionary
        for h ∈ ExtractSet(TID, SID, EID)
          I, OD = extract(h, I, OD)
        // transition to next state
        TID, SID = Tran(TID, SID, EID)
        break // start next iteration after the first
              successful match
  return OD
```

Figure 6: Generic implementation of a parser using TCAMs for state transitions. Once the parser state (SID) reaches accept or reject, the SID and TID do not change for the remaining transitions of the parser.

Correctness. ParserHawk's goal is to generate parser implementations that match the behavior of the specification. An implementation is correct if, on all input bitstreams, the dictionary it outputs is equal to the corresponding specification's output. Formally, let *Spec* be a function from bitstrings to dictionaries representing the specification of a parser, and let *Impl* be the corresponding function for the implementation. Then, *Impl* is correct if $\forall I \in \{0,1\}^*$, $Impl(I) = Spec(I)$.

Implementation Behavior. The output of ParserHawk is a set of rows describing the entries in TCAM tables that implement the parser. Referring Figure 6, each entry row is identified by a TCAM table id $TID \in \mathbb{N}$, a state id $SID \in \mathbb{N}$, and a entry ID $EID \in \mathbb{N}$. An entry row has a *Condition* field, which is a logical predicate that defines when the entry in that row is active. The *ExtractSet* of a row returns the set containing all the packet fields to extract when that row is active. Finally, the *Tran* field gives the TCAM table id and state number to transition to after a row is activated.

Given such a collection of rows, we can define a function *Impl* that describes how the parser corresponding to those TCAM tables executes. Figure 6 shows pseudo-code that defines this function by simulating the device's parsing behavior by visiting TCAM tables for up to K times.[1] The code builds up a dictionary OD mapping packet fields to their values. Initially, the parser starts in TCAM table 0 and state 0. On each step, the parser looks up all of the entries associated with the current table and state ID using *GetEntry*, and then checks if the condition associated with that entry is active. If so, it extracts each field associated with the entry into the mapping OD, and then performs the corresponding transition. Finally, the parser returns the updated OD.

Example. We use 2 examples to show the concrete function of ***Parser Program+Spec*** (Figure 7) and ***Impl*** (Figure 6) with the Tofino switch as the target device. They both assign the first 4 bits

[1] K is a parameter of our synthesis procedure.

```
Spec1.p4:bit<4>field0, field1
  State0:
   pkt.extract(field0)
   transition State1;
  State1:
   pkt.extract(field1)
   transition accept;
```

```
Spec2.p4:bit<4> field0, field1
  State0:
   pkt.extract(hdr.field0)
   transition select(field0[0])
       0 : State1;
       default : accept;
  State1:
   pkt.extract(field1)
   transition accept;
```

```
def Spec1(I):
  OD = {}
  OD[field0] = I[0:4]
  OD[field1] = I[4:8]
  return OD
```

```
def Spec2(I):
  OD = {}
  OD[field0] = I[0:4]
  if field0[0] == 0:
    OD[field1] = I[4:8]
  return OD
```

Figure 7: From parser in P4 to spec in ParserHawk.

Table 1: Suppose $field_0$ and $field_1$ are 4-bit variables. Red cells for *Impl*2 parser only and all other cells are for both *Impl*1 and *Impl*2. "Accept" value in the Tran column means completion and exiting the TCAM table. For these 2 simple programs, we can refer to the *Spec* in Figure 7 as *Impl*.

	Condition	ExtractSet	Tran
TID:0, SID:0, EID:0	True	{field_0}	0,1
TID:0, SID:1, EID:0	field_0[0] == 0	{field_1}	Accept
TID:0, SID:1, EID:1	field_0[0] ≠ 0	{}	Accept

to $field_0$. *Spec*1.*p*4 sets $field_1$ to be the next 4 bits unconditionally, while *Spec*2.*p*4 does so only if the first bit of $field_0$ is 0.

The job of the parser generator is to concretize ***Condition***, ***ExtractSet***, and ***Tran*** for all entries in all TCAM tables. When targeting the Tofino switch, which only has a single TCAM table, the value of TID should always be 0. Using the values in Table 1, ***Impl*1** achieves the same functionality as ***Spec*1** while ***Impl*2** is the same as ***Spec*2**.

5 Parser Synthesis Procedure

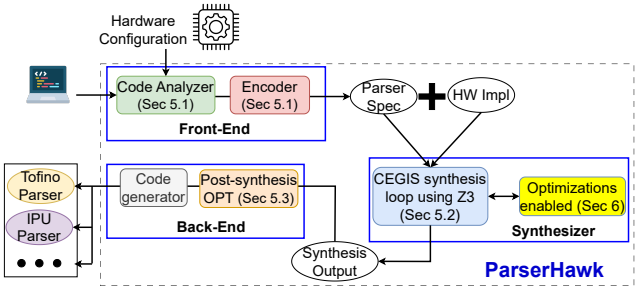


Figure 8: ParserHawk workflow overview.

The whole workflow of ParserHawk is shown in Figure 8. Concretely, the front-end first processes the input parser description and hardware configuration through a Code Analyzer and Encoder (§5.1), and then generates one parser specification and the corresponding hardware implementation. Both of them are fed into a synthesizer (§5.2) written by Z3 with all speedup optimizations (§6)

enabled to generate an output. The synthesis output is then passed to the back-end, containing a Post-Synthesis Optimizer (§5.3) and a Code generator, to output target-specific implementations (e.g., for Tofino or IPU in this paper). The core part is the synthesizer and its optimization algorithms.

ParserHawk uses a program synthesis technique called CEGIS (Counterexample-Guided Inductive Synthesis [40]) to generate parser implementations. The CEGIS process consists of 2 phases: synthesis and verification. During the synthesis phase, ParserHawk uses the Z3 SMT solver to generate a candidate set of TCAM rows that satisfy selected input/output examples, where the input is a bitstream and the output is a dictionary mapping headers to values. In the verification phase, Z3 checks whether the candidate matches the specification parser for *all possible inputs*. If it matches, ParserHawk returns the candidate as a valid solution. Otherwise, Z3 will generate a *counterexample* in which the candidate and the specification disagree. In that case, the synthesis phase repeats and the counterexample is added to the set of input/output examples used during the synthesis phase. The CEGIS loop repeats until either a solution is found or Z3 indicates that no solution exists (e.g., no set of TCAM tables with the specified size and resource limits can correctly match the specification for the current set of input/output examples). The challenge in making this CEGIS approach work lies in finding an encoding in Z3 that can be solved in a practical amount of time.

5.1 Encoding parser implementation and specification

A parser is a finite state machine (FSM) where each state include operations (e.g., packet field extraction) and state transition rules as specified in TCAM entries. ParserHawk encodes this FSM behavior as a series of logical formulas and constraints in Z3. For this encoding, ParserHawk strives for *high-quality* parser implementations and *retargetability*.

High quality here means that, given developers' requirements, ParserHawk should generate a parser implementation using as few resources (e.g., TCAM entries, pipeline stages) as possible. Fewer resource usage can leave more room for future incremental changes to the network parser [39].

To support retargetability to multiple backend devices, ParserHawk divides up the parser encoding into generic rules for how TCAM tables behave and target-specific details. Concretely, our encoding is split into 2 parts, (1) generic FSM simulation rules, which describe how the state transition system should evolve as TCAM entries are processed; and (2) a hardware configuration profile, which encodes constraints imposed by a particular hardware device. To re-target ParserHawk to support new hardware devices, a developer only needs to write formulas for a new hardware configuration profile.

5.1.1 Generic FSM Simulation.

ParserHawk defines a collection of Z3 variables (Table 2) to represent the parser state. The formula ϕ_{common} shown in Figure 9 simulates the parser execution process, essentially encoding the pseudo-code from Figure 6 as a set of logical formulas. The formulas encode a process of visiting the TCAM tables for multiple iterations, with one parser state executing per iteration. In each iteration l, these formulas track (1) which parser state is executing, (2) which fields are extracted, and (3) how to do state transition.

```
// Initialization condition
CurrID[0] == 0 ∧ pos[0] == 0
// Packet field extraction in iteration l
∧_{j,l,k} (CurrID[l] == k ∧ Extract[j][k] == 1) ⇒
∧_{f≠field_j} (OD[l][f] == OD[l+1][f]]) ∧ OD[l+1][field_j] ==
I[pos[l] : pos[l] + sz(field_j)] ∧ pos[l+1] == pos[l] + sz(field_j)
// Build state transition key for parser state k
∧_{i,j,k} Alloc[i][j] == k ⇒ field_i[j] ∈ Trankey[k]
∧_{l,j,k} Lookahead[k][j] == 1 ⇒ I[pos[l] + j] ∈ Trankey[k]
// Do state transition
∧_{l,i,k}(Dstate_i == k ∧ Trankey[k]&M_i == V_i&M_i) ⇒ CurrID[l+1] ==
    Nxt_i
```

Figure 9: Logical formulas for common hardware constraints (ϕ_{common}). $Dstate_i$ represents the state ID in ith iteration.

5.1.2 Hardware configuration.

ParserHawk additionally uses a hardware-specific configuration to encode constraints imposed by different devices. For example, the current profiles in ParserHawk encode the following kinds of constraints:

Extraction length limit. This constraint imposes a limit on the maximum number of bits to extract per entry. For large-size packet fields (e.g., 40-byte IPv6 header), more than one entry may be needed to complete the extraction of the entire field.

State transition key size limit. The size of a state transition key per entry in the TCAM table may have limits. For example, if the limit of the state transition key is 2 bits, a 4-bit transition key must be split into 2 pieces, using >1 TCAM entries to complete the transition.

Lookahead window size. A parser may look ahead to check bits that have not yet been extracted when deciding how to transition. How far a given parser can check is determined by the lookahead window size. Its value varies by device.

Limited number of TCAM entries and stages. The number of TCAM entries and parser stages are limited. ParserHawk must generate an output within the available resources.

ParserHawk use variables $keyLimit$, $tcamLimit$, $lookaheadLimit$, and $stageLimit$ as constants for the maximum # of bits for state transition key, max # of TCAM entries, max lookahead window size, and max # of stages in parser, respectively. Figure 10 and 11 present the high-level formulas encoding the hardware resource constraints for Tofino and IPU. The formula for the implementation's behavior becomes $Impl(I, Device) = \phi_{common} \wedge \phi_{Device}$. Appendix 13 shows more details involved in turning these constraints into the first-order logic formula. We do not constrain the extraction limit in synthesis and leave it to the post-synthesis optimization (§5.3).

```
// Do not use a too-wide state transition key
∧_{k,i,j} Sum(Alloc[i][j] == k) + Sum(Lookahead[k][j]) ≤ keyLimit
// Do not use more than available TCAM entries
∧ len(TCAM) ≤ tcamLimit ∧
// Do not exceed lookahead window size
∧_k len(Lookahead[k]) ≤ lookaheadLimit
```

Figure 10: Hardware constraints for Tofino (ϕ_{tofino}).

Table 2: Z3 variables used to encode parser behavior. They are used to build formulas in Figure 9, 10, 11, and 12

Name	Type	Description
$Extract$	2D array of Bool	$Extract[i][j] = 1$ means $field_i$ is extracted at parser state j.
$Trankey$	1D array of Int	$Trankey[k]$ represents the state transition key in parser state k.
$Alloc$	2D array of Int	$Alloc[i][j] = k$ means the jth bit of $field_i$ is included in $Trankey[k]$.
$Lookahead$	2D array of Bool	$Lookahead[k][j] = 1$ means the jth bit ahead of the extraction pointer is in $Trankey[k]$.
$TCAM$	1D array of Int×Int×Int	$TCAM[i]$ has 3 variables: M_i for mask, V_i for value, and Nxt_i for the next parser state ID.
$Dist$	1D array of Int×Int	$Dist[i]$ has 2 variables: $Dstage_i$ and $Dstate_i$, giving the stage and parser state ID entry i is allocated to.
$CurrID$ and pos	1D array of Int	$CurrID[l]$ and $pos[l]$ record the parser state ID and position in the l^{th} (out of a max K) iteration.
OD	1D array of dictionary	$OD[l][f]$ stores the value of packet field f in the l^{th} (out of a max K) iteration.

$$\wedge_{k,i,j} Sum(Alloc[i][j] == k) + Sum(Lookahead[k][j]) \leq keyLimit$$
$$\wedge\ len(TCAM) \leq tcamLimit\ \wedge$$
$$\wedge_k len(Lookahead[k]) \leq lookaheadLimit$$
$$\wedge_i DStage_i \leq stageLimit\ \texttt{//New1:Only use available stages.}$$
$$\wedge_i DStage_{Nxt_i} > DStage_i\ \texttt{//New2:Move forward to future stages.}$$

Figure 11: Hardware constraints for IPU (ϕ_{IPU}). $Dstage_i$ represents the stage number in ith iteration.

5.1.3 Specification encoding. As part of the CEGIS verification phase, Z3 also needs an encoding of how the specification for the parser behaves. To describe specification behavior, ParserHawk reuses a similar encoding to describe an FSM as in 5.1.1, but instead of deriving transitions and extraction rules based on an unknown (symbolic) TCAM encoding, it directly uses values extracted from the specification parser program. Figure 12 shows the additional formula encoding the specification's behavior. The values of the Val arrays used in this template describe the specific behavior of the specification and are derived from the specification parser program.

```
// Concretize all Z3 variables using ValX arrays
∧_{i,j}(Extract[i][j] == ValE[i][j]) ∧
∧_{i,j}(Lookahead[i][j] == ValL[i][j]) ∧
∧_{i,j}(Alloc[i][j] == ValA[i][j]) ∧
∧_i(Dstage_i == ValDstage[i]) ∧ ∧_i(Dstate_i == ValDstate[i]) ∧
∧_i(V_i == ValV[i]) ∧ ∧_i(M_i == ValM[i]) ∧ ∧_i(Nxt_i == ValN[i])
```

Figure 12: Constraints specification (ϕ_{spec}).

The formula for the specification's behavior is defined as the conjunction formula: $Spec(I) = \phi_{common} \wedge \phi_{spec}$.

5.2 CEGIS loop in ParserHawk

ParserHawk implements the CEGIS loop in Z3 (in Figure 13) to find one parser configuration $Conf$ that is semantically equivalent to the specification's behavior. For a particular field, if it is updated in the parser by extracting a subrange of the input bitstream, then its value should be correctly reflected as the value of the bitstream portion; otherwise, its value remains unchanged.

Synthesis: Find $Conf$ *s.t.* $\forall I \in TestSet,\ Spec(I) == Impl(I, Device)$.
Verification: Given a $Conf$, whether $\exists I \in AllInput$ *s.t.* $Spec(I) \neq Impl(I, Device)$.

We maintain a test case set ($TestSet$), containing input-output pairs used for the synthesis phase. Each element of $TestSet$ includes (1) an input bitstream I_i and (2) all packet fields' values after parsing the input bitstream (stored in a list $\{V_{i,0}, .., V_{i,N-1}\}$). We use Python

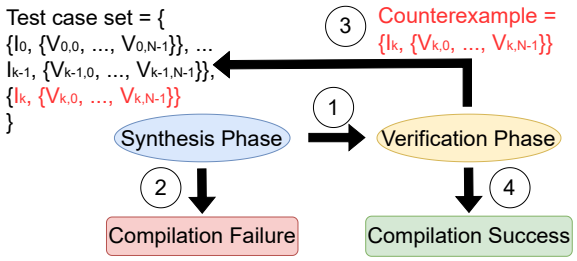


Figure 13: CEGIS loop. ①: go to verification if synthesis finds a solution; ②: compilation failure if synthesis fails to find a solution; ③: return to synthesis by adding the counterexample to the test case set; ④: compilation success if no counterexample is found in verification.

execution to simulate the parser's behavior to get the parsed values. We use $AllInput$ to represent the whole input space. ParserHawk generates an implementation ($Conf$) for $TestSet$ in the *synthesis phase* and verifies this implementation by finding a counterexample in the *verification phase*.

ParserHawk initializes the test case set for synthesis with one randomly generated input-output pair for simplicity. The verification phase checks whether the configuration output from the synthesis phase satisfies all examples within the input space. If there exists a counter-example, ParserHawk adds it to the test case set and re-enters the CEGIS loop.

5.3 Post-synthesis optimization

During the synthesis phase, ParserHawk includes additional restrictions on implementations in order to reduce the search space. For example, ParserHawk disallows one parser state from extracting >1 packet field without constraining the extraction length limit in the synthesis phase(§5.1.2). This simplifies the synthesis procedure because otherwise the synthesis engine would have to consider not only which fields to extract, but also the order to extract them in each parser state.

However, these restrictions, while speeding up synthesis, can lead to potentially more parser state usage than necessary. Therefore, ParserHawk has a post-synthesis optimization phase that improves the output from the CEGIS loop. The optimizations in this phase are simple to perform, but trying to include them in the synthesis phase would be inefficient. For example, ParserHawk recursively merges parser states that do field extraction and have

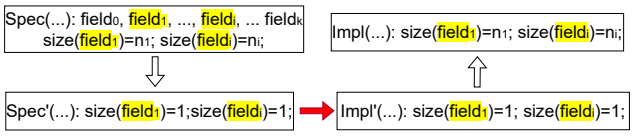


Figure 14: Spec→Spec': scale down all irrelevant fields' size to 1. Spec'→Impl': using program synthesis; Impl'→Impl: Scale up those fields' size to match original size.

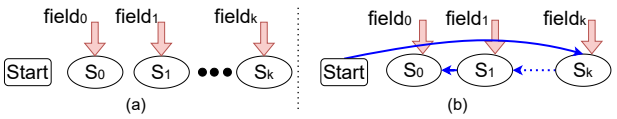


Figure 15: First, ParserHawk pre-allocates packet fields to different parser states (a). Next, ParserHawk determines the state transition logics (→) using program synthesis (b). This works because different parser states are symmetric from the hardware's perspective.

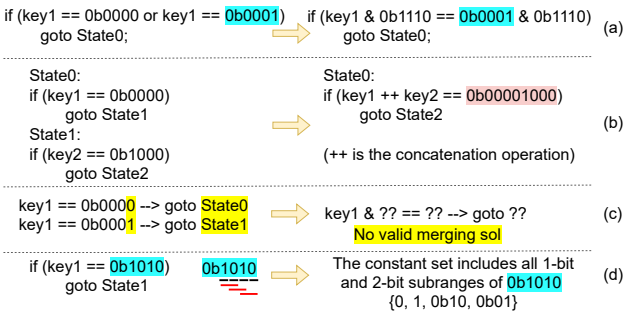


Figure 16: (a) uses values existing in parser spec to merge entries (b) concatenates constants in adjacent parser nodes for value selection (c) cannot merge multiple entries if they transit to different parser states (d) collect all hardware-compliant subranges of the wide key into constant sets.

only 1 default state transition rule with their adjacent states. ParserHawk also will divide a parser state that extracts a large-size packet field into multiple ones.

6 Optimizations in ParserHawk

Naively encoding parser generation as a program synthesis problem is quite time-consuming due to the exponential growth of both the search space and input space. The root cause is that the naive encoding produces a very large number of variables and constraints.

These variables include symbolic constants, which specify the conditions used in state transitions (e.g., value-mask pairs), and structural variables, which define the parser's architecture, including field-to-state assignments and next-state selection. For instance, each TCAM entry contains a value-mask pair, consisting of two symbolic constants of key width (KW) bits, resulting in 2^{KW} possible solutions per constant. As the number of TCAM entries grows, so does the number of symbolic constants, causing the search space to expand exponentially.

Other symbolic structural variables in the parser exhibit similar exponential growth as the implementation scales. Meanwhile, the input space is also a bottleneck: to guarantee correctness, the synthesized parser must produce the same updates as the specification for all combinations of input-output pairs. As the parser specification becomes more complex, the input bitstream length increases, leading to an exponential blow-up in the number of possible input values. We see that with a naive encoding, the compiler fails to generate code for some benchmarks within 24 hours (§7). Therefore, it is necessary to speed up compilation by leveraging several optimization algorithms.

6.1 Opt1: Spec-guided Key Construction

We optimize the state transition key construction by restricting the choice of bits to those *explicitly* used in the specification for state transitions. For instance, if only the first 4 bits of a packet field (e.g., $field_0[0{:}3]$) are used for the state transition key in the specification, we avoid selecting other bits besides those 4 bits to build transition keys in the implementation. This significantly reduces the search space because typically around 1% of the bits of all fields are relevant for state transition.

6.2 Opt2: Bit-Width Minimization

We regard packet fields whose bits are not used in any state transition keys in the specification as *irrelevant fields*. As shown in Figure 14, this simplifies the representation of these irrelevant packet fields by treating their sizes as 1 bit rather than their actual sizes in ParserHawk. This compaction effectively reduces the size of the input bitstream, thereby shrinking the input space and simplifying the synthesis problem. After synthesis, the sizes are replaced with their proper values.

6.3 Opt3: Preallocated field extraction

This optimization leverages the symmetric properties of parser states. By analyzing the input parser program, the algorithm only considers packet fields that are explicitly parsed by at least one parser state. Instead of relying on the synthesis solver to decide which state is responsible for extracting each field, we pre-designate certain states for extracting each field (e.g., $field_0$ is extracted in S_0, $field_1$ in S_1 in Figure 15). This exploits the symmetry in parser states, as their execution order is not fixed (e.g., the execution of S_0 can appear before or after S_1). The synthesis task is determining the execution order (e.g., $S_k \rightarrow \ldots \rightarrow S_1 \rightarrow S_0$) among parser states, narrowing the search space size. *Opt3* is ONLY applicable to parser architectures with such symmetric features (e.g., single-TCAM-table style parser).

6.4 Opt4: Constant synthesis

Symbolic constants contribute to the large search space in synthesis. To address this, we employ several domain-specific constant synthesis algorithms to guide the solver toward correct solutions.

6.4.1 Restricting the search space for value selection. ParserHawk synthesizes value-mask pairs that can merge multiple TCAM entries. For example, if $\forall i = 1...N$, $V \,\&\, M == A_i \,\&\, M$, ParserHawk

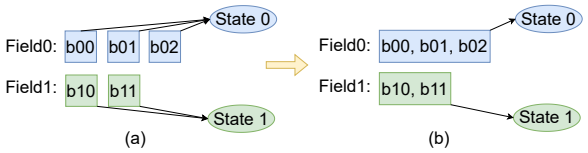


Figure 17: (a) assigns each individual bit to state transition keys across parser states while (b) groups bits from the same field and allocates each group to the same state.

can use one entry/value-mask pair (V, M) to cover $N = 2^n$ entries (e.g., A_1, $A_2 \ldots A_N$). If such a (V, M) exists, then any of the value-mask pairs (A_i, M) are valid as well. One toy example is shown in Figure 16(a). Thus, rather than trying to search for such a V from the space of all possible values, ParserHawk only considers the constant values $A_1, \ldots, A_n$ present in the parser specification, reducing the domain size considerably (e.g., $2^{32} \rightarrow 100$ constants).

While this heuristic significantly speed up synthesis, it may introduce suboptimality. Figure 16(b) shows that it can miss opportunities to reduce TCAM entries by merging state transition rules across adjacent parser states. To mitigate this, we include concatenations of values in adjacent parser states for ParserHawk to search symbolic constant values, thereby recovering better solutions in some cases.

6.4.2 Restricting and parallelizing the search for masks. The previous optimization reduces the search space for the value component of value-mask pairs, but it is also possible to speed up the search for masks as well. Notably, ParserHawk only needs to merge multiple TCAM entries when they transition to the same next parser state. For instance, there is no way to merge entries in Figure 16(c) because of different next parser state values. To exploit this, in the case where all entries of the specification transition to distinct next parser states, we simply set all mask bits to 1. For parser specifications that can do entry merging via mask+value, we can leverage a server pool to parallelize the search. Conceptually, each server in the pool solves a subproblem by fixing the value of a subset of mask variables to all-ones (e.g., 0b111...1), while letting Z3 synthesize the remaining mask variables. For instance, one subproblem may let Z3 determine the value of one mask, another allows two, and so on.

6.4.3 Splitting large state transition keys' values. When the parser specification contains constants exceeding the allowable key width, they cannot be directly used in the implementation. For instance, if the target hardware of Figure 16(d) can support at most 2 bits in its state transition key, it is not helpful to include 0b1010 in the set to search for the symbolic constants' value. We develop specialized algorithms to address this issue. Specifically, for a large-bit constant C shown in the specification, we generate all candidate constants that (1) are subranges of the original constants and (2) fit within the hardware-imposed width limit (KW). Algorithmically, we take all constants of the form $C[i, j]$, where $0 < i \leq j$ and $j - i \leq$ KW. and added them to the solver's constant set. This optimization reduces the constant search space from exponential (2^{KW}) to quadratic (O(KW*len(C)).

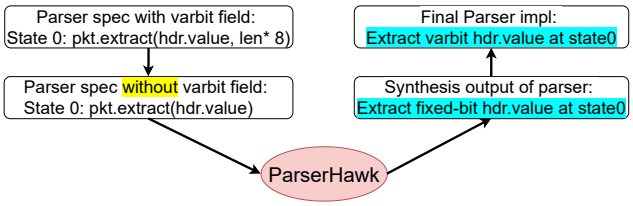


Figure 18: All varbit fields are treated as fixed-bit in synthesis. They are then converted back to varbit fields in the final implementation.

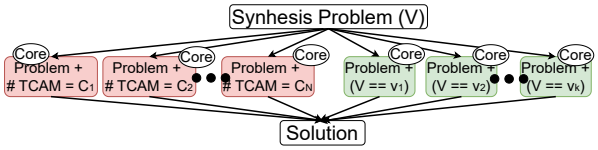


Figure 19: Derive multiple synthesis tasks by fixing the hardware resources (e.g., tran key size limit) or values of selected Z3 variables or both, and run these tasks in parallel (one core per job).

6.5 Opt5: Grouping for transition key allocation

ParserHawk determines which bits should be used as state transition keys at each parser state. Each bit can be assigned to any parser state, and the total search space grows exponentially with # bits considered. However, in many examples, we observe that bits from the same packet field are usually used together for state transitions within a single parser state, unless the field is intentionally split across multiple states. Hence, as illustrated in Figure 17, we group adjacent bits from each field and treat them as an indivisible unit during synthesis. All bits within a group are allocated to the same parser state.

6.6 Opt6: Treat all fields as fixed-size during synthesis

There are 2 types of fields: fields whose sizes are determined at compile time (called FixField) vs at run time (called VarField). This introduces complexity in behavior encoding for extracting VarField, as we must account for all possible sizes of the VarField and update the extraction pointer accordingly. In practice, ParserHawk only needs to determine which parser state extracts a particular field, not how many bits to advance. This decision is independent of the VarField's actual size. We leverage this observation by treating VarFields as if they had a fixed size during the synthesis phase (Figure 18), thereby unifying their treatment with FixFields. During a post-synthesis phase, we convert the synthesized output into a format that correctly extracts VarFields.

6.7 Opt7: Parallelism

Based on specific input parser behavior, we can divide a large synthesis problem into several subproblems and distribute them across a server pool for parallel synthesis.

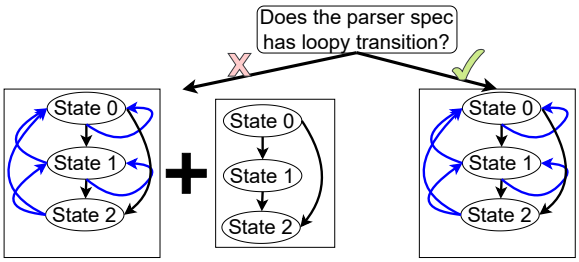


Figure 20: When the parser spec contains no loopy state transitions, we run synthesis for both loop-aware and loop-free implementations in parallel, hoping that the loop-free version can complete faster due to its smaller search space.

6.7.1 Loop-Aware vs. Loop-Free Parser Implementation. The TCAM table in devices such as Tofino allows revisiting the same entry multiple times, which can reduce TCAM entry usage by reusing entries. However, other devices such as IPU do not support such reuse. Therefore, we support both loopy and non-loopy parser generation in ParserHawk. The flexibility of visiting an entry multiple times makes the search space of loopy parser implementation larger. As is shown in Figure 20, for loop-free parser specifications where no input revisits a TCAM entry, the loopy design could be ineffective in reducing TCAM usage. In such cases, we adopt both loopy and non-loopy parser implementation in parallel with the hope that the non-loopy parser can generate a result faster due to its smaller search space.

6.7.2 Parallelism in solution exploration. Z3 has built-in support for trying to solve a query in parallel, but the parallel solver mode does not automatically improve performance on our queries [9, 12]. Instead, we manually leverage the multiple cores to run the code generation in parallel (Figure 19). On the one side, we ask ParserHawk to do compilation under different levels of hardware constraints. For example, assuming the target device supports a 32-bit state transition key size, we derive several subproblems by setting the limit from 32 bits down to 1 bit, and solving them together (1 core per subproblem). Smaller state transition key sizes allow the synthesis solver to find masks or values with fewer bits, simplifying the overall synthesis process. On the other side, we divide the code generation problem into several subproblems by assigning values to selected Z3 variables (e.g., setting a Z3 boolean variable to 0 in one subproblem and 1 in the other). This reduces the search space because the Z3 solver only needs to find the values of the remaining variables. With this strategy, ParserHawk can halt as soon as one subproblem yields a valid outcome or if all subproblems fail.

7 Evaluation

We evaluate ParserHawk and other baseline compilers by answering several questions.

- **Compilation completeness and correctness:** If there exists a solution, can a compiler explore that solution? Is the compilation outcome semantically correct?
- **Compilation speed:** How long does it take to get the compilation outcome?
- **Resource consumption:** How many hardware resources does each compiler's output consume?
- **Retargetablity:** Can these compilers support compilation for various programmable network devices?
- **Optimization impact:** How beneficial are the proposed optimizations in ParserHawk quantitatively?

Baseline selection. We compare ParserHawk against the parser generator in Gibb et al. [13], the open-sourced Tofino compiler [19], and close-sourced Intel IPU compiler. A research prototype developed by Gibb et al. leverages a dynamic programming (DP) algorithm to do parser generation and we call it **DPParserGen**. Its hardware constraints are parameterized and therefore this allows us to compare compilation results under various hardware configurations. Note that the DPParserGen can *ONLY* target single-TCAM-table parser architectures and has restrictions to represent input parser program. For example, DPParserGen disallows mask+value combination/wildcard match (e.g., `0b1**0: N1;` where 0b1**0 matches any 4-bit values starting with 1 and ending with 0) and disallows transition to the accept state on specific value (e.g.,`0: accept;`) in the parser's state transition entry. The state transition key in each state of DPParserGen must come from the field extracted in that state. These constraints are partially due to the fact that the programmable parser was less expressive before. Therefore, we only run DPParserGen over benchmarks that can be represented by its format. By contrast, ParserHawk supports code generation for various parser architectures.

We also compare ParserHawk against a few commercial compilers that we have access to (e.g., Tofino and IPU compiler). Note that developers have also added some optimization algorithms into these commercial compilers to reduce the generated parser's resource usage. However, to our knowledge, these optimizations rely on a variety of heuristics. Our results show the benefit of including program synthesis within commercial compilers, allowing them to generate better compilation outcomes without relying on engineers' effort to rewrite parsers in semantically preserving ways.

Benchmarks. We evaluate 29 benchmarks [6] generated from various sources including realistic parser programs from Gibb et al. [33] and production parsers [4, 11, 15]. Some benchmarks are created by randomly selecting a subset of 2–9 parser states from switch.p4 [17], sai.p4 [15], and dash.p4 [4]. Besides, we create synthetic benchmarks to reflect particular parser patterns suggested in conversations with programmers of such parsers.

We further mutate these benchmarks using a combination of semantic-preserving rewrite rules (Figure 21), including ***+/-R1:*** add/remove redundant entries, ***+/-R2:*** add/remove unreachable entries, ***+/-R3:*** split/merge entries, ***+/-R4:*** split/merge transition key, and ***+/-R5:*** split/merge parser states. These rewrites are intended at capturing the parser development process. For instance, developers may be unaware of the target device's constraints and therefore write large-size state transition keys in parser programs.

Experiment setup. We run our experiments in CloudLab x86_64 c6620 nodes in the Utah cluster, using Ubuntu 22.04 and a 28-core 56-hyperthreads Intel Xeon Gold 5512U processor. Each synthesis task

Table 3: ParserHawk vs. Tofino and IPU compiler over several parser benchmarks [6] (OPT: all *Opt* enabled, Orig: all *Opt* disabled)

Program Name	ParserHawk (Tofino) # TCAM	Search Space (bits)	OPT time (s)	Orig time (s)	speedup	Tofino compiler # TCAM	ParserHawk (IPU) # Stages	Search Space (bits)	OPT time (s)	Orig time (s)	speedup	IPU compiler # Stages
Parse Ethernet	3	86	5.13	12.71	2.48	6	3	122	1.74	17.81	10.24	3
+ R1	3	86	2.8	15.67	5.6	9	3	122	1.74	17.96	10.32	4
- R3	3	86	2.76	13.21	4.69	5	3	122	1.89	20.17	10.67	3
+ R2	3	86	5.18	14.78	2.85	6	3	122	1.75	17.59	10.05	Conflict transition
Parse icmp	4	114	9.51	399.94	42.05	5	3	266	22.35	1980.84	88.63	3
+ R5	4	114	8.04	430.96	53.6	5	3	266	21.52	343.99	15.98	4
- R3	4	114	6.39	410.58	64.25	4	3	266	27.07	147.22	5.44	3
Parse MPLS	8	191	68.56	>86400	>1260.21	10	5	247	65.24	>86400	>1,324.34	Parser loop rej
+ unroll loop	8	191	69.34	>86400	>1,246.03	8	5	247	63.29	>86400	>1,365.14	6
- R1	8	191	67.63	>86400	>1,277.54	8	5	247	63.1	>86400	>1,369.26	Parser loop rej
+ R1	8	191	41.76	>86400	>2,068.97	22	5	247	92.29	>86400	>936.18	Parser loop rej
Large tran key	3	150	1.99	4,747.92	2,385.89	Wide tran key	4	220	0.67	3843.08	5,735.94	4
+ R4	3	150	2.05	8,849.20	4,316.68	3	4	220	0.73	3783.63	5,183.05	4
+ R1 + R4	3	150	2.18	24,611.27	11,289.57	6	4	220	1.82	4061.8	2,231.76	4
+ R3 + R4	3	150	2.08	8,833.46	4,246.86	3	4	220	0.72	3816.73	5,301.01	4
Multi-key (same pkt field)	3	70	0.53	131.17	247.49	6	3	98	1.61	51.06	31.71	4
- R5	3	70	0.6	51.58	85.97	4	3	98	2.03	51.51	25.37	3
- R5 - R3	3	70	0.74	115.20	155.68	3	3	98	1.73	353.72	204.46	3
Multi-keys (diff pkt fields)	3	97	7.98	5596.29	701.29	3	3	169	1.5	3001.19	2,000.79	4
+ R5	3	97	8	5,517.97	689.75	3	3	169	1.51	3060.11	2,026.56	3
- R5	3	97	8.26	5562.58	673.44	5	3	169	1.5	3003.58	2,002.39	4
Pure Extraction states	1	108	7.93	>86400	>10,895.33	1	2	465	244.61	>86400	>353.22	5
+ state merging	1	108	7.84	>86400	>11,020.41	1	2	465	245.7	>86400	>351.65	2
Sai V1 [15]	6	232	5.08	>86400	>17,007.87	6	3	472	13.69	>86400	>6,311.18	3
+ R2	6	232	153.64	>86400	>562.35	Too many TCAM	3	472	8.77	>86400	>9,851.77	Too many stages
Sai V2 [15]	21	797	2292.21	>86400	>37.69	21	5	1697	7837.68	>86400	>11.02	5
+ R1 + R2	21	797	9353.15	>86400	>9.24	Too many TCAM	5	1697	59073.57	>86400	>1.46	Too many stages
Dash V2 [4]	19	28	0.37	8.15	22.03	19	2	42	0.56	>86400	154,285.71	2
+ R1 + R2	19	28	0.41	7.87	19.20	Too many TCAM	2	42	0.57	>86400	151,578.95	Too many stages

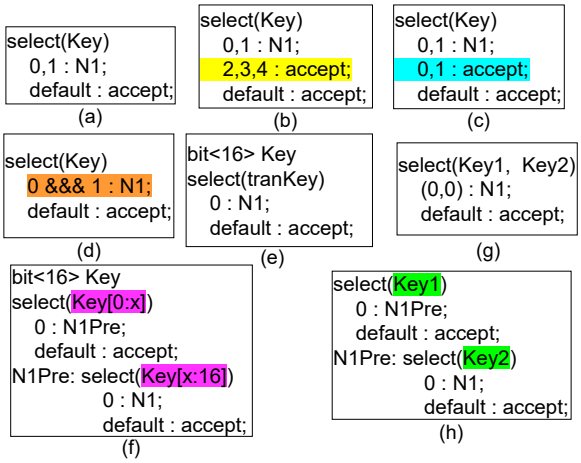


Figure 21: Collection of rewrite rules. R1:(a)→(b); R2:(a)→(c); R3:(a)→(d); R4:(e)→(f); R5:(g)→(h).

(a subproblem within our parallel synthesis procedures) is confined to a single core so any similar Linux x86_64 machine would suffice to run experiments. ParserHawk enables all optimizations (§6) by default.

7.1 Correctness Validation

We encode hardware constraints in ParserHawk based on each device's official documentation, ensuring that our encoding faithfully reflects the documented specifications. We verify that the output generated by ParserHawk is accepted by commercial compilers.

To test correctness, we perform a check using a simulator (Figure 22) by feeding randomly selected bitstreams to the hardware parsers and comparing their parsing results against the expected behavior defined by the specification. Concretely, we generate an Input Space Set by randomly sampling from all possible input bitstreams. Then, we simulate the behavior of both Spec and Impl to get corresponding output dictionaries (OD_{spec} and OD_{impl}). If there exists one input bitstream from the Input Space Set that makes these 2 dictionaries different, the verification fails. Otherwise, the generated Impl passes the verification. Besides, we test the generated parser on the open-source bmv2 simulator [2] by using Scapy [16] to generate test packets and check whether all packet fields are parsed correctly. We use Scapy to generate a TCP packet with a specified destination IP to test the Ethernet-IP parser. If the parsing logic is correct, the packet will be successfully delivered to the target; otherwise, it should be dropped. Currently, all compilation results for the created benchmarks pass the simulator's check.

For resource usage, we report the actual number of TCAM entries for Tofino and parser stages for IPU in Table 3. These numbers align with ParserHawk's resource estimations, indicating that it can accurately predict resource usage without over- or under-reporting.

7.2 Performance: compilation result and hardware resource usage

Table 3 and Table 4 compare all compilers' performance. We impose a 24-hour timeout on the compilation process, as overnight delays are typically impractical for developers. Red cells mean worse compilation outcomes Commercial compilers usually falsely reject input parser programs (11 out of 58) or generate outcome consuming suboptimal hardware resources (19 out of 58), which may finally lead to a compilation failure. By contrast, ParserHawk successfully compiles all benchmarks by exploring a larger space of implementation possibilities.

To be specific, commercial compilers *CANNOT* (1) do ***R4-like*** rewrite, (2) unroll loops within parser states (specific to IPU compiler), and (3) rule out never-reached entries in Figure 21(c). These limitations lead to compilation failures. Manually rewriting the parser into a semantically equivalent format that passes the compilation is not only error-prone but also risks overlooking valid solutions. In contrast, ParserHawk consistently produces semantically equivalent parser programs that use only the necessary hardware resources, while both commercial compilers and DPParserGen may generate outputs that consume more resources than required.

TCAM entry and stages. Compared with baseline compilers, ParserHawk's compilation outcome uses fewer than or equal hardware resources for the same benchmark. Besides, ParserHawk use the same amount of hardware resources while other compilers may use different amount of hardware resource for semantically equivalent parser programs. One reason behind this is that the baseline compiler cannot detect redundant state transition entries, allocating resources (e.g., TCAM entry, parser stage) for those entries as a result. For example, the Tofino compiler uses more TCAM entries for *Parse icmp* and the IPU compiler has to use 2 parser stages to fit all entries of a particular state in *Parse Ethernet + R1*.

Table 4 compares ParserHawk against DPParserGen over different hardware configurations. DPParserGen enables entry merging using mask+value combinations and transition key splitting, but its merging algorithm is suboptimal. Besides, similar to the motivating example in Figure 3.2, DPParserGen's parser generation rules sometimes fail to find a good result that use few TCAM entries. Such issues do not cause problems for ParserHawk because it only cares about the semantics instead of the written style of the input parser program.

Compilation speed. All baseline compilers complete the compilation within 1 minute but it takes much longer for ParserHawk to finish because ParserHawk relies on solver to explore the search space. The search difficulty increases as the input parser programs become complex. In Table 3, we report the search space size (in bits) to quantify the difficulty of compilation problem. However, by leveraging our optimizations, >90% of benchmarks complete compilation within 5 minutes and 44 out of 58 complete within 1 minute. For benchmarks with long compilation time (e.g. Sai V2), the root reason is that these parser specifications are complex (e.g., # parser states and transition rules) and therefore the search space is quite large (797 bits for Tofino, 1697 bits for IPU).

Based on our discussion with industrial developers, it usually takes engineers 1 hour to rewrite the parser program into a format that passes the compilation. Considering that ParserHawk can run 24/7, we believe its relatively long compilation time is acceptable. Given the increasing hourly manual cost [5] and decreasing per-unit computation cost [18], ParserHawk is and will continue to be more cost-efficient.

Summary: ParserHawk compiles parser programs that other compilers reject and finishes most benchmarks in under 5 minutes. Its output uses less than or the same hardware resources as those of existing compilers for all compilable cases.

Table 4: ParserHawk vs. DPParserGen over motivating examples (ME) in Figure 4. ME-1 needs entry emerging strategy; ME-2 needs to split the state transition key; ME-3 contains redundant entries.

	# TCAM		Parametrized Hardware Resources		
	ParserHawk	DPParserGen	State tran key width	Lookahead window	Extraction limit
Large tran key	3	6	Tofino	Tofino	Tofino
ME-1	5	6	4-bit	2-bit	10-bit
ME-2	4	4	16-bit	2-bit	10-bit
ME-2	6	9	8-bit	2-bit	10-bit
ME-3	1	10	16-bit	2-bit	10-bit

Table 5: Speed up effect from various optimizations.

	Tofino			IPU		
Program Name	Other OPT (s)	+ OPT5 (s)	+ OPT4, 5 (s)	Other OPT (s)	+ OPT5 (s)	+ OPT4, 5 (s)
Sai V1	74.13	49.98	5.08	1674.9	39.68	3.92
Dash V1	587.65	11.74	4.45	320.07	6.6	2.43
Large tran key	53.27	13.99	1.99	50.34	2.38	0.67

7.3 Retargetablity: compile cross-device

Each commercial compiler is developed to serve one type of target device. DPParserGen can only do parser generation for architectures with one big TCAM table. Extending DPParserGen to do compilation for pipelined-TCAM-table parser is nontrivial because its clustering strategies might increase the number of entries within a parser state, leading to split this merged state to multiple TCAM-table stages and increasing the final stage usage. By contrast, ParserHawk shows its retargetability in Table 3 and Table 4 because it can compile for both Tofino's and IPU's parser. Concretely, when switching from the Tofino to the IPU backend, ParserHawk only needs to add constraints to prevent TCAM entries from being reused across stages and to disallow revisiting the same entry multiple times. The core synthesis logic—variable generation and symbolic definitions—remains unchanged, resulting in <100 lines of code difference between Tofino- and IPU-targeted versions of ParserHawk, making ParserHawk a lightweight and easily portable compiler across hardware.

Summary: ParserHawk is retargetable by changing hardware constraints in its synthesis procedure while other baseline compilers are specific to a particular type of device.

7.4 Benefits of Optimization

We measure the speed-up effect of our optimization algorithms in Table 3 and Table 5. Across all benchmarks, enabling those algorithms reduces the compilation time for all 58 cases. 22 out of 58 cases fail to get compilation outcome within 24 hours without using optimization techniques. Algorithms described in §6 provide at least 309.44× (geometric mean) speed up on average (ranging from 1.46× to 154,285.71×), some of which achieve a leap from O(hour) to O(second).

We further analyze the performance impact in Table 5, focusing on the speedup that selected algorithms provide. In particular, we evaluate the effectiveness of *Opt4* and *Opt5* on 3 benchmarks by selectively enabling or disabling them, while keeping all other optimizations enabled by default. The result shows that both *Opt4* and *Opt5* can provide ≈ 10× speedup respectively because they can reduce the search space of constants' value selection and state transition key selection. Besides, *Opt1* is useful when a small portion of fields' bit are used to build the state transition key. *Opt2* is useful when there are many *irrelevant packet fields* within a parser

program. *Opt3* is applicable for parser architecture with symmetric features (e.g., the Tofino switch). Using >1 cores for parallel synthesis is generally useful for speedup across all benchmarks.

Summary: Proposed algorithms offer a significant speedup (hour → second) to compilation, and each optimization technique is effective for specific parser features. Usually, a given parser can benefit from more than 1 optimization technique.

8 Future work

We briefly discuss avenues for future improvement. First, *optimizing across parser and the packet-processing pipeline.* Figure 23 shows a hint in the P4 tutorial to write better parser programs, which involves redefining the packet fields. Neither ParserHawk nor other existing compilers can do so. We aim to extend ParserHawk to support co-optimization across packet definition, parser, and pipeline. Second, *further compilation speed-ups.* Table 3 shows an exponential increase of compilation time when the parser spec becomes more complex. For large and complex parser graphs, we could apply graph theory techniques to divide the parser dependency graph into smaller subgraphs, and then apply divide-and-conquer strategies to the program synthesis problem. Third, *more parser features.* We want to add more features in ParserHawk to support compiling parser programs whose functionality is not determined at compile time (e.g., P4 valueset and NPL parse break/continue).

9 Related work

Semantic verifier for programmable parsers. Leapfrog [27] verifies the semantic equivalence of different parsers. ParserHawk solves a harder problem, generating correct parser implementations using synthesis-based techniques.

Synthesis-based compiler for packet processing pipelines. CaT [32] and Chipmunk [30, 31] leverage solver-aided techniques to do compilation for multiple network devices at the level of packet-processing pipelines while ParserHawk generates code for the parser portion of the network device.

Retargetable compiler design across programming languages and devices. Polyglotter [29] and Metalift [24] build transpilers across domain-specific languages and devices without considering various hardware resource constraints. Alkali [36] and Hydride [34] are cross-device compilers for other modern hardware architectures. ParserHawk's contribution in parser compilation is complementary to these work.

10 Conclusion

We build, ParserHawk, a cross-device and program-synthesis-based compiler for line-rate parsers. Several optimization algorithms are incorporated to expedite compilation. ParserHawk can quickly generate compilation outcome that uses fewer resources in target devices compared with state-of-the-art approaches. We believe such a compiler can be applicable to do code generation for more emerging parser architectures in the future.

Acknowledgements

We are grateful to the anonymous SIGCOMM reviewers and our shepherd, Daehyeok Kim, for their insightful feedback. We thank Minlan Yu, Hari Thantry, Shijith Chempeth, Chen Tian, and Tianyao Chen for their valuable comments on previous drafts of this paper. We thank Glen Gibb for generously spending time explaining the DPParserGen's implementation to us. We thank Zhanghan Wang for his help with the bmv2 simulator, and Nikolaj Bjorner for his instant answers about Z3's internal mechanisms. This work was supported in part by grants from the UW FOCI, UW postdoc research award, Alibaba Cloud Research Intern Program, Google Student Researcher Intern Program, a gift from the eBPF foundation and Xilinx, and the National Science Foundation: NSF CAREER award (2340748), NSF-2422076, NSF-2019302.

References

[1] AMD Pensando Networking. https://www.amd.com/en/products/accelerators/pensando.html.
[2] BMV2 simulator tutorial. https://github.com/p4lang/tutorials.
[3] CME Group and Google Cloud Announce New Chicago Area Private Cloud Region and Co-location Facility for CME Group's Markets. https://www.cmegroup.com/media-room/press-releases/2024/6/26/cme_group_and_googlecloudannouncenewchicagoareaprivatecloudregio.html.
[4] Dash parser. https://github.com/sonic-net/DASH/blob/57e599c9d8e1f538ea01fad28781d8a0c2706ba5/dash-pipeline/bmv2/dash_parser.p4.
[5] Employment cost index. https://www.bls.gov/news.release/pdf/eci.pdf.
[6] Eval benchmarks. https://github.com/ParserHawk/ParserHawk/tree/main/z3/cegis_loop/one_short_revision/P4_examples.
[7] geneve: Generic Network Virtualization Encapsulation. https://datatracker.ietf.org/doc/html/rfc8926.
[8] Intel® Infrastructure Processing Unit (Intel® IPU). https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html.
[9] No parallel mode for z3 optimizer. https://github.com/Z3Prover/z3/issues/6642.
[10] NVIDIA BlueField-3 Networking Platform. https://resources.nvidia.com/en-us-accelerated-networking-resource-library/datasheet-nvidia-bluefield.
[11] P4 Compiler. https://github.com/p4lang/p4c.
[12] Parallel for finite domain (bit-vector) benchmarks. https://github.com/Z3Prover/z3/issues/1898.
[13] Parser Generator for Design Principles for Packet Parsers by Glen Gibb et al. https://github.com/grg/parser-gen.
[14] Product Brief Tofino Page | Barefoot. https://barefootnetworks.com/products/brief-tofino/.
[15] Sai parser. https://github.com/sonic-net/sonic-pins/blob/c513dec77a93ca21b8a14e8ddb2d2d725df57815/sai_p4/fixed/parser.p4.
[16] Scapy package. https://scapy.net/.
[17] Switch.p4 program. https://github.com/jafingerhut/p4lang-tests/blob/master/v1.0.3/switch-2017-03-07/out1/switch-translated-to-p4-16.p4.
[18] The cost of computing and the productivity puzzle. https://www.bennettinstitute.cam.ac.uk/blog/cost-of-computing.
[19] Tofino compiler. https://github.com/p4lang/p4c/tree/main/backends/tofino.
[20] Trident 3 Generation of 10/25/100G Ethernet Switches. https://www.broadcom.com/company/news/product-releases/12056.
[21] Trident 4 / BCM56690 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56690.
[22] Trident 5 / BCM78800 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78800.
[23] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. Host congestion control. In *ACM SIGCOMM*, 2023.
[24] Sahil Bhatia, Sumer Kohli, Sanjit A Seshia, and Alvin Cheung. Building code transpilers for domain-specific languages using program synthesis (experience paper). In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2023.
[25] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for sdn. In *ACM SIGCOMM*, 2013.
[26] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.
[27] Ryan Doenges, Tobias Kappé, John Sarracino, Nate Foster, and Greg Morrisett. Leapfrog: certified equivalence for protocol parsers. In *ACM PLDI*, page 950–965, New York, NY, USA, 2022. Association for Computing Machinery.
[28] Xinle Du, Jie Li, Yiyang Shao, Wei Wang, Shuihai Hu, Jingbin Zhou, and Kun Tan. Revisiting congestion control for wifi networks. In *ACM APNet*, 2024.
[29] Xiangyu Gao, Jiaqi Gao, Karan Kumar G, Ennan Zhai, Srinivas Narayana, and Anirudh Sivaraman. Cross-platform transpilation of packet-processing programs

using program synthesis. In *ACM APNet*, 2024.
[30] Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. Autogenerating Fast Packet-Processing Code Using Program Synthesis. In *ACM HotNets*, 2019.
[31] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In *ACM SIGCOMM*, page 44–61, New York, NY, USA, 2020.
[32] Xiangyu Gao, Divya Raghunathan, Ruijie Fang, Tao Wang, Xiaotong Zhu, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Cat: A solver-aided compiler for packet-processing pipelines. In *ACM ASPLOS*, page 72–88, New York, NY, USA, 2023.
[33] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. Design principles for packet parsers. In *Architectures for Networking and Communications Systems*, pages 13–24. IEEE, 2013.
[34] Akash Kothari, Abdul Rafae Noor, Muchen Xu, Hassam Uddin, Dhruv Baronia, Stefanos Baziotis, Vikram Adve, Charith Mendis, and Sudipta Sengupta. Hydride: A retargetable and extensible synthesis-based compiler for modern hardware architectures. In *ACM ASPLOS*, 2024.
[35] Christos Kozanitis, John Huber, Sushil Singh, and George Varghese. Leaping multiple headers in a single bound: Wire-speed parsing using the kangaroo system. In *2010 Proceedings IEEE INFOCOM*, pages 1–9, 2010.
[36] Jiaxin Lin, Zhiyuan Guo, Mihir Shah, Tao Ji, Yiying Zhang, Daehyeok Kim, and Aditya Akella. Portable and high-performance smartnic programs with alkali. In *USENIX NSDI*, 2025.
[37] Sankalp Mittal, Harikrishnan V., Patel Heetkumar, and Praveen Tammana. iguard: Efficient isolation forest design for malicious traffic detection in programmable switches. In *ACM CoNEXT*, 2024.
[38] Andy Myers, Brian Nigito, and Nate Foster. Network design considerations for trading systems. In *ACM HotNets*, 2024.
[39] Fabian Ruffy, Zhanghan Wang, Gianni Antichi, Aurojit Panda, and Anirudh Sivaraman. Incremental specialization of network programs. In *ACM HotNets*, 2024.
[40] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial Sketching for Finite Programs. In *ASPLOS*, 2006.

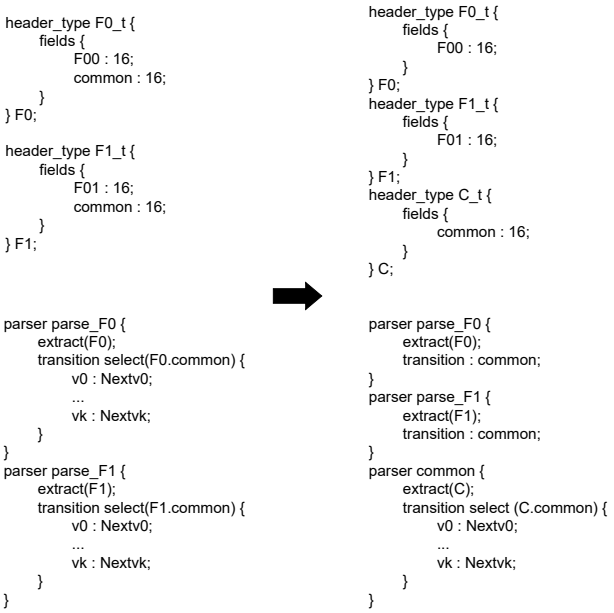


Figure 23: Left: original program; Right: new parser program by separating the common fields from individual fields.

APPENDIX

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

11 Simulator to check correctness of ParserHawk's output

Figure 22 provides a pseudocode of the simulator we built in Python to check the behaviors of both Spec and Impl.

```
Spec(I) \\simulate spec; Impl(I) \\simulate generated parser
for I ∈ Input Space:
    OD_spec = Spec(I); OD_impl = Impl(I);
    if OD_spec ≠ OD_impl:
        \\ Verification Failure
\\ Verification Pass
```

Figure 22: Simulator design to check correctness.

12 Express logical formula of device constraints in Z3

Encoding logical formulas into first-order logic forms in languages such as Z3 is not an easy job. For instance, different from other programming languages, the output for true and false conditions in Z3 should have the *SAME* type (e.g., both outputs are X-bit bitvector). Therefore, to limit the size of the generated state transition key, we go through all members in *Alloc* and *Lookahead* and decide to either prepend a 1-bit variable whose value is 0 or append the corresponding bit within a packet field to *key_sel*. The final state transition key is achieved by truncating the last *keyLimit* bits from *key_sel*.

The concrete code snippet is shown below:

```
// Build key_sel
dummy = BitVec('dummy', 1) Solver().add(dummy == 0)
for all i, j {
key_sel = If(Alloc[i][j] != k, Concat(dummy, key_sel),
Concat(key_sel, Extract(j, j, field[i])))} ...
// Generate Trankey by extracting the last keyLimit bits
Trankey = Extract(keyLimit - 1, 0, key_sel)
```

13 Example for future improvement

Dividing packet fields into 2 categories: common parts shared by multiple fields and individual parts owned only by specific fields. Then, we could redesign the parser to save the TCAM usage in Figure 23.