Acta Cybernetica **24** (2019) 61–81.

Keeping P4 Switches Fast and Fault-free through Automatic Verification*

Dániel Lukács[a], Gergely Pongrácz[b], and Máté Tejfel[a]

Abstract

The networking dataplane is going through a paradigm shift as softwarization of switches sees an increased pull from the market. Yet, software tooling to support development with these new technologies is still in its infancy. In this work, we introduce a framework for verifying performance requirement conformance of data plane protocols defined in the P4 language . We present a framework that transforms a P4 program in a versatile symbolic formula which can be utilized to answer various performance queries. We represented the system using denotational semantics and it can be easily extended with low-level target-dependent information. We demonstrate the operation of this system on a toy specification.

Keywords: P4, network verification, data plane, performance modeling, cost analysis

1 Introduction

Currently in the networking industry, network devices are being commoditized fast and software gets more and more market share as consumers want scalable and easily replaceable devices, while vendors want to keep development costs low. For example, software-defined networking (SDN) and Network function virtualization (NFV) technologies address this need by allowing network administrators to dynamically control network topology, configurations, and protocols.

New languages are emerging, aiming to assist network engineers to define the functioning of switches or network functions in the forwarding plane. Among them, P4 [7, 15] intends to keep the best aspects of both hardware and software by enabling network engineers to communicate their intent in a general high-level language, while the task of compiling high-level protocol description to low-level target

*The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013).

[a]Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages and Compilers, E-mail: {dlukacs,matej}@caesar.elte.hu

[b]Ericsson Hungary, E-mail: Gergely.Pongracz@ericsson.com

DOI: 10.14232/actacyb.24.1.2019.6

architectures is delegated to the backend software. The hybrid approach is highly effective, but burdens backends, static analyzers, and verification frameworks, as now these also have to take into account low-level targets. We recommend [13] and [7] on the transpiring software revolution in networking, the growing empowerment of network operators at the expense of switch vendors, and the question whether P4 will free the network from fixed-function switching interfaces (e. g. OpenFlow) by making protocol-inpedendent packet processing possible.

In this work, we present the first iteration of a framework for verifying *functional requirements* and *non-functional requirements* of network protocols in P4. Our focus in this paper is checking whether P4 programs satisfy performance requirements using cost analysis methodologies. In Section 1.1, we show that in networked environments, conformance to performance requirements determines if the switch can serve its purpose or instead it will get overflowed with packets and introduce network-wide anomalies.

1.1 Motivation

The goal of cost analysis is to approximate the execution cost of a program using the program code, without executing the program itself. A cost analysis system that is capable of giving exact execution costs for any problem is also capable of solving the halting problem. This implies that we have to stay content with approximations.

While it is always nice to have an idea about the costs of executing the programs we develop, this issue is more pressing with packet processing programs (such as P4) running on networked software switches. In any network – with or without P4 –, the number of packets processed in a unit of time (or energy used) correlates strongly with the unit of service prodvided (or rate of profit produced) by the network. Specifically for P4, one of the big promises of the language is that switches executing P4 protocols can combine the generality of NVFs software switches with the speed of earlier SDN switches (such as OpenFlow) that were closer to hardware, but only supported a fixed amount of protocols.

Moreover, beyond "more is better" being a desirable non-functional requirement, an easily overlooked fact is that the performance of a switch program is actually an important factor in the functional correctness and usability of that program, similarly to real time systems. In short, unless the switch is performant enough to serve all incoming requests in time, the packets will start accumulating in the packet buffers (installed for load balancing the temporary increases in demand), and upon buffer overflow, packets start getting lost, producing unexpected network behaviors.

To demonstrate a realistic requirement, let us examine switches in a 10 Gigabit Ethernet network. In such a network, the maximum incoming throughput is $10 Gbps = 1.25 GBps = 1250 MBps$. Assuming somewhat pessimistically that only minimal Ethernet frames with no payload are transferred, the size of the packets will be the size of the Ethernet frames, that is $8B + 64B + 12B = 84B$. From this, the incoming packet rate measured in *Mpps* (Million Packets Per Second) is

$1250 MBps/84 \approx 14.88 Mpps$. This means that a latency of $1/14.88s \approx 0.067s$ is allowed for 1 million packets, which is a latency of $10^9 * 1/(14.88 * 10^6)ns \approx \mathbf{67ns}$ for 1 packet. Assuming a modern CPU with $4.3 GHz = 4.3\ cycles/ns$ clock speed (such as the one alluded by Figure 13 in Section 4), we can conclude that at most $4.3\ cycles/ns \cdot 67ns \approx$ **288 cycles** can be spent for processing a packet to stay safe from buffer overflows.

One observation regarding this calculation can be that performance requirements towards switches turn out be quite strict. Another observation is that verification of such small boundaries will inherently require factoring in machine-level operations, such as the execution costs of various CPU instructions and accessing caches and main memory.

Yet another promising feature of P4 over earlier NFV and SDN approaches is that it is a well-designed high-level programming language with standardised syntax and semantics, enabling formal, well-generalisable analysis of switch and network behavior.

In this work, we present a formal system capable of taking into account the aforementioned low-level operations and statically deriving strict cost estimates from a represenation of P4 program semantics.

1.2 About P4

P4 programs describe the control flow of packet processing network devices, commonly called switches. One peculiarity of P4 is that certain control structures are intentionally left unspecified by the designers. Implementation questions are left to the compilers targeting different platforms: this way compiler designers can choose the most efficient solutions for their target platform. Moreover, the lack of superflous restrictions will enable more platforms to adopt the language. The price of this feature is that P4 programs cannot be generally analyzed without sufficient, low-level knowledge about the selected platforms. One of our goals in this work was to design a system that analyzes and verifies P4 independently of any platforms as deep as possible, and can be easily extended with platform specific information to achieve completeness.

Figure 1 depicts a P4 program. Here, the call to `V1Switch` lists the arguments (similar to function pointers) of a P4 pipeline. The implementation of `V1Switch` is unspecified, but from the interface description we can find out that incoming packets will be first processed by a parser, called `ParserImpl`. After the parsing phase, `V1Switch` will continue with following phases, each operating on the data structure containing the parsed packet (`headers`).

The parser is defined in P4, using state transitions. The parser control flow is illustrated by the state machine diagram in Figure 2 (side effects were abstracted away). Starting from state `start`, state `parse_eth` is immediately reached, and then, if the packet header signifies that the packet is an IPv4 packet (field `ethType` equals to 2048 in decimal), the next state is `parse_ipv4`. In both cases, the state machine goes in the accepting state, `accept`. Whether a packet is an IPv4 packet or not only makes a difference in side effects.

```
// ”basic_routing−bmv2.p4”
V1Switch(ParserImpl(),
          ingress (),
          verifyChecksum(),
          egress (),
          computeChecksum(),
          DeparserImpl()) main;

parser ParserImpl(packet_in packet,
                  out headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t stmeta){

  state start {
    transition parse_eth;
  }

  state parse_eth {
    packet.extract(hdr = hdr.eth);

    transition select(hdr.eth.ethType) {
      16w0x800: parse_ipv4;
      default: accept;
    }
  }

  state parse_ipv4 {
    packet.extract(hdr = hdr.ipv4);
    transition accept;
  }
}
 ....
```

```
 ....

header ethernet_t {
  bit<48> dstAddr;
  bit<48> srcAddr;
  bit<16> ethType;
}

header ipv4_t { ... }

struct headers {
  ethernet_t eth;
  ipv4_t      ipv4;
}

 ....

// ”core.p4”
extern packet_in {

  void extract<T>(
    out T hdr);

  void advance(
    in bit<32> size);

  bit<32> length();
}

// ”v1model.p4”
 ...

```

Figure 1: Excerpt of a P4 program.

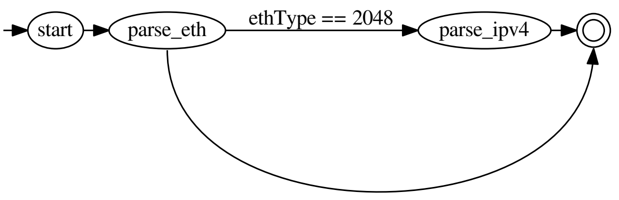

Figure 2: An automaton illustrating the parser in Figure 1.

In certain states the parser reads parts of the packet into the P4 program memory space. The memory space storing the packet, and the related method reading from this storage is declared by the data structure called `packet_in`. *Extern* data structures – such as `packet_in` – and methods are also unspecified. On the other hand, data structures such as `headers`, and `ethernet_t` are completely defined by the P4 code, which means we can work with these inside most P4 function constructs. For brevity, we included only the parts related to the parser that we analyze in this work. We believe the procedure intrduced in this paper can be effectively generalized for other control structures in the language – such as match-action tables – but we deem the validation of this claim as future research topic.

1.3 Contributions

Earlier, we enumerated our current research goals and some of the related analysis problems posed by P4. In this section, we intend to highlight specifically those problems we address in the current paper. We also showed earlier that in networked environments it is critical for switches to conform to specific time requirements, otherwise they cannot reliably provide the expected functionality. In this work, we outline a system that, given switch programs in a subset of P4 and adequate platform specifications, infers performance information that can be utilized to automatically verify whether given program satisifes given performance requirements on the given platform (see Section 2). This language subset was selected in hope that it covers a wide-enough range of challenges (such as target-dependence and low-level semantics) posed by P4, so that our system can be extended for the whole language.

For analyzing the cost of P4 programs, we adapted cost analysis approaches in existing literature [3, 16, 5] to P4. Our approach utilizes program transformations over formal representations of P4 program semantics (see Section 3). Advantages of the denotational approach is that it enables formal reasoning about its correctness, and its compositionality makes it easy to plug in target-specific information. Disadvantages are mostly related to efficiency: to increase precision we need to lower the level of abstraction we work on, and we can expect the amount of information on each abstraction level to grow exponentially (e. g. interfacing with the NIC, memory access, caching, cost of CPU instructions must all be carefully considered). To keep the rules simple, we utilize A-Normal form [9] that immensely simplifies function call evaluation semantics. Size and time efficiency of cost formula evaluation is assured by the introduction of let expressions (or alternatively nested lambda expressions) that can be used to eliminate redundant expressions and memoize intermediate results.

Various queries answering various performance questions can be created simply by parameterizing the symbolic formula resulting from the above process. As giving estimations with industrial-level precision for one or more platforms is out of the scope of this paper, we demonstrate the operation and application of the presented system using a toy specification of a fictional target in Section 4.

Our reference implementation is realized as a backend for the P4C compiler [1] and it heavily utilizes the Pure term rewriting system [6].

2 Cost analysis framework for P4

A challenge specific to P4 (although also occurring in other languages, such as C) is the handling of terms undefined by the specification, hereinafter referred to as *unspecified* terms to avoid confusion with theoretically undefined terms (such as division by zero, and the value of infinite recursion). It is the job of the compiler, to link unspecified structures – such as the extern object `packet_in` or the pipeline `V1Switch` in Figure 1 – to definitions that can be executed efficiently on the platform. Figure 3 depicts the data flow model we defined to address the challenge of

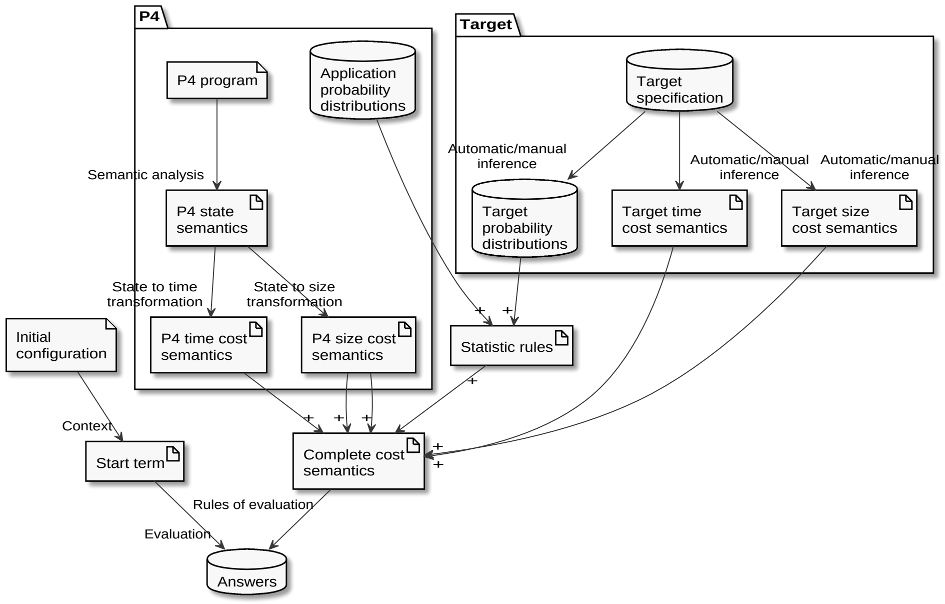
P4
P4 program
Application probability distributions
Semantic analysis
P4 state semantics
State to time transformation
State to size transformation
P4 time cost semantics
P4 size cost semantics
Target
Target specification
Automatic/manual inference
Automatic/manual inference
Automatic/manual inference
Target probability distributions
Target time cost semantics
Target size cost semantics
Statistic rules
Initial configuration
Context
Start term
Complete cost semantics
Rules of evaluation
Evaluation
Answers
+

Figure 3: The complete architecture of the cost analysis framework

analyzing and verifying target-dependent P4 code.

The P4 program is first parsed into an intermediate representation (IR) called *state semantics*, maybe utilizing IRs in existing P4 compiler solutions, such as P4C [1] or T4P4S [1]. This represents the program as mapping between explicit memory states before and after program execution. This representation is then transformed to two kinds of abstract semantics used specifically to analyze execution cost. One we will refer to as *time semantics*, as it maps meaningful size abstractions of input states to a numeric value characterizing program execution cost, such as CPU cycles. The other we will refer to as *size semantics*, as it maps meaningful size abstractions of input states to meaningful size abstractions of output states. We detail these representations and the transformation between them in Section 3.

As these transformations depend only on the P4 code, they are insufficient in themselves for completely deriving the cost of P4 programs, as this requires target-specific information. Our system expects this in the form of *target-specific time semantics and size semantics* rules. These rules either have to be delivered manually by developers employed by the target vendor, or it may be possible to automatically infer them, given a sufficiently formal specification describing the behavior of the target. We should note that such automatic inference from arbitrary target-dependent state semantics (or some other representation) requires further research efforts as it may introduce unexpected problems: on the target level we lose the comfortable guarantees provided by P4, such as upper-bounded loops or no loops at all, compile-time known memory sizes, structured control flow expressions and high-level data structures.

As it is common in static analysis, considering all possible program states (or even program inputs) would be unfeasible, and through abstraction we can acquire feasibility by trading away precision. In cost analysis, the problem manifests itself when we are dealing with conditional control flow. As predicates cannot be evaluated without the input, we either have to represent the costs of conditional execution paths as dependent on an unevaluated predicate, or – by further abstraction – we can treat the predicate as a random variable and apply statistical functions to these costs to produce performance information that is imprecise but useful in practice.

Figure 3 also includes these required statistical functions and informations in the architecture. As some of the predefined statistics (such as the average cost) require knowledge about the probability distribution of the predicates, this information is also required to perform the analysis. The two kinds of probability distributions in the figure relate to a distinction between predicates appearing in the semantics: some predicates are introduced in the target semantics (such as checking for cache hits), while others are dependent on program input and program context (such as deciding whether a given header can be parsed from a packet in a given parser state, or whether a packet matches any entry in the match-action table).

Given all these informations and a target-dependent abstract initial program memory state, we can finally evaluate the abstract call to the program entry point with any or all of the predefined statistics to acquire performance information about the P4 program and verify whether or in what circumstances does it conforms to the performance requirements.

In Section 4, we also go through the most important steps of this process with illustrations.

3 Transforming programs to program costs

In this section, we present the program transformation system used to derive time and size semantics from the state semantics of a P4 program. We realized the transformation as a term rewriting system containing reduction rules, analogous to function definitions in the lambda calculus. An advantage of functional style beyond formality is that it automatically guarantees confluence as per the Church-Rosser theorem.

For the ease of reproducibility, the examples in this paper were formalized in the executable term rewriting language, Pure [6]. Pure mostly follows the notational naming conventions of ML-style functional languages. We give basic description for less familiar syntax elements in Pure, but ultimately we have to refer the reader to the Pure language manual. To separate the meta-syntax (i. e. the syntax of Pure) from the concrete syntax of the semantics (defined by EBNF grammars in this paper), we typeset meta-syntactical symbols in bold, meta-syntactical variables using normal fonts, and typeset all concrete symbolic values in italic. In Figure 4

```
⟨App⟩          ::= ⟨Name⟩ ⟨State⟩ ⟨Scope⟩

⟨TimeCost⟩     ::= 'TIME' ⟨Expr⟩ ⟨SizeCost⟩ ⟨Scope⟩
                |  ⟨TimeCost⟩ '+' ⟨TimeCost⟩
                |  ⟨ProbDist⟩ '*' ⟨TimeCost⟩
                |  ⟨Number⟩ '*' ⟨TimeCost⟩
                |  ...

⟨SizeCost⟩     ::= 'SIZE' ⟨Expr⟩ ⟨SizeCost⟩ ⟨Scope⟩
                |  ...

⟨Reference⟩    ::= ⟨State⟩ ['!⃟' ⟨Scope⟩] {'!' ⟨Name⟩}

⟨RndReference⟩ ::= 'ℛ' ['!⃟' ⟨Scope⟩] {'!' ⟨Name⟩}

⟨ProbDist⟩     ::= '𝒫' ⟨Predicate⟩
```

Figure 4: EBNF syntax for the most frequent expressions used in this paper.

we find mixed rules from the EBNF noindent grammars describing the languages we are using for expressing state, time and size semantics of P4 programs.

Function applications in the state semantics apply the definition referred by the given name, to the argument which is a program state. Note that the grammar enforces A-normal form (ANF) [9]: applications are only allowed to have variable symbols and concrete states as arguments. The state semantics syntax exclusively uses lexical scoping (instead of the mixed lexical-dynamical approach of P4): a mapping of names are passed to called functions. The names in the scope are used in function definitions to resolve references pointing to the state. The exclamation mark (`!`) is Pure syntax for record field access while !⃟ was defined by us to handle sequences of field accesses, since `!` is left-associative in Pure. We represent concrete (i. e. transformable) *let expressions* and *case analyses* in Pure's built-in syntax (with `_when_` and `_case_` respectively).

Time and size costs of function definitions are denoted with the `TIME` and `SIZE` expressions: these work similarly to applications, but they evaluate to time and size costs instead of program states. We also include expressions required for probabilistic handling of case analyses, and a few target-defined constants appearing in Section 4.

Next, we present the rules §3.1–§3.4 forming the transformation system (a meta term rewriting system) between state semantics and time semantics. We expect that only only one system is loaded in the rewriting environment at a time, so the only subexpressions expanded are those appearing on the left sides. We will assume, that term rewriting rules – represented here with an arrow ($\overset{\cdot}{\underset{\cdot}{\Longrightarrow}}$) between the two sides – are also part of the concrete syntax: they can be created, transformed, and added to programs during runtime.

Rules §3.1 and §3.2 apply the time cost function to both sides of a state rule (similarly to how we usually do this to equations). The former one is an exception for program entry, mapping a concrete input state to its size concrete size abstraction.

Rule §3.3 formulates the cost of an application of a function f to argument x given a size cost n, which we expect (and guarantee by the other rules) to be

$$\frac{\textit{main}\ \mathrm{x}\ \mathrm{s}_1 \mathrel{\overset{\cdot}{\underset{\cdot}{\Longrightarrow}}} \mathrm{f}\ \mathrm{y}\ \mathrm{s}_2}{\textit{TIME main}\ \_\ \_ \mathrel{\overset{\cdot}{\underset{\cdot}{\Longrightarrow}}} \textit{TIME}\ \mathrm{f}\ (\textit{SIZE}\ \mathrm{y}\ \{\}\ \{\})\ \mathrm{s}_2} \qquad \S3.1$$

$$\frac{\mathrm{f}\ \mathrm{x}\ \mathrm{s}_1 \mathrel{\overset{\cdot}{\underset{\cdot}{\Longrightarrow}}} \mathrm{rhs}}{\textit{TIME}\ (\mathrm{f}\ \mathrm{x}\ \mathrm{s}_1)\ \mathrm{n}\ \mathrm{s} \mathrel{\overset{\cdot}{\underset{\cdot}{\Longrightarrow}}} \textit{TIME}\ \mathrm{rhs}\ \mathrm{n}\ \mathrm{s}}\ \ \mathrm{f} \neq \textit{main} \qquad \S3.2$$

$$\frac{\textit{TIME}\ (\mathrm{f}\ \mathrm{x}\ \mathrm{s}_1)\ \mathrm{n}\ \mathrm{s}_2}{\textit{TIME}\ \mathrm{f}\ \mathrm{n}\ \mathrm{s}_1} \qquad \S3.3$$

$$\frac{\textit{TIME}\ (\_\ \_\mathrm{when}\_\ \mathrm{args})\ \mathrm{n}\ \mathrm{s}}{\begin{array}{l}(\textbf{foldl1}\ (+)\ \mathrm{times})\ \_\mathrm{when}\_\ \mathrm{sizes} \\ \\ \mathrm{where} \\ \quad \mathrm{sizes}\ =\ ...; \\ \quad \mathrm{times} = \ ...;\end{array}} \qquad \S3.4$$

$$\frac{\textit{TIME}\ (\_\mathrm{case}\_\ (\_\ \Diamond\ \mathrm{fields})\ \mathrm{cs})\ \mathrm{n}\ \mathrm{s}_2}{\begin{array}{l}\textit{TIME cacheIn}\ \mathrm{n}\ \{\ \textit{ref} \Rightarrow \mathrm{fields}\ \} \\ +\ \_\mathrm{case}\_\ (\mathscr{R}\ \Diamond\ \mathrm{fields})\ \mathrm{tcs} \\ \\ \mathrm{where} \\ \quad \mathrm{tcs}\ =\ ...;\end{array}} \qquad \S3.5$$

Figure 5: Program transformation rules mapping state semantics to time semantics.

the size abstraction of x. Note that while the various call semantics would require inclusion of different costs in this rule, we solved this problem by enforcing ANF: as function compositions are disallowed we now know that function arguments are evaluated (analyzed) at a separate program point. Without ANF this rule would have to be more complex. The size argument is only used in loop analysis: while do not perform loops analysis in this paper as it is currently not relevant for P4, we prepared the notation in preparation for future research.

Rule §3.4 transforms let expressions in the state semantics for let expressions in the time semantics. Let expressions assure size and time efficiency of cost formula evaluation as they can be used to eliminate redundant expressions and memoize intermediate results. While nested lambda expressions can be used for the same purpose, let expressions proved to be a far more human-readable alternative. ANF is a must in both cases. To aid readability, we omitted implementation details of the body of this rule, and instead recommend the reader to look at the input-output chart in Figure 6. The rule will bind the size abstraction of the program state after each operation in a sequence to variables in the let expressions. These variables are then utilized in the summation of the time costs of these operations that is returned by the expression. Rule §3.4 requires that bodies of let expressions in the state semantics refer to a single bound variable of their parent let expressions (non-conforming let expressions can be easily translated to conforming ones). On Figure 6, we can observe the effect of the required changes: instead of featuring a sum of time costs in the body of the let expression, we only feature the size cost of the last application, which depends on the size cost of the expression before the last one, and so on.

Rule §3.5 rewrites a case analysis returning a state, to a case analysis returning a time expression. To aid readability, we omitted implementation details of the body of this rule, and instead recommend the reader to look at the input-output chart in Figure 6: Here, we first add the (target-dependent) potential cost (c) of caching the head value (i. e. reading from main memory to CPU cache), followed by conditionally adding the execution of executing the matching case body (b_i), plus the cost of the comparison between the head and the case pattern (c_i) (also taking into account the costs of all preceding matches). As the time semantics abstracted away program state information, we cannot precisely evaluate the case analysis anymore. Yet, we can still extract valuable information by applying statistical transformations, such as taking the maximum or the average of the case costs. We signified this by transforming the case head into a random variable ($\mathscr{R}$).

For brevity, we do not include the system transforming state semantics to size semantics, as it is mostly analogous to the system in Figure 5, but without the requirement to sum up the sizes of the sizes of intermediate operations.

```
TIME (x_3
      _when_
        [ x_1 --> f x_0 s_f
        , x_2 --> g x_1 s_g
        , x_3 --> h x_2 s_h
        ]) n s
```

(a)

```
(TIME f n_x s_f + TIME g n_y s_g + TIME h n_z s_h)
_when_
  [ n_x --> SIZE x n  s
  , n_y --> SIZE f n_x s_f
  , n_z --> SIZE g n_y s_g
  ]
```

(b)

```
SIZE (x_3
      _when_
       [ x_1 --> f x_0 s_f
       , x_2 --> g x_1 s_g
       , x_3 --> h x_2 s_h
       ]) n s
```

(c)

```
SIZE h n_z s_h
_when_
  [ n_x --> n
  , n_y --> SIZE f n_x s_f
  , n_z --> SIZE g n_y s_g
  ]
```

(d)

Figure 6: Example of time and size cost reductions of a let expression.

Finally in Figure 8, we present examples of two families of statistical rules. Such rules can be used to handle conditional control flows in partially reduced time cost expressions. For example, when applied to branching expressions §3.6 will return the cost of the most expensive branch (deriving the *worst case execution time*), while §3.7 will weigh the cost of each branch with the probability of the branch being executed times the probability that non of the preceding branches are being executed (deriving the *mean execution time*). All statistical rules behave as identity

$$\begin{array}{l} \textit{TIME}\ (\_\text{case}\_\ (\text{x} \diamond \text{fields}) \\ \qquad [\ \text{patt}_1 \dashrightarrow \text{b}_1 \\ \qquad ,\ \text{patt}_2 \dashrightarrow \text{b}_2 \\ \qquad ,\ \_ \qquad \dashrightarrow \text{b}_3 \\ \qquad])\ \text{n s} \end{array}$$

(a)

$$\begin{array}{l} \text{c} + \\ \_\text{case}\_\ (\mathscr{R} \diamond \text{fields}\,) \\ \quad [\ \text{patt}_1 \dashrightarrow \text{c}_1 + \qquad\quad \textit{TIME}\ \text{b}_1\ \text{n s} \\ \quad ,\ \text{patt}_2 \dashrightarrow \text{c}_1 + \text{c}_2 + \textit{TIME}\ \text{b}_2\ \text{n s} \\ \quad ,\ \_ \qquad \dashrightarrow \text{c}_1 + \text{c}_2 + \textit{TIME}\ \text{b}_3\ \text{n s} \\ \quad] \\ \\ \text{where} \\ \quad \text{c} \ \ = \textit{TIME cacheIn}\ \text{n}\ \{\ \textit{ref} \Rightarrow \text{s ! fields}\,\}; \\ \quad \text{c}_1 = \textit{TIME cmp}\quad\ \ \text{n}\ \{\ \textit{ref}\ \ \Rightarrow \text{fields} \\ \qquad\qquad\qquad\qquad\quad ,\ \textit{const} \Rightarrow\ \text{patt}_1\,\}; \\ \quad \text{c}_2 = \textit{TIME cmp}\quad\ \ \text{n}\ \{\ \textit{ref}\ \ \Rightarrow \text{fields} \\ \qquad\qquad\qquad\qquad\quad ,\ \textit{const} \Rightarrow\ \text{patt}_2\,\}; \end{array}$$

(b)

Figure 7: Time cost reduction of a case analysis expression.

for non-branching expressions, as these are corresponding to one-element samples. Further statistics such as *best case execution time* and *variance* can be realized as similar rules.

$$\frac{\textit{MAX}\ (\_\_\textbf{ifelse}\_\_\ \_\ \text{t}_1\ \text{t}_2)}{\textbf{max}\ \text{t}_1\ \text{t}_2}\ \S 3.6 \qquad\qquad \frac{\textit{AVG}\ (\_\_\textbf{ifelse}\_\_\ \text{c}\ \text{t}_1\ \text{t}_2)}{(\mathscr{P}\ \text{c}) * \text{t}_1\ +\ (1 - (\mathscr{P}\ \text{c})) * \text{t}_2}\ \S 3.7$$

Figure 8: Statistics rules for handling conditional control flow

4 Case study

In this section, we illustrate the system presented in the previous sections by going through the intermediate representations and target-dependent components used in the analysis of the parser in the small P4 program in Figure 1. To assist with this demonstration and keep this paper concise at the same time, we also provide a toy-sized target-dependent specification that can be substituted in the partial formula to obtain the final performance formula. In Figure 15, we illustrate a possible application of this performance formula to show how the estimated performance of this program on the specified platform changes w. r. t. the (application-dependent) probability of the transmitted packet being an IPv4 packet, and the (target-dependent) probability of cache misses.

4.1 State semantics transformation

In the first step of the analysis the P4 program is transformed to state semantics representation: a system of reduction rules describing the program as a composition of functions over the program state. We expect that production of such a representation is relatively straightforward after the P4 program was parsed into an IR, such as the one used by the P4C compiler [1]. Figure 9 depicts the reduction rules

```
start x s
─────────────────────────────
x2
_when_
 [ x1 --→
    extract
      x
      { hdr    ⇒ s!hdr!eth
      , packet ⇒ s!packet}
 , x2 --→
    _case_
     (x1 ◇ (s!hdr)!eth!ethType)
     [ 2048 --→ parse_ipv4 x1 s
     , _    --→ accept x1 s
     ]
 ]
```

(a)

```
parse_ipv4 x s
─────────────────────────────
x2
_when_
 [ x1 --→
    extract
      x
      { hdr    ⇒ s!hdr!ipv4
      , packet ⇒ s!packet}
 , x2 --→ accept x1 s
 ]
```

(b)

Figure 9: Formal semantics of `start` and `parse_ipv4` generated from P4 program

corresponding to the `start` and `parse_ipv4` state transitions. Here, the input program state x is modified by copying the bits at the current cursor position of `packet` to the memory segment specified by the `s!hdr!eth` field of the scope. Then, a value by the name in the `s!hdr!eth!ethType` variable is read from the resulting program state and is pattern matched to select the transition to the next parser state (formalized as a function application). The program state after executing the selected transition will be returned. We should note that the `extract` method in the `packet_in` extern is not defined in the P4 program (as it is target-dependent). If the state semantics is intended for execution, then an evaluation rule must be defined for this method. For cost analysis this is not required.

We may also note that transition `parse_eth` was eliminated in a compiler optimization step. As P4C was designed to separate target-independent P4 code optimizations from target-dependent ones into parts called *frontend* and *backend* respectively, we are free to rely on optimizations in the frontend, but are required to steer clear of those in the backends. This way, our cost estimations will always assume that any P4 compiler it is used with generates at least as efficient intermediate code as the P4C frontend. This assumption is automatically satisfied for P4 compilers realized as P4C backends, such as T4P4S [11].

In the next analysis step, the state semantics in Figure 9 is abstracted into time and size semantics using the program transformation system presented in Section 3. Figure 10 depicts the time semantics rule corresponding the `start` transition of the parser. We first calculate the n_1 size abstraction (a structure) of the program state after `extract` based on the size abstraction of the input program state n, and then the n_2 one after the case analysis (not depicted) based on n_1. Using these size abstractions, we can return the sum describing the execution costs of each operations and also the additional costs of the program control flow.

We defined the transformation rules so that the time cost semantics of a function call such as parameter passing (defined to be *copy-in/copy-out* by the P4$_{16}$ specification [15]) are included in the rule describing the function definition. This

$$\frac{\textit{TIME start}\ \mathrm{n\ s}}{\begin{array}{l}
\textit{TIME extract}\ \mathrm{n}_0\ \{\ \dots\ \} \\
\\
+\ \textit{TIME cacheIn}\ \mathrm{n}_1\ \{\ \dots\ \} \\
+\ \_\mathrm{case}\_\ (\mathscr{R}\ \diamond\ (\mathrm{s}!\textit{hdr})!\textit{eth}!\,\textit{ethType}) \\
\quad [\ 2048 \dashrightarrow\ \ \textit{TIME cmp}\ \mathrm{n}_1\ \{\ \dots\ \} + \textit{TIME parse\_ipv4}\ \mathrm{n}_2\ \mathrm{s} \\
\\
\quad ,\ \_ \qquad \dashrightarrow\ \ \textit{TIME cmp}\ \mathrm{n}_1\ \{\ \dots\ \} + \textit{TIME}\ \mathrm{accept}\ \mathrm{n}_2\ \mathrm{s} \\
\quad] \\
\_\mathrm{when}\_ \\
\ [\ \mathrm{n}_0 \dashrightarrow \mathrm{n} \\
\ ,\ \mathrm{n}_1 \dashrightarrow \textit{SIZE extract}\ \mathrm{n}_0\ \{\ \dots\ \} \\
\ ,\ \mathrm{n}_2 \dashrightarrow \_\mathrm{case}\_ \\
\qquad\qquad\qquad \dots \\
\ ]
\end{array}}$$

Figure 10: Time cost semantics of `start` derived using the system in Figure 5.

is the reason why the costs of the function do not appear in any of the applications (transitions do not require parameter passing). At this point, to represent the costs of the program control flow we also include the costs of reading an operand from memory into the cache, and the costs of performing the comparisons in the branches (the default case does not require a comparison, so in this case, only the preceding comparisons are counted).

4.2 Target dependent semantics

We utilize the semantic rules by applying them to concrete terms, i. e. function calls parameterized by a concrete program state. A model program state is depicted by Figure 11a, while the transformed size abstraction of this state is depicted by Figure 11b.

Conceptually, the concrete state is partially defined by the target, as it also describes the memory allocation scheme as prescribed by the target-specific compiler backend. For example, a backend implementing copy-in/copy-out semantics will allocate data-size memory for every function arguments, while another backend may choose to depart from the language specification and implement call-by-reference semantics with stacks to save both time and space. Moreover, parts of the concrete state may be explicitly target-dependent, such as the memory reserved for externs, and the formal representations of related storages (such as I/O buffers, L1, L2, L3 caches, NUMA memories). Figure 11a describes a program state in which the extern memory (called `packet_in`) reserved for storing a raw incoming packet is a 1 KB buffer and a pointer points to the position of the last parsed byte. As extern memory is ultimately target-dependent, we modeled this structure after the identically named C structure in Figure 12a.

It also includes the `header` structure declared in the original P4 program code with fields having the respective sizes. For simplicity, we omitted more intricate details, such as copy-in/copy-out semantics for this model. The state also includes

```
{ headers ⇒
    { eth  ⇒
          { ethType ⇒ mkarray 0 2
          , dstAddr ⇒ mkarray 0 6
          , srcAddr ⇒ mkarray 0 6
          }

    , ipv4 ⇒
          {
            ...
          }

    }
, packet_in ⇒
     { cursor ⇒ 0
     , buffer ⇒ mkarray 0 1000
     }
, cacheLineSize ⇒ 64
, cpuWordLength ⇒ 8
, cache         ⇒ mkarray 0 32000
, mem           ⇒ ...
, nic           ⇒ ...
, ...
}
```

(a) A simplified model of a concrete P4 program memory state. Sizes are given in bytes.

```
{ headers ⇒
    { eth  ⇒
          { ethType ⇒ 2
          , dstAddr ⇒ 6
          , srcAddr ⇒ 6
          , sizeof   ⇒ 14
          }
    , ipv4 ⇒
          { ...
            sizeof ⇒ 20
          }
    , sizeof ⇒ 34
    }
, packet_in ⇒
     { ...
       sizeof ⇒ 1000
     }
, cacheLineSize ⇒ 64
, cpuWordLength ⇒ 8
, cache         ⇒ 32000

, ...
, sizeof = ...
}
```

(b) Size abstraction of the P4 program state in Figure 11a.

Figure 11

explicitly target-dependent segments such as the 32 KB sized L1 `cache` field. Since we are working with small packets, we may assume infinite RAM memory without losing practical soundness.

Unless we want to execute the state semantics, we do not need the concrete program state. We described it to make it easier to understand its size abstraction. Figure 11b depicts this size abstraction. Abstracting the concrete state is usually non-trivial: P4 structures get a special field (denoted here as `sizeof`) storing its aggregated sizes, target-dependent constants are kept as is, and – to enable loop analysis in the packet parser – the size abstraction of `packet_in` is the size of the yet unprocessed part of the packet. As we do not yet perform loop analysis that would require intricate size abstractions, we left answering the questions of state abstraction for future research.

Any cost semantics derived from P4 code only cannot be a complete description of program behavior: as unspecified P4 constructs are defined by targets, we require target-specific information about how this target implements the unspecified P4 constructs (such as the pipeline and externs, as seen before). Figure 12b provides a partial example that formally specifies such target-specific information.

Note that we devised the target model in this section manually: while we suspect it to be reasonable, it is far too simplistic to be used for predicting real targets with common but advanced low-level features (such as NUMA, DMA, multiple cores, etc.).

```
typedef unsigned char byte;

typedef struct packet_in {
  byte buffer [1024];
  byte* cursor;
} packet_in;


void extract(packet_in* packet,
             void* hdr,
             unsigned long hdrLen) {

  memcpy(hdr, packet->cursor, hdrLen);

  packet->cursor = packet->cursor + hdrLen;

}
```

(a) An implementation of the P4 extern method `extract` in C.

```
        TIME extract n s
  ------------------------------
   TIME cacheIn
         n
         { ref ⇒ s!packet }
  + TIME memcpy                         §4.2.1
           n
           { src ⇒ s!packet
           , dst ⇒ s!hdr }
  + CPU_ADD


        TIME memcpy n s
  ------------------------------
  ceil (d/l)
   *
  (L1_TO_CPU
   + CPU_MOV                            §4.2.2
   + L1_FROM_CPU)

  where
    d = n◇(s!dst);
    l = n◇cpuWordLength;
```

(b) Cost model based on 12a.

Figure 12

Target-dependent time semantics may be delivered automatically from a sufficiently formal target specification, but – due to the lack of various invariants guaranteed by P4, such as compile-time known memory requirements and no loops or upper-bounded loops only – we deem this problem to be non-trivial and out of the scope of this paper.

In Rule §4.2.1 of Figure 12b, we model the unspecified `extract` operation that attempts to parse a packet header (i. e. copies bits of the incoming packet to a memory segment representing a header), as a call to a system-level copy operation such as C's `memcpy`, followed by incrementing the counter by the size of the parsed data (see Figure 12a). As a simplification, we assume that only valid (as in, parseable) packets arrive: if this were not the case (as usually), the rule would have to be extended with the cost of checking for input end and also the early return should be calculated in.

Our cost model of `memcpy` in Rule §4.2.2 assumes that the part of packet under operation is already cached (and fits entirely in the cache, which is reasonable for $32k$ cache size), and then read its 64 bit size chunks (size of the CPU registers) into the CPU registers with the aim of performing the CPU-level copy instruction. The size of part is the compile-time known destination size, i. e. the size the header we want to parse from the raw packet. Note that we may model the cost of comparison (used in calculating the cost of case analysis) very similarly, with the only difference that we need to read two words into the CPU registers instead of one, and perform the CPU-level comparison operation instead of copying.

Description	**LHS**	**RHS (cycles)**
Cost of comparing the contents of two CPU registers.	*CPU_CMP*	1
Cost of copying data between CPU registers.	*CPU_MOV*	2
Cost of addition with a constant number.	*CPU_ADD*	5
Cost of copying a word from L1 to a CPU register.	*L1_TO_CPU*	5
Cost of copying a word from a CPU register back to L1.	*L1_FROM_CPU*	5
Cost of copying a cache line from memory to L1 cache.	*MEM_TO_CACHE*	79 + 200

Figure 13: CPU architecture model[1] specifying CPU instruction execution costs.

In Figure 13, we defined all cost constants required for completely reducing the preceding formulae, based on external specifications and benchmarks of the respective operations of the selected CPU architecture.

As expected, the costs are seemingly dominated by the reads from memory to caches, but we should note that this operation handles cache lines (64 byte in in our example), while the others handle register-sized data (64 bit in our example) and thus will be repeated in succession for larger data (so if copying 8 byte from cache to CPU registers requires 5 cycles, the same operation for 64 byte will require 40 cycles in this model).

4.3 Symbolic time cost formula and applications

At this point, we introduced most of the basic components required for evaluating the time cost of `start`, given the abstracted state in Figure 11b. We merge the cost semantics generated from P4 with the target-specific cost semantics to derive an intermediate formula (not depicted here because of its size) from the cost expression of `start`, and finally apply our chosen statistics transformation.

Figure 14 depicts the worst case execution time of `start` we derived using Rule §3.6. This approximates the execution time of `start` when every incoming packet is an IPv4 packet (i. e. `ethType` is `2048`) and the cache misses in each attempt.

The best case execution time of `start` is the formula: 8 * (`L1_TO_CPU` + `CPU_MOV` + `L1_FROM_CPU`) + `CPU_ADD` + 1 * (2* `L1_TO_CPU` + `CPU_CMP` + `L1_FROM_CPU`). This is the execution time when no incoming packets are IPv4 packets, and the cache hits in each attempt.

We can derive the average case similarly, but instead of taking each member for granted, we appropriately have to weigh costs corresponding to parts of the code with with the probabilities of that part being executed. For example, the costs of

[1]Based on Intel Skylake X (4.3 GHz, 32KB L1) [4, 2]. *MEM_TO_CACHE* value was adapted assuming 0.25ns per cycle.

TIME extract n_0 { ... }	
• Cost of reading the packet into cache	*MEM_TO_CACHE*
• Cost of extracting the header	+ 8 ∗ (*L1_TO_CPU* + *CPU_MOV* + *L1_FROM_CPU*)
• Cost of incrementing the cursor	+ *CPU_ADD*
Case analysis (WCET)	
• Cost of reading the case head into cache	+ *MEM_TO_CACHE*
• Cost of comparing the case head with the pattern of the most expensive case (i. e. the first one).	+ 1 ∗ (2∗*L1_TO_CPU* + *CPU_CMP* + *L1_FROM_CPU*)
TIME parse_ipv4 n_2 s	+ *MEM_TO_CACHE* + 8 ∗ (*L1_TO_CPU* + *CPU_MOV* + *L1_FROM_CPU*) + *CPU_ADD*

Figure 14: The WCET of state `start`.

the `parse_ipv4` state transition is weighed with the probability of the `ethType` field of the header being the value `2048`. For conciseness, we omit the resulting expected value formula from this paper, and instead show only the final values in Figure 15, given various probability distributions.

Using the constants in Figure 13, we can finally evaluate the partially evaluated formulas to a single numeric value characterizing the performance. In Figure 15, we depicted three constants and the average cost computed over various probabilities. The constant values marked with `BCET` and `WCET` denote the best and worst case execution times of `start` as we discussed earlier. `TOL` was introduced in Section 1.1 as the maximum number of cycles that can be spent for processing a packet without causing buffer overflow in the long run by a switch residing in a 10 Gigabit Ethernet network and implemented on the CPU architecture in Figure 13. Note that `TOL` encompasses the entirety of the packet processing process starting with packet arrival on the NIC, while the rest of the numbers only characterize the costs of packet parsing starting from `start`. To derive a more meaningful chart, we would need to reduce the time cost formula for the program entry point instead of `start`, or set `TOL` lower.

Instead of deriving the average execution time with arbitrary probability distributions, we plotted the average (measured in cycles) over several different distributions. Because of the case analysis and the possible need of caching, the average depends on the probability of the packet being an IPv4 packet, and the probability of cache misses. This means that we are working with pairs of probability distributions, each defined over two values (the predicate in question being true and false). Two keep the plot in 2 dimensions, we only used two probabilities of packets having IPv4 types (0 meaning no IPv4 packets arrive at all, and 1 meaning only IPv4 packets arrive), each represented by two different lines. We keep track of the cache miss probabilities on the x-axis.

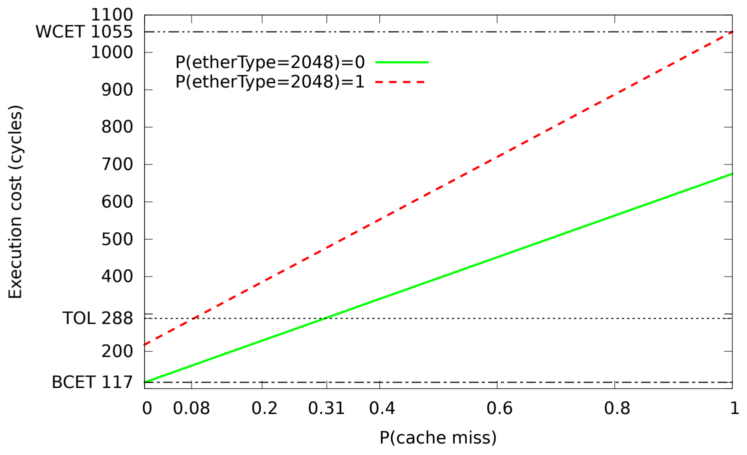

Figure 15: Latency characteristics of `start` over IPv4 and cache miss probabilities.

By looking at the plot, we can conclude that a cache miss ratio of 0.08 and below will guarantee that the examined P4 function call will not take longer than the allowable limit even if we have to process all packets as IPv4 packets. On the other hand, ratios above 0.31 are clearly dangerous: if cache the misses that frequently, buffer overflows cannot be avoided even if no IPv4 packets arrive at all. As these extremities rarely occur in practice, cache miss ratios between 0.08 and 0.31 can be candidates for further testing (or cost analyses with more precise estimations) to find the highest cache miss ratio with which the switch can still run sustainably (i. e. without causing buffer overflows) in a given application environment.

5 Related work

In this paper, we presented a system for verifying conformance of P4$_{16}$ programs to performance requirements. Our long term goal is to innovate a framework that is capable of verifying both functional and non-functional requirements using the same base representation. To handle cost analysis methodically, we utilized the seminal work of Wegbreit [16] and the idea of cost relation systems (CRS) [5].

Verification of functional correctness of P4 programs seems to be hot topic lately in the switching industry and the field of network languages, although most works we found were targeting P4$_{14}$ the previous, still maintained version of the language. *P4V* [12] verifies various properties – such that no headers are used unless they were extracted from a packet beforehand – by extending the language with assertion statements, transforming the program code including assertions to predicate transformer semantics, and then applying the Z3 SMT solver to prove theorems. *Vera* [14] follows a different route, and uses symbolic execution and computation tree logic over an intermediate representation to find or prove non-existence of bugs in P4 programs, delivering also an input configuration producing the fault. *P4K* [10] is a formal semantics for P4, written in the K framework. Using reachability logic in K, the authors automatically prove Hoare-style assertions for P4 involving stateful data plane elements and unbounded streams of packets.

Both in correctness and performance verification a core concern is efficiency: for deriving the execution cost of a program statically, analysis of all execution paths is unavoidable. This means that the complexity of the analysis is at best exponential. To offset the costs of path analysis, we borrowed a simple divide and conquer idea from [8] to utilize the highly decomposable nature of network programming languages in verification: instead of analysing paths in the full program, we first analyze just the components, and only perform those transformations on the full program that actually require the full program. Thanks to our symbolic and compositional denotation, we can choose an arbitrary small segments for analysis instead of analyzing the full pipeline.

Our work can be considered an automatization of the approach following [3]. Here, the authors manually analyze the Ethernet protocol and a specific hardware architecture, then synthesize the information into a sequence of primitive packet processing actions called *elementary operations (EOs)* in order to quantify performance.

6 Conclusion and Future Work

We conclude this paper by first enumerating problems and opportunities to extend the presented framework in future research, and then summarizing the contributions of this work.

In the current paper, we only analyzed the parser of a P4 program. One important step for full language coverage will be the analysis of match-action tables: match probabilities can be computed from given match-action tables, and low-level costs of matching and actions can be inferred using the presented procedure. On the other hand, the number of branches in the control flow will be equal to the number of distinct actions that can be performed by the table, so to avoid combinatorial explosion with nested branches, it is important to analyse match-action tables separately.

In this paper, we demonstrated the operation of our system on a toy example. It will be an important and useful research task to apply the procedure to real and complex targets, such as the P4C reference switch [1] and the DPDK switch generated by T4P4S [11]. First, to validate the presented system in real environment, and second to use the retrieved performance information to improve the aforementioned targets.

By introducing random variables in time cost formulas, we effectively modelled P4 programs as memoryless Markov-chains. Feasibility of providing conditional probability distributions for more precise models involving Markov-chains with longer memory may also worth further investigation.

At the time of writing, state-of-art compilers such as the official P4C compiler [1] as well as the P4C-based T4P4S reject all P4 programs containing loops in the parser, and the language specification [15] disallows loops everywhere else. For the lack of support in the software ecosphere and a seeming lack of use cases for loops in P4, we decided not to implement cost analysis of loops in the current work, but we intentionally choose a representation that can be extended for this purpose applying approaches involving e. g. generating functions [16], or cost relation systems [5], and also see loop analysis a possible direction towards system completeness.

With this, we conclude our paper. We showed that networked switches have to comply with strict performance requirements, and also observed that unspecified constructs in P4 require low-level, target-dependent information. We outlined the architecture of a cost analysis framework for addressing both problems. We presented a program transformation system based on term rewriting, that is used to derive a symbolic formula, in which the symbols can be substituted in with numeric constants by various queries to deliver the requested performance information. We went through the main steps of this process using a toy example, and showcased a possible application of the symbolic formula to find ideal cache miss ratios for the aforementioned target. We also situated our paper among works related to the verification of P4, and network function cost analysis. Finally, we marked possible directions to improve this work in order to provide a practical solution for analysis and verification of high-efficiency network platfroms.

References

[1] P4C reference compiler for the $P4_{16}$ programming language. https://github.com/p4lang/p4c, 2017. [Online; accessed 30-September-2018].

[2] 7-Zip LZMA Benchmarks. https://www.7-cpu.com/cpu/Skylake_X.html, 2018. [Online; accessed 30-September-2018].

[3] A. Sapio and M. Baldi and G. Pongrácz. Cross-Platform Estimation of Network Function Performance. In *2015 Fourth European Workshop on Software Defined Networks*, pages 73–78, Sept 2015. DOI: `10.1109/EWSDN.2015.64`.

[4] Agner Fog. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. https://www.agner.org/optimize/instruction_tables.pdf, 2018.09.15. [Online; accessed 30-September-2018].

[5] Albert, Elvira, Arenas, Puri, Genaim, Samir, and Puebla, Germán. Cost relation systems: A language-independent target language for cost analysis. *Electron. Notes Theor. Comput. Sci.*, 248:31–46, August 2009. DOI: `10.1016/j.entcs.2009.07.057`.

[6] Albert Gräf. The Pure Programming Language and Library Documentation. https://agraef.github.io/pure-docs/, 2018. [Online; accessed 30-September-2018].

[7] Bosshart, et. al. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014. DOI: `10.1145/2656877.2656890`.

[8] Dobrescu, Mihai and Argyraki, Katerina. Software dataplane verification. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 101–114, Berkeley, CA, USA, 2014. USENIX Association.

[9] Flanagan, Cormac, Sabry, Amr, Duba, Bruce F., and Felleisen, Matthias. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM. DOI: `10.1145/155090.155113`.

[10] Kheradmand, Ali and Rosu, Grigore. P4K: A formal semantics of P4 and applications. *CoRR*, abs/1804.01468, 2018.

[11] Laki, Sándor, Horpácsi, Dániel, Vörös, Péter, Kitlei, Róbert, Leskó, Dániel, and Tejfel, Máté. High speed packet forwarding compiled from protocol independent data plane specifications. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 629–630, New York, NY, USA, 2016. ACM. DOI: `10.1145/2934872.2959080`.

[12] Liu, et al. P4V: Practical Verification for Programmable Data Planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 490–503, New York, NY, USA, 2018. ACM. DOI: `10.1145/3230543.3230582`.

[13] Sivaraman, Anirudh, Kim, Changhoon, Krishnamoorthy, Ramkumar, Dixit, Advait, and Budiu, Mihai. Dc.p4: Programming the forwarding plane of a data-center switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 2:1–2:8, New York, NY, USA, 2015. ACM. DOI: `10.1145/2774993.2775007`.

[14] Stoenescu, et. al. Debugging p4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 518–532, New York, NY, USA, 2018. ACM. DOI: `10.1145/3230543.3230548`.

[15] The P4 Language Consortium. P4_{16} Language Specification. https://p4.org/specs/, 2017. [Online; accessed 30-September-2018].

[16] Wegbreit, Ben. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, September 1975. DOI: `10.1145/361002.361016`.