Research Article

Dániel Lukács*, Gergely Pongrácz, and Máté Tejfel

Control flow based cost analysis for P4

https://doi.org/10.1515/comp-2020-0131
Received Feb 29, 2020; accepted Mar 31, 2020

Abstract: The networking industry is currently undergoing a steady trend of softwarization. Yet, network engineers suffer from the lack of software development tools that support programming of new protocols. We are creating a cost analysis tool for the P4 programming language, that automatically verifies whether the developed program meets soft deadline requirements imposed by the network. In this paper, we present an approach to estimate the average execution time of P4 program based on control flow graphs. Our approach takes into consideration that many of the parts of P4 are implementation-defined: required information can be added in through incremental refinement, while missing information is handled by falling back to less precise defaults. We illustrate application of this approach to a P4 protocol in two case studies: we use it to examine the effect of a compiler optimization in the deparse stage, and to show how it enables cost modelling complex lookup table implementations. Finally, we assess future research tasks to be completed before the tool is ready for real-world usage.

Keywords: data plane, control flow graph, static profiling, refinement, soft real-time

1 Introduction

In modern network engineering, network scalability is becoming more and more important than raw throughput numbers. This has resulted in the ongoing industry trend known as network softwarization: flexible software nodes and network virtualization are preferred over rigid hardware, even at the price of lower packet processing speed. This trend gave birth to a new class of networks called software defined networks (SDN): agile, dynamically reconfigurable networks automatically controlled by software.

An emerging new technology supporting SDN is the P4 programming language [1]. Programs written in P4 are network protocols, executed by smart network switches: P4 programs describe how the switch should manipulate and forward the packet that it receives. Thus, P4-enabled switches can be dynamically reprogrammed (possibly even after deployment) to accommodate changes in network configuration.

An important selling point of P4 is that it alleges to be as fast as fixed protocol set switches (e.g. OpenFlow), and at the same time being capable of expressing custom (arbitrary) protocols. And switch performance is important for networks, even if high end-to-end throughput is less valued these days: when an individual switch repeatedly fails to process incoming packets in time, the delay will cause buffer overflows and network congestions, which in turn can ultimately lead the switch into failing out, or worse, lead the whole network into a breakdown. To paraphrase, switches are real time systems with soft deadlines.

1.1 Motivation

Unlike in the case of hardware switches, where there are well-established methodologies for verifying such requirements, P4 still lacks tooling that could provide information to developers about whether the developed protocol will meet said deadlines or not. Unfortunately, the underlying complexity of software systems, – that makes analysis difficult or even NP-hard in some cases –, is an additional trade-off of softwarization.

Our current work is part of our ongoing effort to tackle this problem: our intention is to develop a cost analysis tool for P4. In our plans, this tool will make it possible to estimate the execution time (and possibly other factors, such as energy efficiency) of a P4 program, enabling verification of soft network deadlines and other requirements. In the future, we also hope to enable other operations, such as proposals of program optimizations, and automatic inference of execution environment parameters required to meet known deadlines.

***Corresponding Author: Dániel Lukács:** Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary;
ORCID: 0000-0001-9738-1134; Email: dlukacs@inf.elte.hu
Gergely Pongrácz: Ericsson Hungary Ltd., Budapest, Hungary;
ORCID: 0000-0002-5115-9973; Email:
Gergely.Pongracz@ericsson.com
Máté Tejfel: Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary; 3in Research Group, Martonvásár, Hungary;
ORCID: 0000-0001-8982-1398; Email: matej@inf.elte.hu

In this work, we present an approach for automatic P4 cost analysis, that can be incrementally refined with new information to improve its precision, and we illustrate this approach and refinement in two case studies. In Section 2, we enumerate related topics in the literature, including our earlier results on this subject. Also in that section, we highlight and describe the most important parts of the P4 language through a code example. In Section 3, we detail our refinable, CFG-based probabilistic representation of P4 programs, and provide an algorithm for performing cost analysis on this representation. Section 4 houses our first, introductory case study: we refine and estimate the cost of the deparsing stage of a P4 program. Section 5 details our second, more involved case study: refinement and analysis of P4 table lookups. We conclude our paper in Section 6.

2 Related work

In our earlier approach to P4 cost analysis [2], we syntactically transformed the code of P4 parsers into cost expressions. In the current paper, we utilized our earlier insights and continued this analysis with deparsing and lookup tables, but our current algorithm is based on CFG traversal. Our main issue with the transformational approach was that syntax-level analysis is more resource-intensive and computational costs of symbolic rewriting are hard to predict: even though it is important for exponential use cases such as ours.

As we noted earlier, P4-enabled switches can be considered soft real-time systems as there are timing requirements in addition to functional requirements. Davis et. al. [3] offers a classification and current survey of approaches for analysing timing requirements. In this classification, our work is considered static probabilistic timing analysis (SPTA).

A mature example of general SPTA theory can be found in Baier et. al [4]. Here, the authors use various Markov-chains for probabilistic analysis of semantic properties (such as state reachability), but also for calculating expected reward. While our work is more focused on application, in the future, we plan to discover the possibilities for improving fundamentals in this work.

A current example for dynamic timing analysis is Iyer et. al [5]. The authors developed Bolt, a tool based on dynamic instrumentation, that generates performance expressions from the C code of network functions. Performance expressions are terms (similar to the expected value formulas in our current work), containing performance critical variables (variables describing factors of hardware, implementation and execution environment that have high influence on performance). While we cannot directly compare this work to our work because of the difference in approach, the capabilities of the tool are astounding and serve as a goal for future improvement.

2.1 About the P4 language

Programs written in the P4 programming language [1] are high-level descriptions of packet pipelines: a sequence of packet processing operations every packet will go through. An example P4 code we will use here and in later sections (discussing cost analysis of deparsing and table lookups) is displayed by Listing 1. This code excerpt is was extracted from the `basic_routing-bmv2` test case of the official P4 reference compiler [6]. It specifies a simple protocol describing basic IP4 forwarding.

Before we proceed to explain this example, it is important to highlight, that in P4 the bulk of packet forwarding work is performed by match-action tables. As a generalization of routing tables, these can be envisioned as key-value stores, where keys are patterns and values are actions. When the table is applied to a packet, the table is searched for the first entry whose pattern matches the packet, and the action of this entry is executed. P4 control flow can also depend on the executed action. Note that P4 does not define the implementation of match-action tables: choosing the optimal data structures and search algorithms is the responsibility of compiler developers. Moreover, P4 programs do not describe the contents of the match-action tables: they are filled by the SDN controller switch during runtime. P4 programs only describe the table schema.

The elements of the forwarding pipeline are specified in the call to `V1Switch`. First, headers of the incoming packet are parsed by `ParserImpl` into the structure named `headers`. Then the `ingress` block specifies that in case the packet was successfully parsed as an IP4 packet, we apply the `fib` match-action table (`exact` lookup, requiring full match) to the packet. If this fails, we apply the more lenient `fib_lpm` table (LPM lookup). The `table` block `fib` declares a table schema: the `dstAddr` field of the packet is matched to table entries, which can be either one of the actions `nexthop` and `on_miss` and their arguments (e.g. the destination port). Action `nexthop` sets the `egress_port` metadata field that is read, in turn, by the switch implementation to forward the packet to the nexthop gateway. Then, `nexthop` decreases the IP4 time-to-live field by the 8-bit unsigned integer 1. The final step of the pipeline is `DeparserImpl`, selecting which headers should be in-

Listing 1: P4 code specifying a simple IP4 switch

```
#include <core.p4>
#include <v1model.p4>

struct headers { ethernet_t eth;
                 ipv4_t  ipv4;
}

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

header ipv4_t { bit<8> ttl;
                bit<32> dstAddr;
                [...]
}

V1Switch(p = ParserImpl(),
         ig = ingress(),
         dep = DeparserImpl()) main;

parser ParserImpl [...] { [...] }
control ingress(
 inout headers h,
 out standard_metadata_t sm) {

action on_miss() { }

action nexthop(bit<9> port) {
 sm.egress_port = port;
 h.ipv4.ttl     = h.ipv4.ttl - 8w1;
}

table fib {
 actions = { on_miss; nexthop; }
 key = { h.ipv4.dstAddr : exact; }
 size = 131072;
}

table fib_lpm {
 actions = { on_miss; nexthop; }
 key = { h.ipv4.dstAddr : lpm; }
 size = 16384;
}

apply {
 if (h.ipv4.isValid()) {
    switch (fib.apply().action_run) {
     on_miss: { fib_lpm.apply(); }
    }
  }
 }
}
// end of control ingress


control DeparserImpl(
  packet_out packet,
  in headers hdrs) {

 apply {
   packet.emit(hdrs.eth);
   packet.emit(hdrs.ipv4);
 }
}
```

cluded in the outgoing packet. After this step, these headers are copied in front of the payload and the switch sends out the packet.

3 A probabilistic model of acyclic program flow

One factor posing considerable difficulty in cost analysis of P4 is that many elements in the language are undefined in the language specification, by design. This is one of the key features enabling highly abstract P4 programs to be competitive performance-wise with lower-level switches: P4 compiler developers can choose the best implementation of these undefined elements for their target switch-platform.

We handle this analysis problem through an incremental refinement approach. We treat P4 programs as abstract descriptions (specifications or models) of what kind of packet processing behavior is expected from the switch. To perform cost analysis, parts of this abstract description must be refined or concretized by including more information, e.g. about the executing hardware, the language implementation, and the runtime environment. The more concrete the description is, the more precise our estimations will be. On the other hand, adding concrete information will restrict the range of targets to which our model can be applied.

In the following section, we provide a probabilistic model of program execution. We then give an algorithm for calculating the expected execution cost of a program given its control flow graph (CFG). For now, we will assume that the modelled programs have no explicit loops. We exclude looping constructs from the current discussion, because (a) in the general case, the expected value of a loop is not computable, and (b) because unstructured loops (such as those constructed using jump instructions) would make analysis more involved. As we will also see in Section 5, special cases of structured loops can be easily included into the model, and this is sufficient for P4. We also omit error statements from our discussion: defining the meaning of execution cost of a program resulting in error is out of the scope of this paper. Moreover, the P4 language lacks exception handling control structures, so this omission only affects a small class of P4 programs.

3.1 The model

In the following, we informally assign to CFGs a semantics similar to the semantics of Bayesian networks. During this discussion, we will use the CFG in Figure 2 of Section 5 as an example. Nodes of the CFG, called blocks (denoted as n), correspond to a value of the random variable called *program counter*: each one of these values has an associated execution cost. We treat CFGs as hierarchical, so we allow blocks to be mapped to lower level CFGs. A directed edge e_i between two blocks corresponds to the possible event (also denoted as e_i) of updating the program counter. Conventionally, we denote the probability of this event as $P(e_i)$. A finite *program execution* (denoted by π) is any directed path (i.e. sequence of events) selected from the CFG starting at the entry point and terminating in the exit point. Each prefix of a program execution has an associated program state. Each e_i edge has a special condi-

tion label $cond(e_i)$. If a node has multiple outgoing edges, the control always chooses the edge with the condition satisfied in the current program state. As a consequence: $P(e_i) = P(cond(e_i))$. As such, we use $P(e_i)$ and $P(cond(e_i))$ interchangeably in this paper.

We define the expected value of a g CFG as the sum of the execution costs of each execution path, weighted by the probability of that execution:

$$E(g) = \sum_{\pi \in paths(g)} P(\pi) cost(\pi) \tag{1}$$

Since conditions depend on the program state, not all conditions are independent from each other. In other words, the probability of a condition being satisfied is a conditional probability. By the definition of conditional probability, the probability of a length-2 execution path e_1, e_2 can be calculated using the individual probabilities of the constituent elements: $P(e_2 \cap e_1) = P(e_2|e_1)P(e_1)$. For paths longer than that, we can decompose the probability of a path using the chain rule of conditional probability:

$$\begin{aligned} P(e_n, e_{n-1}, \ldots, e_2, e_1) = P(e_n|e_{n-1}, \ldots, e_2, e_1) \\ \cdot P(e_{n-1}|e_{n-2} \ldots, e_2, e_1) \\ \ldots \\ \cdot P(e_2|e_1) \\ \cdot P(e_1) \end{aligned} \tag{2}$$

When we talk about the cost of some $\pi = e_1, e_2, \ldots, e_k$ path, we mean the sum cost of the blocks on that path. As before, we need to take into account that the execution cost of block n_i may be dependent on the path it is on:

$$cost(\pi) = \sum_{n_i \in nodes(\pi)} cost(n_i|\pi) \tag{3}$$

In most cases, we expect expressions in the form of $cost(n|\pi)$ to be directly translated to known constants (elementary operational costs), or to conditional expectations in the form of $E(n|\pi)$.

Conditional expectation provides us with compositionality, which is important for efficiently computing expectation. Let g be CFG, and n be a component (a block which is mapped to a sub-CFG) of g. Let us consider random variables $X : \Omega \to paths(g)$ and $Y : \Omega \to paths(n) \cup \{\varepsilon\}$. Assuming n is deterministic, each π_y value of Y corresponds to a class of inputs of n. Inputs are generated by values of X, and so $rng(Y)$ determines a partitioning of $rng(X)$.

We should also notice that the component has no cost on those paths that do not execute it: $E(n|Y = \varepsilon) = 0$. Moreover, path π_y is only executed on π_x if and only if π_x generates an input inducing π_y:

$$P(Y = \pi_y|X = \pi_x) = \begin{cases} 1, & \text{if } \pi_x \text{ induces } \pi_y \\ 0, & \text{otherwise} \end{cases}$$

Then, by the law of total expectation,

$$\begin{aligned} E(n|X = \pi_x) &= \\ E(E(n|Y = \pi_y)|X = \pi_x) &= \\ \sum_{\pi_y \in rng(Y)} E(n|Y = \pi_y) P(Y = \pi_y|X = \pi_x) &= \\ \sum_{\pi_y \in paths(n)} E(n|Y = \pi_y) \end{aligned}$$

This means, we can calculate $E(n|\pi_y)$ independent of π_x, and use these to calculate $E(n|\pi_x)$. The law in the form of $E(n) = E(E(n|\pi_y))$ also justifies our handling of the top-level component in Equation 1. Simple examples (assuming independence of component cost and preceding execution) of utilizing such expressions will be demonstrated in Section 4 and Section 5.

3.2 Algorithms for computing expected value

Algorithm 1 enumerates all paths and computes the probabilities and costs for each. Unfortunately, we cannot

Algorithm 1: A revisiting BFS, calculating the expected value of an acyclic CFG G from source node r

Input: G `// an acyclic CFG`
Input: r `// starting node for traversing the CFG`
Output: A list containing for each path in the CFG a (n, π, p, s) tuple characterizing the path, where
- n is the last block on the path
- π is the path itself, i.e. a sequence of CFG edges
- p is the probability of the path
- s is the cost of the path

```
1  Function ExpectedValue_G(r):
2      out := ∅
3      q := [(r, [], 1, 0)]
4      while q ≠ [] do
5          (n, π, p, s) := Dequeue_q()
6          foreach e in OutEdges_G(n) do
7              m := Dst_G(e)
8              data := (m, [e] ++π, p · P(e|π), s + c(m))
9              if OutEdges_G(m) = ∅ then
10                 out := out ∪ {data}
11             else
12                 Enqueue_q(data)
13             end
14         end
15     end
16     return out
```

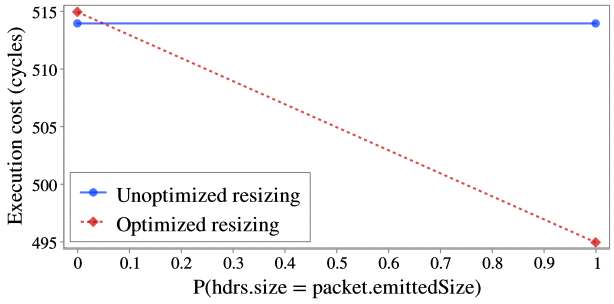

Figure 1: Cost of `resize` and `resize*`

avoid enumerating all (exponentially many) execution paths induced by the CFG: if two paths merge (e.g. as in an Y-shaped component), we still need to record the different histories of the suffix in the probability conditions. On the other hand, we could take advantage of the fact that paths share some prefixes with each other in order to avoid redundantly computing probabilities and cost sums for the same prefixes. That is, given the weighted costs $(p_i p_{i+1} \dots p_n P_1)c_1$ and $(p_i p_{i+1} \dots p_n P_2)c_2$ of two paths, we can calculate the expected value as $(p_i p_{i+1} \dots p_n)(P_1 c_1 + P_2 c_2)$.

The algorithm can be optimized further by e.g. pruning low-probability branches, and selectively reversing the direction of traversal (starting from the exit).

3.3 Data requirements

In P4, implementation of certain language elements is intentionally left undefined by the specification, so the implementors can maximize efficiency on very different executing hardware targets. Moreover, some input data is naturally only available at runtime (e.g. packets, and match-action table contents). This missing information is required to calculate the elementary costs and conditional probabilities in the expected value formula of Equation 1. But this is also the exact reason we chose this approach: new information regarding costs and probabilities can be easily added by the time and in the quality it becomes available. Until then, we can utilize sane, but far less precise defaults: we may treat all conditionals independent from each other (thus taking advantage of the fact that $A \cap B = \emptyset \implies P(A|B) = P(A)$).

This is consistent with the meaning of independence stating that any value of each variable contains no information regarding values of the other variable. We can also assign mathematical (e.g. uniform) probability distribution for each conditional. Or we may decide to calculate conditional probabilities using static analysis techniques, or infer them with data mining on existing usage data.

We can also refine the abstraction level of the CFG (and achieve more precise results) by expanding a node into the CFG of its implementation (or a selected abstraction of its implementation). We illustrate this refinement in the following sections. In Section 4, we discuss the relatively straightforward packet deparsing, and analyze the cost trade-off introduced here by a possible compiler optimization step. In Section 5, we analyze the costs of (undefined) P4 table lookups by assuming that the target implements lookups using a specific algorithm, called DIR-24-8.

4 Case study: Packet deparsing

Deparsing is the final step of the packet processing pipeline, the "inverse" operation of parsing (which we already discussed in [2]). In `DeparseImpl` in Listings 1, we select the headers to be included in the outgoing packet. These headers, together with the payload, are then copied to a location from where the packet will be finally transported outside the machine by the NIC.

Implementation of P4 deparsing is not specified by the P4 specification: P4 compiler developers can choose the approach that runs the fastest on the target switch hardware. As a result, this information is specific to each compiler, and thus it is simply not available to us. To mitigate this problem, we have to refine the deparsing stage, by providing a more concrete model of the implementation (i.e. the runtime code generated by the compiler).

We model compiler generated implementations of the deparser with the function `deparse` in the pseudocode in Listings 2. In the code, we assume that maximal size of `packet.emitted` is known compile-time and is allocated before deparsing (in P4, all sizes have a known upper bounded, by design). We also assume there is enough space allocated for adjusting `packet.cursor` back and forth. For now, we also assume the payload is already in place, and we do not model the transmission.

In function `deparse`, the involved structures are read from the main memory, the two `emit` statements in `DeparserImpl` (selecting headers for the outgoing packet) are executed, the size of the store storing the outgoing packet is actualized, and finally the headers are copied to this store. The implementation of `emit` stores a pointer (pointing to header instance, residing already at some temporary store) in an array and keeps count of the size in bytes. Function `resize` calculates the difference between

Listing 2: Pseudocode illustrating compiler generated deparser code

```
void deparse(PacketOut packet,
             Headers hdrs){
  cache.ensure(packet, hdrs);
  emit(packet, hdrs.ethernet);
  emit(packet, hdrs.ipv4);

  resize(packet, hdrs.size);
  memcpy(packet.cursor,
         packet.emitted,
         packet.emittedSize);
}
void emit(PacketOut packet,
          Header hdrRef){
  packet.emitted[packet.numEmitted] =
      hdrRef.raw;
  ++packet.numEmitted;
  packet.emittedSize = packet.emittedSize +
      hdrRef.size;
}


void resize(PacketOut packet,
            int oldSize){
//if(oldSize==packet.emittedSize){
//  return;
//}
  int diff = oldSize - packet.emittedSize;
  packet.cursor += diff;
}
```

size of the headers of the incoming packet, and size of the now emitted headers. The cursor pointer (where the emitted headers will be copied) is adjusted forward or backward so that the headers fit tightly before the payload.

It is important to note that compilers may generate code that diverges from this model. Here, we assumed that the packet headers are extracted into temporary stores in `Headers` during parsing (as is done by our model in [2], and also by the P4 reference switch [6]). As such, in the deparsing stage we only had to move around pointers without copying larger data (except in the last line). On the other hand, for example, the T4P4S compiler [7] sets pointers during the parsing (without copying anything), and thus it has to introduce a temporary store to perform the deparsing. This means that we need to use a slightly different deparsing model to estimate costs in P4 reference switch, and in T4P4S.

Using the pseudocode, we can now easily generate a cost formula per Equation 1, characterizing the average execution cost of function `deparse`. Using the data in Table 2 of Section 5.2 (partly utilized also in our earlier work [2]), we map the statements to elementary instructions whose costs are known, and sum up the results. As the control flow is linear, there is only one path and it is sure to be executed. During cost analysis, we purposefully avoid considering the cost of function calls: as these can be inlined, calls to separate function definitions can be treated as notation.

Equation 4 depicts the derived cost formulas for each function. We parameterized `memcpy` with the size 34: this is the size (in bytes) of the `headers` structure in Listings 1, and we emit both of its fields. This information can also be inferred using simple static analysis. We included no unknowns in the model, and the execution cost of `deparse` can be easily calculated by substituting in the known constants: it is 514 CPU cycles.

Note that `resize` in Listings 2 includes a commented section. This is a compiler optimization we found in the T4P4S source code: comparison is usually cheaper than arithmetic operations, so – in case outgoing packets are *often* the same size as the incoming ones – avoiding superfluous calculations may pay off.

Yet, it is not evident where is the point at where the extra overhead of the comparison operation starts yielding better performance. This is a question for cost analysis. The last formula in Equation 4 is the cost of `resize` with the commented section uncommented. As the control flow is branching, we now have to weigh each path with its probability. Variable p is the probability $P[size(\text{incoming packet}) = size(\text{outgoing packet})]$, i.e. the probability of early return. (By looking at the P4 code, we can deduce that now this is always 1. But, for the sake of illustration, and since the programmatic deduction of this is still something we have to research in depth, we keep treating p as an unknown.)

We plotted the execution costs of `resize` and `resize*` against various values of p on Figure 1. By solving equation $31(1-p) + 11p = 30$ for the intersection of the two lines, we can see that it is as early as $p = \frac{5}{100}$. So even if just 6 out 100 packets leaves with the same size as which with it came, we are already better off with the optimization.

$$\begin{aligned} E[deparse] &= c_{M\to C} + 2 \cdot E[\text{emit}] + E[\text{resize}] \\ &\quad + cost(\text{memcpy}_{34}) \\ E[emit] &= 5 \cdot c_{C\to R} + 3 \cdot c_{\text{ADD}} + c_{\text{MOV}} + 3 \cdot c_{R\to C} \\ E[resize] &= 3 \cdot c_{C\to R} + c_{\text{DEC}} + c_{\text{ADD}} + c_{R\to C} \\ E[resize^{*}_{p}] &= 2 \cdot c_{C\to R} + c_{\text{CMP}} \\ &\quad + (1-p) \cdot (c_{C\to R} + c_{\text{DEC}} + c_{\text{ADD}} + c_{R\to C}) \end{aligned} \tag{4}$$

5 Case study: Lookup tables

We now proceed with another case study on a more realistic scale. We demonstrate an application of the cost analysis procedure outlined earlier by examining how the cost of one LPM lookup changes for different parametrisations given data extracted from P4 source code. Specifically, we analyze the invocation `fib_lpm.apply()`, that will perform a longest prefix match on table `fib_lpm`.

5.1 Cost models of lookup algorithms

The P4 specification does not specify how lookup tables should be implemented: the actual packet matching algorithms and data structures are selected by the P4 compiler. We will refine table applications using two concrete models: one for linear search, and one for DIR-24-8, the longest prefix match (LPM) algorithm used by DPDK [8], which is in turn the primary target platform of the T4P4S [7] P4 compiler.

Since most bounded loop analysis problems are exponential in the bound, we choose an approach to predefine solution templates for some specific loops.

For now, our purpose is to illustrate how to extend the probability model in Section 3, so we use simplified models. We plan to evaluate and improve these models with real-world P4 compilers in future work.

5.1.1 Linear model for LPM lookup

First, we consider modeling the execution cost of the simplest search algorithm: linear search. Since LPM is basically a search for maximum, we have to assume that the table is lexicographically sorted with decreasing mask lengths. It is unlikely for any real world application to use linear search for lookup: we feature it here because it is simple, fundamental, and we intend to demonstrate how to include loops in the presented model.

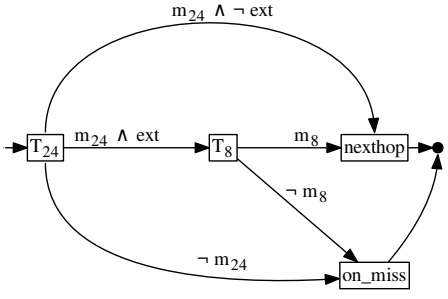

Figure 2: A flowchart illustration of the DIR-24-8 algorithm

Listing 3: Pseudocode illustrating linear lookup

```
1 tbl := memory.open(tbl);
2 for(r in tbl){
3     cache.ensure(r);
4     b := r.match(hdr);
5     if(b.isMatch)
6         return b;
7 }
```

Consider the pseudocode in Listing 3. First, we read a cache line sized chunk of the table into memory, start the search loop and, in case we left the cached part of the table during the search, we cache the next chunk of the table. The average execution cost of this simple algorithm can be approximated using the following formula:

$$\begin{aligned} &E[L(n,p,e,c_{M\to C},c_m)] \\ &= \sum_{i=1}^{n} iG_p(i)(c_m + q(n,e)c_{M\to C}) \\ &= \frac{c_m + q(n,e)c_{M\to C}}{p} \end{aligned} \tag{5}$$

In Equation 5, n stands for the number of entries in the match-action table, e for the size of the pattern to be matched, and – since, we do not have any information on arriving packets and tables – we can chose $G_p(i)$ to be the geometric distribution: $G_p(i) := (1-p)^{i-1}p$. Here, $G_p(i)$ the probability of the exit condition (successful match of the packet) failing $i-1$ times and succeeding the ith time, where p is the constant probability of the exit condition succeeding. Term c_m is the cost of matching an entry, $c_{M\to C}$ is the cost of caching a cache line sized chunk of memory, and $q(n,e)$ is the probability that caching must be performed in the ith iteration. Note, that we decided to make these members independent of the loop index.

We can approximate $q(n,e)$ using amortized analysis. The number of rows fit in the cache is $n_{\text{cache}} := \lfloor \frac{\text{size}_{\text{cache}}}{e} \rfloor$, and in the worst case, we need to perform caching $w := \lceil \frac{n}{n_{\text{cache}}} \rceil$ times. In average, we cache $\frac{w}{n}$ times per iteration.

Table 1: Characteristics of each path in Figure 2

ID	Probability	Cost
π_1	$P(m_{24} \wedge ext, m_8)$	$c(T_{24}\|\pi_1) + c(T_8\|\pi_1) + c(nexthop)$
π_2	$P(m_{24} \wedge ext, \neg m_8)$	$c(T_{24}\|\pi_2) + c(T_8\|\pi_2)$
π_3	$P(m_{24} \wedge \neg ext)$	$c(T_{24}\|\pi_3) + c(nexthop)$
π_3	$P(\neg m_{24})$	$c(T_{24}\|\pi_4)$

Table 2: Example configuration specification used for lookup cost analysis

Symbol	Value	Meaning	Knowledge source
w	4B	CPU word length	Hardware configuration
$c_{M\to C}$	279	Cost of read from memory to cache	Hardware configuration
$c_{C\to R}$	5	Cost of read from cache to CPU register	Hardware configuration
$c_{R\to C}$	5	Cost of read from CPU register to cache	Hardware configuration
$cost(\text{memCpy}_v)$	$\lceil \frac{v}{w} \rceil (c_{C\to R} + c_{\text{MOV}} + c_{R\to C})$	Cost of copying a value	Implementation
$cost(\text{binDec}_v)$	$\lceil \frac{v}{w} \rceil (c_{C\to R} + c_{\text{ADD}} + c_{\text{POW}} + c_{R\to C})$	Cost of transforming IP4 address to array index	Implementation
$cost(\text{arrIdx}_v)$	$c_{M\to C} \lceil \frac{v}{size_{cache}} \rceil$	Cost of array lookup with v sized entries	Implementation
$cost(\text{nexthop}_v)$	$cost(\text{memCpy}_v) + (c_{C\to R} + c_{\text{DEC}} + c_{R\to C})$	Cost of `nexthop` action with v-bit destination port	Implementation
$cost(\text{match}_v)$	$\lceil \frac{v}{w} \rceil (c_{C\to R} + c_{\text{CMP}} + c_{R\to C})$	Cost of matching a v sized header to a table entry	Implementation
p_8	0.1	Probability of arbitrary T_8 entry matching	Inferred from table and the packet distribution. (Now arbitrary.)
p_{24}	0.1	Probability of arbitrary T_{24} entry matching	Inferred from table and the packet distribution. (Now arbitrary.)

5.1.2 DIR-24-8 algorithm

DIR-24-8 was introduced by Gupta et al. [9] as a fast solution for LPM-based IP4 routing. In real networks most packets can be routed using just the first 24 bits: the basic idea is to take advantage of array indexing and cache efficiency by removing masks from the first 24 bits via prefix expansion and applying linear probing to match the last 8 bits.

The algorithm creates in RAM a table (named T_{24}) storing an entry for all (2^{24}) permutations of 24-bit addresses: each entry is either a pointer to T_8 (see below) or a 15-bit next hop address (another 1 bit denotes whether the value is a pointer or a next hop). Another table (named T_8) will match the remaining 8 bits of the address to the next hop. The set of all 24-bit addresses is a sequence between 0 and 2^{24} in base-256: that means T_{24} can be represented as an array with 2^{24} entries, each having a size of 16 bits. Thus, T_{24} will require $2 \cdot 2^{24}$ bytes (32 MiB) of space, while a section in T_8 corresponding to a 24-bit prefix will require at most $2^8(1+2) = 768$ bytes. The main advantage of the algorithm is that looking up the first 24 bits in T_{24} has constant cost (the address changed from base-256 to decimal, or some other base used for array indexing), and – if the entry contains a pointer – we only need to read an additional 768-byte parcel of T_8, which easily fits even the smallest L1 caches. In this paper, we assume that T_8 lookups are performed using linear search: we expect practical implementations to use more efficient lookup schemes. Figure 2 depicts a schematic CFG of DIR-24-8. As in Section 3, we calculate the expected value as a weighted sum of the CFG paths. The products and sums resulting from executing Algorithm 1 on Figure 2 are displayed by Table 1. Below, let $\pi_1 := m_{24} \wedge ext \wedge m_8$, $\pi_2 := m_{24} \wedge ext \wedge \neg m_8$, $\pi_3 := m_{24} \wedge \neg ext$, and $\pi_4 := \neg m_{24}$. For now, we assume m_{24}, ext, and m_8 are independent, and as such we will use the notation $r_8 := P(m_8|ext, m_{24})$, $r_{ext} := P(ext|m_{24})$, and $r_{24} := P(m_{24})$.

Let p_{24} and p_8 be the probability that an arbitrary entry is matching in T_{24} and T_8 respectively. Then $r_8 = 1-(1-p_8)^{2^8+1}$ and $r_{24} = 1-(1-p_{24})^{2^{24}+1}$.

It follows that $P(\pi_1) = r_8 r_{ext} r_{24}$, and $P(\pi_2) = (1-r_8)\, r_{ext}\, r_{24}$, and $P(\pi_3) = P(m_{24}, \neg ext) = (1-r_{ext})r_{24}$, and $P(\pi_4) = (1-r_{24})$. As a result, only the probabilities r_{ext}, p_8, p_{24} need to be supplied as input.

$$
\begin{aligned}
&E[\text{LPM}_{\text{DIR-24-8}}(p_{24}, p_8, r_{ext})] \\
&= P(\pi_1)(E(T_{24}|\pi_1) + E(T_8|\pi_1)) \quad + \\
&\quad P(\pi_2)(E(T_{24}|\pi_2) + E(T_8|\pi_2)) \quad + \\
&\quad P(\pi_3)E(T_{24}|\pi_3) \quad + \\
&\quad P(\pi_4)E(T_{24}|\pi_4) \quad + \\
&\quad (P(\pi_1) + P(\pi_3))cost(\text{nexthop}_9)
\end{aligned}
\tag{6}
$$

where

$$
\begin{aligned}
E(T_{24}|\pi_1) &= cost(\text{binToDec}_4) + cost(\text{arrIdx}_2) \\
E(T_{24}|\pi_4) &= E(T_{24}|\pi_3) = E(T_{24}|\pi_2) = E(T_{24}|\pi_1) \\
E(T_8|\pi_1) &= E[L(2^8, p_8, 3, c_{M\to C}, cost(\text{match}_1)] \\
&= \frac{cost(\text{match}_1) + q(2^8, 3)c_{M\to C}}{p_8} \\
E(T_8|\pi_2) &= (2^8+1)(cost(\text{match}_1) + q(2^8, 3)c_{M\to C}))
\end{aligned}
$$

In the final formula, we only have to substitute runtime information for p_8, p_{24}, and r_{ext}, and hardware or implementation specific information $cost(\text{binToDec}_4)$, $cost(\text{arrIdx}_2)$ $cost(\text{match}_1)$, $c_{M\to C}$, $cost(\text{nexthop}_9)$. Examples for these can be seen in Table 2.

For T_{24} lookup, we assumed the cost consists of two substeps: transforming the address to an array index, and performing an array lookup in memory. As we modeled T_8 lookup using linear search, on π_1, we use Equation 5 and prepare for an early exit. On π_4, we use the information from π_4 that T_8 had to be read through in its entirety, i.e. the probability of the algorithm succeeding in the ith step

is:

$$P(exit(i)|\pi_4) = \begin{cases} 1 & \text{, if } i = 2^8 + 1 \\ 0 & \text{, otherwise} \end{cases}$$

5.2 Evaluation

We now evaluate the cost formula with specific parameters and inspect how parameters change program behavior. Using the data (partly utilized also in our earlier work [2]) in Table 2 to instantiate the formula in Equation 6, we obtained the graph in Figure 3. This plot exemplifies how increasing the cache size from $128B$ to $1024B$ results in decreasing execution costs. While L1 caches of this size are superseded today, translating DIR-24-8 to data larger than IP4 addresses also requires larger caches. We should also remember that real world implementations of DIR-24-8 are unlikely to use linear search for T_8 lookup. Larger cache means fixed sized data can be loaded in less memory reads, yet it is not evident how important is caching in the overall cost. By calibrating control event probabilities, we can observe that different program paths gain more weight in the expected cost. In the figure, we experiment with increasing the probability of an address requiring T_8 lookup (*ext* event). According to Gupta et al. [9], 99.93% of prefixes in IP4 backbone routers in 1998 were at most 24 bits long, that is, the probability that a random address must be matched to a more than 24 bits long prefix were 0.0007. The plot confirms that for this kind of packet distribution DIR-24 is very efficient: the expected cost is just slightly above the cost of one memory read. As expected, the cost grows fast with every increase in T_8 lookup probability, suggesting that the worst case cost of the algorithm is very high. Note that if matching an entry costs only as much as 11 CPU cycles, reading through $2^8 = 256$ entries is already 2816 cycles, and we did not even factor in the required caching. For comparison, a 4.3GHz processor operating in a 10 Gigabit Ethernet network can spend only 288 cycles to forward a packet without risking buffer overflow [2]. Another property of DIR-24-8 illustrated by this plot is that cache size only becomes critical when T_8 lookups have a high probability (as T_{24} lookup only requires constant one memory read). Already, the cost difference between a 128B and a 1KB sized cache is non-negligible.

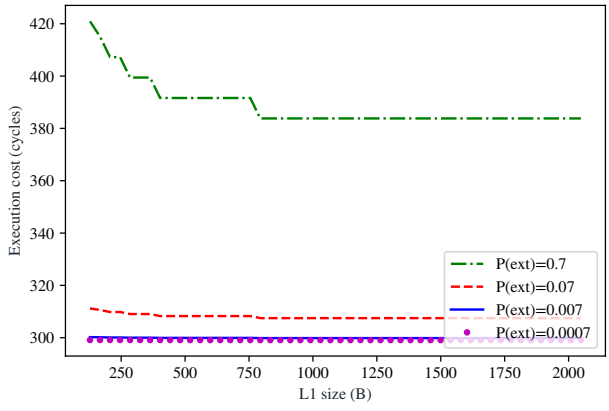

Figure 3: Cost of `fib_lpm.apply()` over cache size and prefix lengths

6 Conclusions

We now conclude this paper. We shared our vision of a cost analysis tool that enables P4 developers to automatically verify in development time that the program under construction satisfies soft deadline requirements imposed by the network. As the core component of this tool, we introduced a control flow based approach, together with an algorithm, for estimating expected execution time. In principle, our algorithm enumerates all execution paths, and calculates a probability-weighted sum of the costs of each path.

An important practical feature of this approach was that it allows incremental refinement: as many parts in P4 are implementation-defined, precise cost estimation necessitates the ability to extend the description with concrete information. If this information is not available, our model can still fall back to less precise defaults. We illustrated these refinement capabilities and the CFG algorithm by two case studies: the cost analysis of deparsing and of DIR-24-8 based lookup table application, each leaning on a simple model of the execution environment. While our approach is still far from being production-ready, we are hopeful that now we have the basic scaffolding on which we can build the aforementioned cost analysis tool.

6.1 Future work

During this article, we left several questions unanswered, which require future research before we can develop a tool that P4 network engineers can use in practice.

We have not yet compared estimation capabilities with real-world P4 compilers. We intend to validate our approach by trying to predict runtime characteristics of executables generated by T4P4S and P4C.

We hinted that component CFGs in hierarchical CFGs can be analyzed separately (and thus, efficiently) if their initial state is known. As it is generally not known, we need

to find a method for inferring it, or at least inputting it (or risk exponential explosion in the cost analysis runtime).

During refinement, we derived the cost formulas from an informal description of the concrete implementations by hand. We are searching for a unified computational model that can express refining components and execution environment parameters, and at the same its cost can be analyzed by our presented approach.

Finally, we plan to create a layer for filling in the missing parameters automatically, e.g. by inspecting the probability distributions of inputted packet traces and linking them with conditionals in the P4 code.

Acknowledgement: The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

References

[1] Pat Bosshart, Dan Daly, Martin Izzard, Nick McKeown, Jennifer Rexford, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[2] Dániel Lukács, Gergely Pongrácz, and Máté Tejfel. Keeping P4 Switches Fast and Fault-free through Automatic Verification. *Acta Cybernetica*, 24(1):61–81, May 2019.

[3] Robert Davis and Liliana Cucu-Grosjean. A Survey of Probabilistic Timing Analysis Techniques for Real-Time Systems. *Leibniz Transactions on Embedded Systems*, 6(1):60, 2019.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[5] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. Performance contracts for software network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 517–530, Boston, MA, February 2019. USENIX Association.

[6] The P4 Language Consortium. P4C reference compiler for the $P4_{16}$ programming language. https://github.com/p4lang/p4c, 2017. [Online; accessed 28-February-2020].

[7] Sándor Laki, Dániel Horpácsi, Péter Vörös, Róbert Kitlei, Dániel Leskó, and Máté Tejfel. 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.

[8] Intel Corporation. LPM Library, Chapter 24 in DPDK Documentation Programmer's Guide. https://doc.dpdk.org/guides/prog_guide/lpm_lib.html, 2014. [Online; accessed 12-May-2019].

[9] P. Gupta, S. Lin, and N. McKeown. Routing lookups in hardware at memory access speeds. In *Proceedings. IEEE INFOCOM '98, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No.98*, volume 3, pages 1240–1247 vol.3, March 1998.