electronics

MDPI

Article

Model Checking-Based Performance Prediction for P4

Dániel Lukács [1,*], Gergely Pongrácz [2] and Máté Tejfel [1]

[1] Faculty of Informatics, Eötvös Loránd University, 1117 Budapest, Hungary; matej@inf.elte.hu
[2] Ericsson Hungary, 1117 Budapest, Hungary; Gergely.Pongracz@ericsson.com
* Correspondence: dlukacs@inf.elte.hu

Abstract: Next-generation networks focus on scale and scope at the price of increasing complexity, leading to difficulties in network design and planning. As a result, anticipating all hardware- and software-related factors of network performance requires time-consuming and expensive benchmarking. This work presents a framework and software tool for automatically inferring the performance of P4 programmable network switches based on the P4 source code and probabilistic models of the execution environment with the hope of eliminating the requirement of the costly set-up of networked hardware and conducting benchmarks. We designed the framework using a top-down approach. First, we transform high-level P4 programs into a representation that can be refined incrementally by adding probabilistic environment models of increasing levels of complexity in order to improve the estimation precision. Then, we use the PRISM probabilistic model checker to perform the heavy weight calculations involved in static performance prediction. We present a formalization of the performance estimation problem, detail our solution, and illustrate its usage and validation through a case study conducted using a small P4 program and the P4C-BM reference switch. We show that the framework is already capable of performing estimation, and it can be extended with more concrete information to yield better estimates.

Keywords: P4; performance prediction; cost analysis; PRISM; model checking; static analysis

Citation: Lukács, D.; Pongrácz, G.; Tejfel, M. Model Checking-Based Performance Prediction for P4. *Electronics* **2022**, *11*, 2117. https://doi.org/10.3390/electronics11142117

Academic Editors: Elisa Rojas, Sándor Laki and Christian Esteve Rothenberg

Received: 31 May 2022
Accepted: 1 July 2022
Published: 6 July 2022

1. Introduction

1.1. Background

Next-generation computer networks must solve a serious problem. On the one hand, they need automatization (programmability and virtualization) in order to be scalable and satisfy the diverse demands of a rapidly growing range of applications (in the cloud, 6G, IoT, etc.), while on the other hand, they depend on specialized hardware resources to maximize throughput and optimize costs. As a result, several technological trends have emerged in the last decade, such as software-defined networking (SDN), network function virtualization, hardware offloading, and programmable switches.

For our work, arguably the most important of these developments is the introduction of the P4 programming language [1]. P4 enables network operators to write arbitrary, SDN-capable network protocols in a high-level, domain-specific language (in contrast to writing them in low-abstraction, error-prone languages such as C) while retaining high performance by taking advantage of hardware offloading. P4 runs on both programmable hardware switches (e.g., Intel Tofino can run P4 with line rate, relying on TCAM to perform a lookup in SDN control tables) and on virtual switches (e.g., T4P4S [2] compiles P4 to DPDK, a networking library enabling direct interaction with the NIC through bypassing the Linux kernel).

1.2. Objectives

Unfortunately, this evolution had a price in the form of steadily rising complexity. First, network designers and business decision makers have to take into account a very large number of interconnected parameters. Second, complexity makes it difficult to ensure

that the network is working correctly and efficiently (i.e., the functional and non-functional requirements of the application are satisfied).

The long-term objective of our ongoing work is to develop a tool that assists in automatically checking network software against the functional and non-functional requirements in silico, (i.e., without the need to deploy the software into an actual network). Such a tool can prove useful during the whole life cycle of the network. In the design phase, network designers can use it to mix and match existing components and make buying decisions regarding a network that does not exist yet. In the implementation phase, developers can validate the efficiency of their implemented network protocols without having to deploy the protocol in a real network. In the operational phase, such a tool can be used to predict behavior and prevent or lower network downtime.

1.3. Existing Methods

The classical approach to requirement evaluation in networks is testing (which we also refer to as "dynamic analysis"), and it is still valid in many cases both in industry and research. Practitioners of this approach are the main audience of [3], proposing a set of performance standards in order to compare P4 compilers and to help compiler developers evaluate their optimizations by comparing the measured performance of the generated code against the standards.

Yet, with the scale and complexity of next-generation networks, in-house testing with scripted benchmarks became increasingly time-consuming and, as is it requires expert personnel, very expensive. On the other hand, centralized network control in SDN makes it easier to collect and process data about the network. For this reason, automatic requirement evaluation regarding networks and network components became a popular research question.

An increasingly useful approach to this end is simulation. In [4], the authors solved the problems of handling multiple protocols and a complex network structure in medical IoT networks by proposing an SDN-based network architecture, where prioritization and machine learning-based (ML) load balancing are performed by the SDN controller. They show how the introduction of SDN and ML improved network performance by using the Riverbed network simulator, a sophisticated tool that estimates network performance based on parameters such as the network topology, protocols used, and speed of individual nodes.

Beyond simulation, another common approach is formal verification. The authors of [5] verified updates in SDN networks in real time by formalizing the network topology and functional network policies as Computation Tree Logic (CTL) formulas and passing these to the NuSMV tool. Very similar techniques were applied in the application layer for service-oriented networks and the IoT by [6,7] as well.

The aforementioned works (with the exception of [3]) check the requirements on the network level, treating individual nodes in the data plane as black boxes. Unfortunately, the nodes can also have errors, and this becomes much more prevalent in the world of SDN and P4, where network nodes are updated frequently and with custom programs. The authors of [8] presented a method for performance evaluation and verification of the non-functional properties for P4 programmable switches. Their approach is to synthesize the latency estimates for a program source based on isolated measurements of the selected P4 features. In this work, we intend to tackle this problem as well but with a different approach.

1.4. Approach

Our focus is on developing a framework that takes advantage of the high-level nature of P4 and uses source code-based techniques (e.g., static analysis or model checking) to automatically infer the costs (such as the time cost of execution (i.e., latency)) and properties (e.g., correctness) related to packet processing on the data plane. In this paper, we focus specifically on estimating latency, relying on automatic inference instead of benchmarking.

One important challenge of this endeavor is that to accurately predict latency, one has to take into account not just the parameters of the application (e.g., the P4 data plane, control plane, and expected network traffic) but also the parameters of the execution environment (e.g., machine specifications, the presence of specialized hardware, and software switch implementation). This is especially true for P4, as most of the work during packet processing is spent performing table lookups, whose implementations are architecture-dependent.

To handle this challenge, we attack the problem using a top-down approach. From the P4 program code, we can automatically extract an abstract, but complete understanding of what the program does requires understanding not just the high-level program control flow but also the complete semantics (program execution depends on the program input and also the subsequent program states reached during execution). Then, we can proceed even lower and plug architecture-specific details into this "skeleton". How low we will go and how much we detail the information we have depends on how much information we have (for example, detailed hardware-level information is usually difficult to gather, or even worse, it may be a propriety and withheld by the vendor) but also on our computational capabilities to execute the analysis. In general, more detailed information will lead to better estimates but also requires more computational resources to calculate.

1.5. Contributions

Our contributions in this paper are the following. In Section 2, we describe the previously outlined problem in detail, and introduce a formal notation to help us unambiguously refer to the numerous factors and components involved in the process and its validation. In Section 3, we introduce our performance estimation solution that makes use of the PRISM probabilistic model checker [9] in order to handle the P4 program semantics and integrate architecture-specific execution environment models of arbitrary complexity. In Section 4, we present a case study to illustrate the complete performance estimation process, including data requirement and collection, the parts handled by PRISM, and the validation of the framework. We also briefly evaluate the current capabilities of the framework. We conclude the paper with listing future directions in Section 6. Executable code of the tool is available online (https://github.com/P4ELTE/P4Query, accessed on 5 July 2022).

2. Problem Description

In this section, we first concisely introduce the static cost analysis problem we solve in this paper along with the main obstacles, and then we formalize the problem in terms of basic statistical concepts.

2.1. Informal Problem Description

Our cost analysis tool estimates the latency of programmable switches. Given a target switch and a P4 program, the tool estimates how long it will take the switch to execute the P4 program for one input packet.

There are two main obstacles to delivering a useful, efficient cost analysis system. First, static cost analysis solves the halting problem [10], which means there is no general solution. Even in the case of the P4 programming language, where all executions are guaranteed to be finite, analysis time is exponential. The second obstacle consists of the large amount of unknown (or variable) factors we need to deal with: the input packets that will be processed by the program are unknown, the internals of the machine that will execute the program are unknown, and the implementation of the P4 compiler that generates the machine code is unknown (or at least highly complex). We partially solved this second problem by relying on probabilistic models. For example, instead of dealing with specific input packets, we rely on a probability distribution that tells us the likelihood that we will receive one or another packet as input. Such data are easy to collect or assemble, given one has access to the historical records of network traffic in which the analyzed switch will be deployed. Thus, we envision cost analysis as a two-step process: in

the first, which we call *bootstrapping*, probabilistic data area collected about the unknown factors (independent from the program code), and in the second, the static, deterministic part of the cost analysis is performed using the actual program code and the previously collected probabilistic data.

To validate (test) whether the cost analysis tool gives usable estimates, we need to perform dynamic performance analysis (i.e., take actual measurements (*benchmarks*) by running the program code on the target device and compare these to the estimates).

2.2. Formal Problem Description

Conventionally, benchmarks are performed in a statistical framework. Even though static analysis is fundamentally a problem in formal semantics, we have seen that it is still linked to statistics, since both modeling missing data and validation require statistical sampling. For this reason, we see it as important to precisely specify the static cost analysis problem in the language of statistics as well, and we will also rely on the notation introduced here in later sections. Another motivation behind this effort was to harmonize probabilistic model checking (see Section 2.4) with classic benchmarking. Using these notations, we will define measurement-based and static cost analysis-based estimators side-by-side. This more formal description of the roles of benchmarks, bootstrapping, and cost analysis is given in Table 1. We now give the interpretation of this table.

Table 1. The place of cost analysis among statistical concepts used in conventional benchmarking.

Category	Symbol	Definition
Populations	$\mathbb{I}$	Possible input packets
	Π	Possible execution paths of a P4 program
	$\mathbb{E}$	Possible environment behaviors
	$\Omega = \mathbb{I} \times \mathbb{E}$	Possible executions
Probability distributions	$f_{\mathbb{I}} : \mathbb{I} \to [0,1]$	Probability of input packets (known)
	$f_{\Pi} : \Pi \to [0,1]$	Probability of execution paths (known)
	$f_{\mathbb{E}} : \mathbb{E} \to [0,1]$	Probability of environments (unknown)
	$f : \Omega \to [0,1]$	Probability of executions (unknown)
Random variables	$X : \Omega \to \mathbb{R}_+$	Program execution time (benchmark)
	$Y : \mathrm{Inst} \times \mathbb{E} \to \mathbb{R}_+$	Instruction execution times (bootstrap)
	$Z : \Pi \times \mathbb{E} \to \mathbb{R}_+$	Path execution times (bootstrap)
Population parameters	$\theta_X : \mathcal{P}(\Omega) \to \mathbb{R}_+$	Population min, mean, and max of X
	$\theta_Y : \mathrm{Inst} \times \mathcal{P}(\mathbb{E}) \to \mathbb{R}_+$	Population min, mean, and max of Y
	$\theta_Z : \Pi \times \mathcal{P}(\mathbb{E}) \to \mathbb{R}_+$	Population min, mean, and max of Z
Samples	$S_X : \Omega^n \to \mathbb{R}_+^n$	Benchmark sampling process
	$S_Y : \mathrm{Inst} \times \mathbb{E}^n \to \mathbb{R}_+^n$	Instruction bootstrap sampling process
Estimators	$\widehat{\theta}_X : \Omega^n \to \mathbb{R}_+$	Sample min, mean, and max of X (using S_X)
	$\widehat{\theta}_Y : \mathrm{Inst} \times \mathbb{E}^n \to \mathbb{R}_+$	Sample min, mean, and max of Y (using S_Y)
	$\widehat{\theta}_Z : \Pi \times \mathbb{E}^n \to \mathbb{R}_+$	Sample min, mean, and max of Z (using $\widehat{\theta}_Y$)
	$\vec{\theta}_X : \mathbb{E}^n \to \mathbb{R}_+$	CA min, mean, and max of X (using $\widehat{\theta}_Z$ and $f_{\mathbb{I}}$)

Any program execution is uniquely determined by the program input space (denoted as $\mathbb{I}$)—or the space of the incoming packets in the case of P4—the program code (denoted as the Π set of execution paths in the program), and the execution environment space ($\mathbb{E}$). For validation purposes, we will send in selected packets so we can assume $\mathbb{I}$ is known. (In a real world scenario, $\mathbb{I}$ depends on the surrounding network.) We can also assume that at least an abstract representation of Π is known. Programmers and static analysis tools can completely understand high-level P4 program code. (On the other hand, the true Π would be expressed in terms of the actual post-compilation machine code that is executed by the CPU.) Each element in $\mathbb{E}$ covers all deep-rooted factors such as the machine specs (architecture, cores, and TCAM memory modules), OS choice (e.g., caching behavior and scheduling policies), and software implementation of the virtual switch (e.g., lookup algorithms), including changes in their behavior during execution. As such, $\mathbb{E}$ is generally unknown (although parts of it can be analyzed and made known). Notationally, the rest of the table is to be interpreted in terms of a given $(\mathbb{I}, \Pi, \mathbb{E})$ triple (i.e., an input space, a program code, and an execution environment space). Ω denotes the complete execution space, where each execution depends on a pair of well-understood factors (currently, only the input) and deep factors (the complete behavior of the background environment during the execution). Note that we did not include Π in Ω, since each execution depends on the whole program.

Our knowledge of the probability distribution of these spaces is also partial. The execution path taken by the program is solely dependent on the program input (and the program itself), which means f_Π is completely conditional on $f_\mathbb{I}$, and we can define it as follows:

$$f_\Pi(\pi) = \sum\{\, f_\mathbb{I}(p) \mid p \in \mathbb{I}, \text{ such that input packet } p \text{ triggers path } \pi \,\} \tag{1}$$

Thus, the probability of observing a specific execution f is only dependent on $\mathbb{I}$ and $\mathbb{E}$, making f a joint probability distribution of $f_\mathbb{I}$ and $f_\mathbb{E}$. If we assume the incoming packets are independent from the execution environment (which intentionally does not include the surrounding network), then $f(p, e) = f_\mathbb{I}(p) \cdot f_\mathbb{E}(e)$ holds, but as $f_\mathbb{E}(e)$ is unknown, so is $f(p, e)$.

In performance analysis, we are interested in estimating the specific numeric attributes of $\omega \in \Omega$ executions, such as the time it takes to complete a specific ω. This attribute, denoted by X, coincides with the concept of latency. Other attributes (energy used, profit made, etc.) can also be of interest and be estimated similarly, provided we have the means to sample them.

We will also observe in Section 4 that real hardware rarely exhibits stable latency, making individual measurements practically unpredictable. As such, our goal is to estimate the selected characteristics of X (e.g., the minimum, mean, or maximum latency). Table 1 denotes these characteristics collectively as the population parameter θ_X.

In conventional (dynamic) performance analysis, θ_X is estimated by a process we refer to as benchmarking. We start up the switch, send in random traffic, and sample the latency. S_X denotes one such sampling process, resulting in a sample. In benchmarking, the estimators (denoted as $\widehat{\theta}_X$) use an S_X sampling process to collect a sample and then use a statistic (e.g., the minimum, mean, or maximum) to aggregate the sample into a single number that estimates θ_X.

In static cost analysis, on the other hand, our goal is also to estimate θ_X, but since static analyses are not allowed to execute the program, we cannot directly rely on $\mathbb{E}$. Instead, we rely on select attributes of $\mathbb{E}$, whose set we call a *cost model*. We assume any $e \in \mathbb{E}$ can be decomposed or projected into a set of primitives, and we can express the attributes in terms of these primitives. For example, both [11] and [8] cite the belief that by measuring the execution time of primitive instructions (of varying granularities), we can inductively infer the execution time of a complete instruction sequence. We also rely on the existence and attributes of such primitives, where Y denotes the time it takes to execute an $i \in \text{Inst}$ primitive instruction in a given $e \in \mathbb{E}$ environment. Note that Y is independent of $\mathbb{I}$ and Π. Y only models $\mathbb{E}$, and we have to be able to observe it independent from specific program

codes and inputs. (As such, these observations can be made by third-parties.) As in the case of X, $\mathbb{E}$ is generally unknown and unpredictable, so instead of Y, we have to rely on abstract characteristics of Y (e.g., the minimum, mean, or maximum execution time), denoted by the population parameter θ_Y. The cost analyzer assumes a *bootstrapping* process (preferably conducted by a third-party, such as the switch vendors), which uses an S_Y sampling process to generate a sample and aggregate it into an estimator $\widehat{\theta}_Y$, which can estimate θ_Y. As paths are just sequences of primitive instructions, by knowing the $\widehat{\theta}_Y$ cost of each $i \in \text{Inst}$ primitive instruction, we can extrapolate from these the cost of the execution paths in Π (denoted by $\widehat{\theta}_Z$). Specifically, given an $\underline{e} \in \mathbb{E}^n$ sequence of environment outcomes during sampling, the cost of an arbitrary m-length path $\pi = (i_1, i_2, \ldots, i_m) \in \Pi$ can be calculated as

$$\widehat{\theta}_Z((i_1, i_2, \ldots, i_m), \underline{e}) = \sum_{k=1}^{m} \widehat{\theta}_Y(i_k, \underline{e}) \tag{2}$$

We now understand all the factors the static cost estimator ($\vec{\theta}_X$) is using to estimate θ_X, and we can now give a general overview of the process. First, we have to acquire from the vendor (or generate ourself) the estimator $\widehat{\theta}_Y$ (i.e., a cost model that describes the execution time of each primitive instruction $i \in \text{Inst}$). Using that, we can derive $\widehat{\theta}_Z$, which is the cost of each execution path (Equation (2)). Next, the cost analyzer can use an $f_{\mathbb{I}}$ probability distribution of the input packets to produce f_{Π}, which is the probability distribution of the paths (Equation (1)). Finally, we use the formulas in Table 2 to estimate the θ_X parameter.

Table 2. Comparison of dynamic and static cost estimators.

Parameter (θ_X)	Dynamic Cost Analysis ($\widehat{\theta}_X$)	Static Cost Analysis ($\vec{\theta}_X$)
$\theta_X^{\texttt{min}}(\Omega) = \min_{\omega \in \Omega}(X(\omega))$	$\widehat{\theta}_X^{\texttt{min}}(\underline{\omega}) = \min(S_X(\underline{\omega}))$	$\vec{\theta}_X^{\texttt{min}}(\underline{e}) = \min_{\pi \in \Pi}(\widehat{\theta}_Z^{\texttt{min}}(\pi, \underline{e}))$
$\theta_X^{\texttt{avg}}(\Omega)$ $= \sum_{\omega \in \Omega}(f(\omega)X(\omega))$	$\widehat{\theta}_X^{\texttt{avg}}(\underline{\omega})$ $= \frac{\sum(S_X(\underline{\omega}))}{\lvert S_X(\underline{\omega})\rvert}$	$\vec{\theta}_X^{\texttt{avg}}(\underline{e})$ $= \sum_{\pi \in \Pi}(f_{\Pi}(\pi)\ \widehat{\theta}_Z^{\texttt{avg}}(\pi, \underline{e}))$
$\theta_X^{\texttt{max}}(\Omega) = \max_{\omega \in \Omega}(X(\omega))$	$\widehat{\theta}_X^{\texttt{max}}(\underline{\omega}) = \max(S_X(\underline{\omega}))$	$\vec{\theta}_X^{\texttt{max}}(\underline{e}) = \max_{\pi \in \Pi}(\widehat{\theta}_Z^{\texttt{max}}(\pi, \underline{e}))$

Table 2 collects the static and dynamic estimators for the three population parameters of the execution time (X): the minimum ($\theta_X^{\texttt{min}}$), the mean ($\theta_X^{\texttt{avg}}$), and the maximum ($\theta_X^{\texttt{max}}$) with respect to all possible outcomes (executions). In real-world terms, the minimal and maximal execution times can be interpreted as the upper and lower bounds of the execution time, while the mean execution time tells us what kind of performance we can realistically expect from the program when it is processing a long (possibly infinite) and varied packet stream.

The second column summarizes how these parameters are estimated using conventonial benchmarking. In essence, we take a random sample of execution times ($S_X(\underline{\omega})$, where $\underline{\omega}$ denotes a given sequence of random outcomes) and calculate its minimum, mean, or maximum. For example, we calculate the mean simply by summing up the measured execution times in $S_X(\underline{\omega})$ and dividing by the sample size $|S_X(\underline{\omega})|$. The third column describes how we estimate these parameters in static cost analysis. For example, to estimate the mean execution time, we calculate the weighted average of the path costs ($\widehat{\theta}_Z^{\texttt{avg}}$, itself an estimate, based on dynamic performance analysis as shown in Equation (2)), where each weight is the probability of the path being executed ($f_{\Pi}(\pi)$, calculated statically based on the input probabilities and the program code as shown in Equation (1)).

We may interpret the `min`, `avg`, and `max` labels as a specific performance scenario. A machine that executes a heavy load in the background will exhibit worse performance than a lightly burdened one, and we would like to make separate estimates for each scenarios. For example, $\widehat{\theta}_Z^{\texttt{min}}$ estimates the execution time of a path in the best-case scenario regarding execution environments (i.e., the case when the environment is favorable (no cache misses, no rival background processes, etc.)). $\vec{\theta}_X^{\texttt{min}}$ estimates the execution time when both the

environment and the program input are favorable. Note that combinatorially, we can define nine such static cost estimators altogether. In addition to the three already in Table 2, we could calculate the estimators for favorable inputs in an average environment, favorable inputs in an unfavourable environment, average inputs in a favorable environment, etc. (In Table 3 of Section 4, we define some of these additions along with the dynamic estimators we used to validate them.)

2.3. Limitations

We know of two hidden limitations regarding this model of P4 program execution time. One is that it does not recognize that the input and environment can depend on each other across executions, whereas in a real-world scenario, specific packet streams processed by the switch will have a lasting influence on the environment. A common example of this is MAC learning. Another one is that receiving similar packets in succession can help with optimizing memory allocations (relevant in software switches). Moreover, P4 programs can also store metadata which are preserved across executions. In our model, this can be considered to be covered by $\mathbb{I}$ (in which case $\mathbb{I}$ needs to store the start states and not just the input packets), but this still fails to acknowledge, for example, that the switch can cache the metadata during an execution and take advantage of this during subsequent executions.

The other limitation is more about practicality. P4 allows updating the lookup tables between packets. One possible solution is to include all possible table contents probabilistically, as we do with $f_{\mathbb{I}}$ in the case of packets. Unfortunately, this is most likely unfeasible for all but the simplest use cases, as each table candidate multiplies the number of potential program paths exponentially. Alternatively, we can think of a table update as an operation that changes program Π_1 to some other program Π_2. Since all estimators are defined to predict the execution time of just one (random) execution, our model can handle table updates by simply recalculating everything with Π_2 instead of Π_1 (although program-independent, third-party produced estimators that require sampling—such as $\widehat{\theta}_Y$—do not need recalculation). Here, given n table updates, we need to recalculate everything n times.

2.4. Probabilistic Model Checking

An important component of our cost analysis tool is the PRISM probabilistic model checker [9]. We chose model checking over common static analysis methods (e.g., control flow analysis, which was examined in our earlier work [12]) because these methods usually do not understand program semantics fully and cannot take into account changes in the program state entirely. The reason we chose probabilistic reward-based model checking in particular over other forms of model checking (for example, UPPAAL, which is built around timed automata and used for the verification of real-time requirements (e.g., in [7])) was due to the large number of unknown factors in the case of P4. Probabilistic model checking allows its users to handle these unknowns probabilistically. PRISM offers several algorithms for calculation (performing better or worse depending on the use case) and good documentation. As probabilistic model checking is a less well-known term, we introduce here the most important definitions based on [13] (Chapters 6.2, 10.1, 10.2, and 10.5 of the monograph). In the following paragraphs, we assume our readers already have a basic understanding of temporal logic and the syntax and semantics of construction tree logic (CTL).

CTL formulas are interpreted over transition systems. A **transition system** (TS) is an $(S, Act, \longrightarrow, I, A, L)$ tuple, where S is a set of states, Act is a set of actions for labeling transitions, $\longrightarrow \subseteq S \times Act \times S$ is a set of transitions, $I \subseteq S$ is the set of initial states, A is a set of atomic propositions, and $L : S \to \mathcal{A}$ is a labeling function deciding which states satisfy which atomic formulas.

Probabilistic CTL (PCTL) is an extension of CTL. PCTL formulas are interpreted over discrete-time Markov chains. A **discrete-time Markov chain** (DTMC) is an $(S, \mathbf{P}, \iota, A, L)$ tuple, where S is a countable set of states, $\mathbf{P} : S \times S \to [0, 1]$ is the transition probability function, $\iota : S \to [0, 1]$ is the initial state probability distribution, A is a set of atomic

propositions, and $L : S \to \mathcal{A}$ is a labeling function. The **P** transition probability function can naturally be for paths such that $\mathbf{P}(s_0\, s_1\, \ldots\, s_n) \stackrel{\text{def}}{=} \prod_{i=0}^{n-1} \mathbf{P}(s_i, s_{i+1})$.For every $M = (S, P, \iota, A, L)$ DTMC with a labeling function $l : S \times S \to Act$ over some action set Act, there exists a transition system $TS(M) = (S, Act, \longrightarrow, I, A, L)$ such that $I = \{s \in S | \iota(s) > 0\}$ and $\longrightarrow = \{(s,\, l(s,t),\, t) | P(s,t) > 0\}$.

Probabilistic reward CTL (PRCTL) is an extension of PCTL. PRCTL formulas are interpreted over Markov reward models. Given an M DTMC, a **Markow reward model** (MRM) is a pair (M, r), where $r : S \to \mathbb{N}$ is an arbitrary reward function, assigning a "reward" (or "cost") to each state that is earned whenever that state is left. Given an $s_0\, s_1\, \ldots s_n$ path, the reward function can be naturally generalized to the **path reward** function $r(s_0\, s_1\, \ldots s_n) \stackrel{\text{def}}{=} \sum_{i=0}^{n-1} r(s_i)$.

We define the language of **PRCTL state formulas** (i.e., propositions that make claims about states) as the set of expressions generated by the grammar $\Phi ::= \top \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \forall\varphi \mid \exists\varphi \mid\mid \mathbf{P}_J(\varphi) \mid \mathbf{E}_K(\Phi)$, where p is an arbitrary atomic state formula and J is an interval such that $J \subset [0, 1]$. **PRCTL path formulas** (i.e., propositions that make claims about paths) can be defined by the grammar $\varphi ::= \bigcirc\, \Phi \mid \Phi_1\, \mathbf{U}\, \Phi_2$. Like in CTL, $\bigcirc\, \Phi$ is satisfied in path π if Φ is satisfied in the second (usually called the "next") state on π. $\Phi_1\, \mathbf{U}\, \Phi_2$ is satisfied in π if a state $s \in \pi$ satisfies Φ_2, and all states in π preceding s satisfy Φ_1. State formulas that also appear in CTL have the same meaning. For example, the satisfaction of state formula a in a state đ is given by the L labeling function of the MRM. Similarly, $\forall\varphi$ is satisfied in state s if the path formula φ is satisfied for *all* paths starting from s. It follows that $\forall(\top\, \mathbf{U}\, \Phi)$ is satisfied in state s if Φ is satisfied on each path starting from s. For this reason, we define the syntax $\Diamond\, \Phi \stackrel{\text{def}}{=} \top\, \mathbf{U}\, \Phi$ as well, where $\Diamond$ is usually called the *eventually* operator. Given a π path that satisfies a formula $\Diamond\, \Phi$ with its shortest prefix π', which still satisfies $\Diamond\, \Phi$, we call π' the **minimal path fragment** of π.

Formula $\mathbf{P}_J(\varphi)$ is satisfied in s if the probability that a path starting from s satisfies φ is inside the bounds of the J interval. Formula $\mathbf{E}_K(\Phi)$ is satisfied in s if the expected reward—accumulated over the minimal path fragments of paths that start in s and satisfy $\Diamond\, \Phi$—is in inside the bounds of the K interval.

A PRCTL model-checking problem is the following decision problem: given a state s of a DTMC M and a PRCTL formula Φ, decide whether Φ is satisfied in s. As $\mathbf{P}_J(\varphi)$ and $\mathbf{E}_K(\Phi)$ presume producing a numeric value, we can derive from these formulas $\mathbf{P}_\Delta(\varphi)$ and $\mathbf{E}_\Delta(\Phi)$, which are meant to be solved for the variable Δ. In the case of $\mathbf{P}_\Delta(\varphi)$, the solution is the probability, while in the case of $\mathbf{E}_\Delta(\Phi)$, the solution is the expected reward according to the original definitions. We also consider solving such formulas as part of PRCTL model checking.

We should note that these definitions concern how PRISM works internally. In most of the paper, we are concerned with higher abstraction levels and thus use some of these terms with slight differences in meaning and notation. For example, in the above definitions, the paths connect the program states, while in our terminology, the paths connect the program statements (i.e., transitions). The translation between the two levels will be discussed in Section 3.

3. Solution

In this section, we propose our solution for the P4 static cost analysis problem described in Section 2, and more specifically for calculating the estimator $\vec{\theta}_X$. In the first part, we go over the data requirements and software architecture of the tool, and then we focus on our approach to difficult computational questions inherent in calculating $\vec{\theta}_X$, such as how to account for memory state changes occurring during program execution.

3.1. System Description

Figure 1 models the data requirements and architecture of the software we developed for estimating the program execution time (i.e., for calculating the estimator $\vec{\theta}_X$). In Section 2, we gave a theoretical view on what kind of data $\vec{\theta}_X$ could rely on, and now we refine this view into a more practical one.

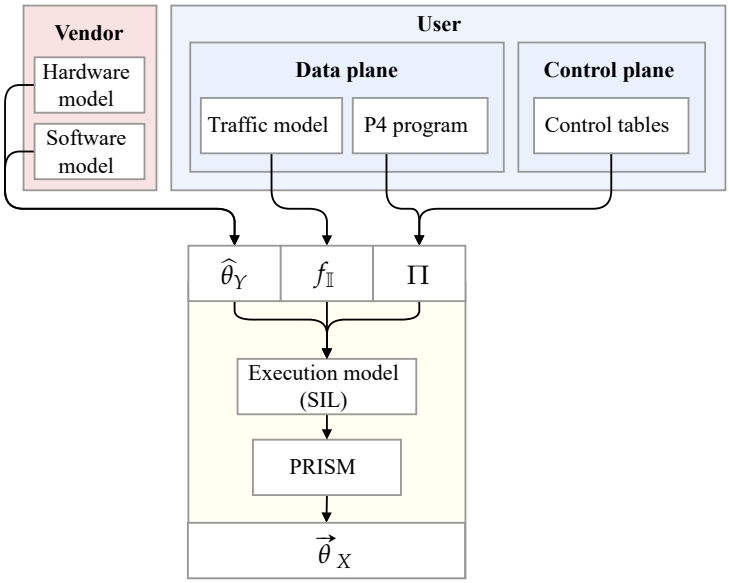


Figure 1. Overview of the cost analysis funnel.

In our intended usage scenario, there are two important stakeholders: (1) vendors, who manufacture hardware or software switches and possess in-depth (possibly secret or proprietary) knowledge about their product and (2) users who intend to run their P4 programs on the software switch in order to transmit packets and who would appreciate knowing $\vec{\theta}_X$ before they invest into buying a specific switch, building a specific network, etc. Unfortunately, none of the stakeholders have all the information needed to calculate $\vec{\theta}_X$; the vendors have no knowledge of the intended use case of their customers, while the customers lack (and possibly are prohibited from acquiring) in-depth knowledge about the internals of the product they use for running their use cases. As such, the cost analyzer needs input from two sources. Vendors—having insider knowledge about and an adequate testbed for the hardware or software they sell—know most about the program execution environment ($\mathbb{E}$), and so they can successfully benchmark the program-independent, instruction-level cost model ($\widehat{\theta}_Y$) and ship it together with their product. The concrete instruction set (Inst) in question is codified by the cost analyzer API. Users own the Π program code to be executed (composed of the P4 program code and the lookup table contents), and they usually know enough about the expected network traffic to model it probabilistically as $f_{\mathbb{I}}$. With this, all the necessary information ($f_{\mathbb{I}}$, Π, and $\widehat{\theta}_Y$) is available for the cost analyzer to calculate the estimator $\vec{\theta}_X$. In a business setting, either the vendor runs a cost analyzer instance preloaded with $\widehat{\theta}_Y$ and exposes this to potential users (e.g., through a web interface) who can upload there data ($f_{\mathbb{I}}$, Π) or the users run their own cost analyzer instances and somehow acquire the missing $\widehat{\theta}_Y$ information from the seller of the prospective product. Either way, the cost analyzer needs a robust frontend to merge all this information together so that the tool can process it. Internally, we deliver the "heavy lifting" part of cost analysis to the PRISM probabilistic model checker [9], a tool specifically designed to deal with the computational complexity inherent in static cost analysis. As translating the input information to the computational model used in PRISM is non-trivial, we dedicate the rest of this section to detailing this aspect of our solution.

3.2. Implementing a Sequential P4 Interpreter in PRISM

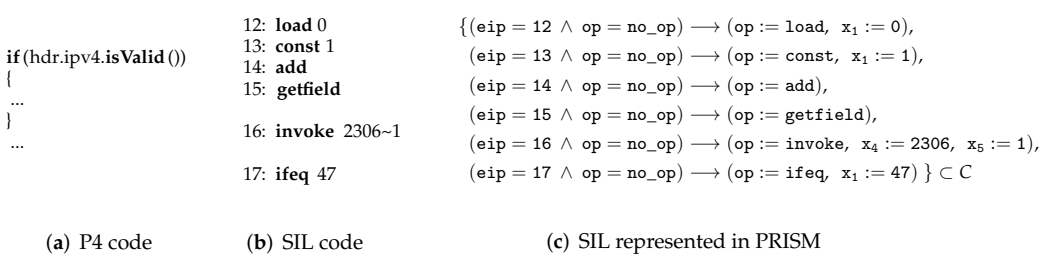
While P4 programs tend to spend most of their time performing table lookups, cost analysis requires a complete understanding of the semantics of P4, including the control flow and side effects. For example, a program may conditionally execute an expensive table or not based on the outcome of an earlier table lookup (see Listing 1 in Section 4). To successfully estimate the execution time, we need to know whether this expensive table was executed or not, which in turn requires us to know the outcome of the earlier lookup and so on going back to the very start of the program. (We also need to know how table lookups are performed. We dealt with this topic in [12], where we presented a probabilistic (Markov) model of the DIR-24-8 algorithm.) For this reason, we compile P4 down to the PRISM probabilistic model checker which, given the appropriate operational semantics of P4, performs probabilistic cost analysis out of the box, taking into account the side effects and program control flow as well.

The scope of PRISM is more general than programming languages, as its programs have to be described using a simple, state-based language, which can easily be translated into Markov models. In this section, we show our idea for compiling P4 code (or, in fact, code written in any other C-like language) to the PRISM modeling language.

The subset of PRISM we target can be represented as a pair (V, C), where V is a set of variable declarations in the form of $(\texttt{name}, \texttt{type}, \texttt{initial\_value})$ triplets and C is a set of guarded commands in the form of $\texttt{probability}, \texttt{guard} \longrightarrow \texttt{effect}$ rules. The rules declare that from an s state satisfying the `guard`, there is a state transition leading to the state produced by applying the `effect` to s. PRISM allows both non-deterministic and probabilistic evaluation. In case multiple rules are matching, one is chosen randomly using the probability distribution specified by the `probability` attribute of the respective rules. Naturally, `probability` attributes in rules with identical `guard` expressions should add up to one. Guards are simple boolean expressions over constants and variables in V. Effects are lists of assignments where the left-hand side is a variable in V and the right-hand side is an expression over constants and the variables in V. We will omit `probability` from the notation in case it is one.

Remark 1. *In this work, we will not make use of PRISM's more sophisticated features such as concurrent processes and synchronization, but these are interesting for future work with respect to parallel network switch architectures.*

Figure 2 illustrates how we translate high-level P4 control flow to low-level PRISM commands. Our central idea was to implement a stack-based instruction language (SIL) interpreter in PRISM. An SIL supports call-by-reference function calls as well. We designed the SIL to resemble JVM bytecode for two reasons: (1) to aid the ease of understanding, since many programmers have at least some superficial familiarity with JVM bytecode, and (2) because JVM bytecode is a tried-and-true target for many popular programming languages, so we did not have to face unforeseen challenges when we compiled P4. We emphasize that JVM bytecode is currently just an inspiration of the SIL, and faithful implementation of the JVM bytecode is not among the immediate goals of this work.



(**a**) P4 code (**b**) SIL code (**c**) SIL represented in PRISM

Figure 2. Translating P4 code to bytecode in PRISM.

One downside of this design is that the SIL interpreter may be a very different runtime environment to the real-world environments (components of $\mathbb{E}$) in which P4 programs are executed. (For example, the P4 specification expects copy-in/copy-out call semantics [14], but there are implementations, such as T4P4S, deviating from this for optimization reasons, and SIL deviates as well) This is counterweighted by two factors: (1) as we will see, cost models ($\widehat{\theta}_Y$) have a free hand in deciding the cost of instructions and may use this to downplay (or overplay) elements in the SIL that are not relevant (or highly relevant) to the target architecture in question, and (2) in real-world usage, P4 programs spend most of their time in lookup tables anyway, so we expect that even large differences in the control flow representation will introduce only minor errors in the estimates produced by $\vec{\theta}_X$.

Figure 2a depicts a simple conditional structure in P4, where the condition is a call to the `isValid` function in data field `hdr.ipv4`, which returns whether the validity bit of `hdr.ipv4` is set (which usually indicates that the data structure stores a successfully parsed packet header). In Figure 2b, we load the address of `hdr` (supposedly stored in parameter 0) to the stack and then increment the address by one to obtain the address of `hdr.ipv4`. (We store the struct size in the first cell at struct addresses followed by the field contents, of which the first in the case of `hdr` is that of `ipv4`.) We then obtain the reference stored at `hdr.ipv4` (which supposedly points to the start of the actual IPv4 header) and pass this reference to the `isValid` function (supposedly in line 2306) that we call. In case `isValid` comes back as false, we jump to the else branch or the merge point of the conditional (presumably starting at line 47); otherwise, we continue into the true branch. Figure 2c depicts how the SIL actually looks when represented in PRISM. Each command is only triggered when the program counter (`eip` : $\mathbb{N}$ variable) stores its line number, and no other instructions are currently in progress (stored in the `op` : $\mathbb{N}$ variable, though note that PRISM has no strings, so we had to represent instructions as named integer constants). In case a command is triggered, we set `op` to the name (numeric identifier) of the current instruction and store its argument in one of the registers (represented by $x_i : \mathbb{N}$ variables). The *C* set of PRISM commands must also contain the implementation of the SIL interpreter (not depicted). On the one hand, these program-independent commands have to perform housekeeping (implement heap and stack, increment the program counter after instructions, etc.), and on the other hand, they have to implement the instruction set of the SIL.

We will not detail too deeply how we implemented the interpreter in PRISM, but we present the most important ideas. The state of the interpreter consists of the following variables (and possibly more): (`eip` : $\mathbb{N}$, `esp` : $\mathbb{N}$, `ebp` : $\mathbb{N}$, `op` : Inst, `nop` : Inst, `error` : Errors, $(\mathtt{x})_k : \mathbb{N}^k$, $(\mathtt{z})_n : \mathbb{N}^n$, $(\mathtt{s})_m : \mathbb{N}^m$).

Register `eip` stores the instruction pointer (a label), `esp` stored the stack pointer, and `ebp` stores the base pointer. Register `op` stores the identifier of the current instruction. Some of these are meant to be internal (e.g., `iread`, `iwrite`, and `ipush`), while others, such as `const`, `add`, or `invoke`, are public. Internal instructions are used in the implementation of public instructions. `op` = `end_op` means that the last instruction was finished and `eip` can be incremented. `op` = `no_op` represents an idle state (i.e., that a new instruction (at the current `eip`) can be started). `nop` stands for the next operation being used to chain multiple steps of instructions internally. In case an illegal state is reached, `op` is set to `error_op` which is used if the machine gets stuck, and an error code is stored into the register `error`.

$\mathtt{x}_0$, $\mathtt{x}_1$, etc. and $\mathtt{z}_0$, $\mathtt{z}_1$, etc. are for passing arguments to instructions, as in the case of `const1`. Finally, a very long $\mathtt{s}_0$, $\mathtt{s}_1$, etc. sequence stores the contents of the stack. In the case of P4, the first segment of the stack is reserved for representing the program memory. PRISM has no concept of random access arrays nor of other collections. This means we have to generate $(\mathtt{s})_m$ automatically as a sequence of individual registers. Moreover, we need to generate access (read and write) rules for each register as well. The `iread` and `iwrite` rules generated for accessing address 23 on the stack are illustrated by Equation (3). PRISM will apply the first rule when the current operation is `iread` and its arguments measure is 23, and it "returns" the value at position 23 (i.e., the content of register s_{23})

by storing it in the internal register. The second rule with `iwrite` is similar to `iread`, except in this case, there are two internal arguments excepted: the first one is a value to be written to a register, and the second one stores the address to be written. The third rule illustrates handling illegal arguments, which in this case is an address pointing out of the stack. Finally, the fourth rule with `ipush` is another internal instruction illustrating how one generated `iwrite` can be used. It "passes" z_1 to `iwrite` and loads the address of the top of the stack into z_2, which will cause the value in z_1 to be written to the top of the stack (unless the stack is full), leading to an error:

$$
\begin{aligned}
&C_{\texttt{internal}} = \\
&\quad \{(\texttt{op} = \texttt{iread} \ \wedge \ z_1 = 23) \longrightarrow \\
&\qquad\qquad (\texttt{z}_0 := s_{23}, \ \texttt{op} := \texttt{nop}, \ \texttt{nop} := \texttt{no\_op}), \\
&\quad (\texttt{op} = \texttt{iwrite} \ \wedge \ z_2 = 23) \longrightarrow \\
&\qquad\qquad (\texttt{s}_{23} := z_1, \ \texttt{op} := \texttt{nop}, \ \texttt{nop} := \texttt{no\_op}), \\
&\quad (\texttt{op} = \texttt{write} \ \wedge \ z_2 > m) \longrightarrow \\
&\qquad\qquad (\texttt{op} := \texttt{error\_op}, \texttt{error} := \texttt{access\_violation}), \\
&\quad (\texttt{op} = \texttt{ipush}) \longrightarrow \\
&\qquad\qquad (\texttt{op} := \texttt{iwrite}, \ \texttt{z}_2 := \texttt{esp} + 1, \ \texttt{esp} := \texttt{esp} + 1, \ \texttt{nop} := \texttt{nop}), \\
&\quad \ldots \\
&\quad \}
\end{aligned}
\tag{3}
$$

As emphasized in Section 2, one of the most important obstacles to static cost analysis is handling unknowns during the execution (for example, not knowing what kind of input packet the program will operate on), and our approach to handling this issue is to model these unknown factors probabilistically. In terms of model checking, this means that program execution can branch either probabilistically or non-deterministically (in case the branching probability is unknown). For example, in the example in Section 4, it is unknown which packets will be processed by the switch, so there are multiple possible program executions ($|\Pi| \geq 1$). Yet, in this example, we do know the $f_{\mathbb{I}}$ probability distribution of possible packets, so it is possible to calculate the f_{Π} probability distribution of possible program executions (see Equation (1)).

In our implementation, this calculation is performed automatically by PRISM. We just need to tell PRISM where and how it should branch executions. Equation (4) illustrates a probabilistic PRISM command in the SIL that we use for this purpose. It states that the `goto` instruction in line 661 should step forward to line 662 with a probability of 0.33, jump to line 1055 with a probability of 0.34, and jump to line 1448 with a probability of 0.33. In particular, we use this command in the case study in Section 4 to simulate receiving one of three packets at the beginning of the P4 program's execution. Line 662 is supposed to assign one specific packet to the input buffer, line 1055 assigns a different one, and line 1448 assigns yet another:

$$
\begin{aligned}
&\{(0.33, \ \texttt{eip} = 661 \ \wedge \ \texttt{op} = \texttt{no\_op}) \longrightarrow (\texttt{op} := \texttt{goto}, \ \texttt{x}_1 := 662), \\
&\ (0.34, \ \texttt{eip} = 661 \ \wedge \ \texttt{op} = \texttt{no\_op}) \longrightarrow (\texttt{op} := \texttt{goto}, \ \texttt{x}_1 := 1055), \\
&\ (0.33, \ \texttt{eip} = 661 \ \wedge \ \texttt{op} = \texttt{no\_op}) \longrightarrow (\texttt{op} := \texttt{goto}, \ \texttt{x}_1 := 1448) \ \} \subset C
\end{aligned}
\tag{4}
$$

Note that in Section 2, we limited the probabilistically modeled phenomena to input packets ($\mathbb{I}$), while we used statistics ($\widehat{\theta}_Y$) to address other environment-related unknowns ($\mathbb{E}$). Yet, with the above mechanism, we could easily expand the probabilistically modeled factors (and shrink $\mathbb{E}$ at the same time) as well, which allows ample room for future work (for example, to include the probabilistic model of DIR-24-8 in [12]).

3.3. Static Cost Analysis for P4 in PRISM

As we specified earlier in Table 2, our approach to estimating the execution time (that is, to calculating $\vec{\theta}_X$) relies on the f_Π probability distributions of possible executions and also on a cost model $\widehat{\theta}_Y$ that assigns costs to elementary instructions (based on the statistics of isolated measurements taken for the target architecture).

In Section 2.4, we discussed how PRISM, as a probabilistic model checker, augments traditional property checking with the checking of probabilistic properties (that is, calculating the probability of whether a property is satisfied by the program) and also with the checking of reward-based properties (accumulating expected rewards or costs over the set of possible executions). For static cost analysis, we can rely on this latter facility.

In PRISM, a reward (or cost, as this is only a moral distinction) can be specified as an $r : \mathbb{G} \to \mathbb{N}$ reward function, where $\mathbb{G}$ denotes the set of logical propositions over the variables of the state space. In the case of a definition $r(\texttt{guard}) = \texttt{score}$), whenever execution reaches a state that satisfies `guard` (a predicate over the program variables), the reward is incremented by `score` (a number).

We implemented cost models ($\widehat{\theta}_Y$) as specific r reward functions. For example, in the case study in Section 4, we used the reward definition in Equation (5) to represent a cost model $\widehat{\theta}_Y^{\texttt{min}}$, which assigned a 0.114-ms overhead cost to the start of the program and assigned an estimated cost formula (see Equation (7) in Section 4) to the function invocation (supposedly a table lookup invocation, treated here as an elementary instruction) triggered at line 45:

$$\begin{aligned} r_{\texttt{best\_case}}(\texttt{eip} = 0 \ \wedge \ \texttt{op} = \texttt{no\_op}) &= 0.114 \\ r_{\texttt{best\_case}}(\texttt{eip} = 45 \ \wedge \ \texttt{op} = \texttt{invoke}) &= 20000 \quad \cdot \frac{2.031 - 0.114}{100000} \end{aligned} \tag{5}$$

For static cost analysis, we need PRISM to check or solve the reward-based properties in the form of PRCTL predicates (see Section 2.4). In addition to $\mathbf{E}_K(\Phi)$ (checking whether the expected reward is within interval K) and $\mathbf{E}_\Delta(\Phi)$ (solving the formula for variable Δ (i.e., returning the expected reward)), PRISM can also calculate and check the maximum and minimum rewards. We will denote the maximum reward operator as $\mathbf{M} \uparrow_K(\Phi)$ (and $\mathbf{M} \uparrow_\Delta(\Phi)$) and the minimum reward operator as $\mathbf{M} \downarrow_K(\Phi)$ (and $\mathbf{M} \downarrow_\Delta(\Phi)$). In case the name of the reward function is relevant, we write it in superscript.

Specifically, to calculate the minimum value of cost $r_{\texttt{best\_case}}$ (i.e., to calculate the estimator $\vec{\theta}_X^{\texttt{min}}$ in Table 2), we need PRISM to solve the property in Equation (6) for the variable Δ:

$$\mathbf{M} \downarrow_\Delta^{\texttt{best\_case}}(\texttt{op} = \texttt{done}) \tag{6}$$

The PRCTL formula accumulates the rewards on the *minimal path fragments* that satisfy $\Diamond(\texttt{op} = \texttt{done})$ (i.e., on the shortest path prefixes on which the program state, designated as the final state, is eventually reached (the SIL register `op` eventually stores the special `done` instruction)). For each of these paths, PRISM will keep a separate account for reward $r_{\texttt{best\_case}}$ and increment it in each program state as per the rules defined for $r_{\texttt{best\_case}}$. Finally, PRISM returns the minimum of these reward instances.

Remark 2. *In the case of bound checking (with $\mathbf{E}_K(\Phi)$, $\mathbf{M} \uparrow_K(\Phi)$, $\mathbf{M} \downarrow_K(\Phi)$, and even with $\mathbf{P}_J(\varphi)$), we expect users to specify their requirements manually. The exact requirements depend entirely on the application. For example, a network engineer replacing a hardware switch with a software switch will be possibly interested in checking the latency requirements of 10 Gigabit Ethernet networks. One the other hand, a developer testing a P4 compiler will be more interested in benchmarks, such as those in [3]. Alternatively, requirements can also be generated automatically. In [6], Gao et al. generated PRCTL properties with exact bounds by finding suitable values for Δ and then gave the user the option to relax these bounds if needed. Recently, in [15], the authors also experimented with automatically generating complete CTL formulas. Such an approach is viable in cases where manual requirement design cannot provide full coverage for a large system, as in the*

case of modern enterprise networks. Exploration of possible application scenarios with requirements where the tool can be applied is an interesting and viable topic for future work.

4. Case Study

In this section, we introduce a case study to illustrate the usage of our cost analysis tool. We will use the notations introduced in Section 2. First, we informally describe our goal, namely to validate our static estimates ($\vec{\theta}_X$) against actual measurements $\widehat{\theta}_X$. This section is followed by the description of the software and hardware under testing. After that, we describe the measurement environment and preparation of the data ($\widehat{\theta}_Y$) needed for initializing the cost analysis tool. Finally, we perform validation and evaluate its results.

4.1. Objectives

Our goal is to illustrate the cost analysis process from beginning to end through a simple case study. In Table 2 of Section 2, we introduced three statistics for estimating the minimum, average, and maximum performance. For each $(\widehat{\theta}_X, \vec{\theta}_X)$ pair in the table, we conduct measurements to find $\widehat{\theta}_X$, use the cost analyzer to estimate $\vec{\theta}_X$, and then compare the error between the two.

As we are currently restricted to simple models of complex processes (such as the interoperation of lookup tables and hardware caches), our primary goal is not to show how accurate the static cost analysis is but rather to show how it composes data from various sources into cohesive performance estimates. Nonetheless, we will also attempt to evaluate the estimation and highlight its weaknesses with the goal to facilitate future work.

Additional Estimators

In Section 2, we noted that we could construct nine possible estimators altogether by combining the minimum, average, and maximum statistics of the respective components of the three estimators in Table 2. In Table 3, we introduce two more of these estimators, because we found them meaningful and relatively easy to infer using static cost analysis. By simply taking the minimum (maximum) of the whole pool (as in the case of $\theta_X^{\texttt{min}}$ and $\theta_X^{\texttt{max}}$), we failed to capture the distinction between having good (bad) performance due to fortunate (unfortunate) inputs from $\mathbb{I}$ versus having good (bad) performance due to fortunate (unfortunate) environments from $\mathbb{E}$. With $\theta_X^{\texttt{avg\_best}}$, we intend to represent a population parameter that tells us how the program performs on average across multiple inputs (i.e., across a packet stream) in case we find the most fortunate environments. We denoted these beneficial outcomes with $\Omega^{\texttt{best}}$. Note that $\Omega^{\texttt{best}} \subseteq \Omega$. The other parameter, $\theta_X^{\texttt{avg\_worst}}$, has the same purpose, except it tells us about the average performance in unfortunate environments ($\Omega^{\texttt{worst}}$).

Table 3. An extension of Table 2 with two more estimators.

Parameter (θ_X)	Dynamic Cost Analysis ($\widehat{\theta}_X$)	Static Cost Analysis ($\vec{\theta}_X$)
$\theta_X^{\texttt{avg\_best}}(\Omega)$ $= \sum_{\omega \in \Omega^{\texttt{best}}} (f(\omega)X(\omega))$	$\widehat{\theta}_X^{\texttt{avg\_best}}(\underline{\boldsymbol{\omega}})$ $= \sum_{\pi \in \Pi} \frac{\min(S_X(\underline{\boldsymbol{\omega}}\|_\pi))}{\lvert S_X(\underline{\boldsymbol{\omega}}\|_\pi)\rvert}$	$\vec{\theta}_X^{\texttt{avg\_best}}(\underline{\boldsymbol{e}})$ $= \sum_{\pi \in \Pi} (f_\Pi(\pi)\ \widehat{\theta}_Z^{\texttt{min}}(\pi, \underline{\boldsymbol{e}}))$
$\theta_X^{\texttt{avg\_worst}}(\Omega)$ $= \sum_{\omega \in \Omega^{\texttt{worst}}} (f(\omega)X(\omega))$	$\widehat{\theta}_X^{\texttt{avg\_worst}}(\underline{\boldsymbol{\omega}})$ $= \sum_{\pi \in \Pi} \frac{\max(S_X(\underline{\boldsymbol{\omega}}\|_\pi))}{\lvert S_X(\underline{\boldsymbol{\omega}}\|_\pi)\rvert}$	$\vec{\theta}_X^{\texttt{avg\_worst}}(\underline{\boldsymbol{e}})$ $= \sum_{\pi \in \Pi} (f_\Pi(\pi)\ \widehat{\theta}_Z^{\texttt{max}}(\pi, \underline{\boldsymbol{e}}))$

In static cost analysis, we estimated the $\theta_X^{\texttt{avg\_best}}$ parameter with $\vec{\theta}_X^{\texttt{avg\_best}}$. Like $\vec{\theta}_X^{\texttt{avg}}$ did, this estimator calculates the expected performance across all paths, except that $\theta_X^{\texttt{avg\_best}}$ uses the cost model $\widehat{\theta}_Z^{\texttt{min}}$, which estimates the performance of a path in the best possible environment. $\theta_X^{\texttt{avg\_worst}}$ is analogous. To validate $\vec{\theta}_X^{\texttt{avg\_best}}$, we used the estimator $\widehat{\theta}_X^{\texttt{avg\_best}}$. Here, we partitioned the $\underline{\boldsymbol{\omega}}$ outcomes based on which $\pi \in \Pi$ path was taken (determined solely by the input and the program code). $S_X(\underline{\boldsymbol{\omega}}\|_\pi)$ denotes samples taken

in an environment where the input triggered the path π. To estimate the performance of a path in the best possible environment, we took the minimum of $S_X(\underline{\omega}|_\pi)$. We then estimated the performance across inputs (paths). We calculated the mean of the minimums, weighted by the size of each $S_X(\underline{\omega}|_\pi)$ sample (which is again, determined solely by the input and the program code).

4.2. Use Case

We will now enumerate the software and hardware components of the switch, whose performance we will estimate both statically and dynamically. In particular, we will examine the performance of the P4C Behavioral Model (BM) software switch [16] as it executes a P4 program that we selected from the test cases of the P4C compiler.

4.2.1. P4 Program

The particular P4 program whose performance we attempted to estimate is called `basic_routing-bmv.p4`, a minimal L2 or L3 pipeline freely available from the P4C test cases [17] with slight modifications (e.g., we configured each table to use ternary lookup for the reasons we detailed earlier). We chose this program as it is publicly available, relatively easy to read, and can be analyzed in a feasible time due to its size. The code excerpt most relevant to this study is depicted in Listing 1, together with its abstract control flow graph in Figure 3. When the `ingress` block of this P4 pipeline is reached, the program checks whether there is a valid IPv4 packet stored in the `hdr.ipv4` data structure, and it proceeds to perform the table lookup on tables `port_mapping` and `bd`. Then, another table lookup is performed on table `ipv4_fib`. Only if this lookup fails will we perform `ipv4_fib_lpm` as well; otherwise, this is skipped. In the final line, we set the `egress_spec` field to `1` to forward the packet to port 1 (where we listen for its arrival). As noted in Section 2, we will assume the contents of the lookup tables are hardwired and therefore not changing (we can handle different table contents by recalculating the estimators).

Remark 3. *In P4, table lookups can have side effects. For example, a successful match in table* `port_mapping` *may alter the current packet in a way that affects whether* `ipv4_fib` *succeeds, in turn deciding whether the potentially time-intensive* `ipv4_fib_lpm` *will be executed. In Section 3, handling this problem was one of the reasons we implemented the complete P4 language semantics in PRISM.*

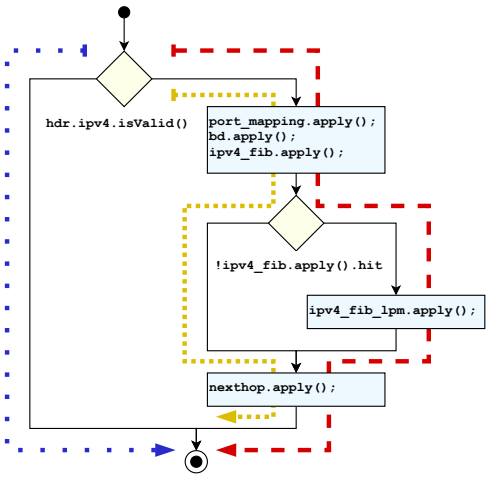


Figure 3. High-level control flow of Listing 1.

Listing 1. Modified P4 source code from [17].

```
control ingress(inout headers hdr, [...]) {
  table bd {[...]}
  table ipv4_fib {[...]}
  table ipv4_fib_lpm {[...]}
  table nexthop {[...]}
  table port_mapping {[...]}
  [...]
  apply {
    if (hdr.ipv4.isValid()) {
      port_mapping.apply();
      bd.apply();
      if (!ipv4_fib.apply().hit) {
        ipv4_fib_lpm.apply();
      }
      nexthop.apply();
    }
    standard_metadata.egress_spec = 1;
  }
}
```

Remark 4. *Figure 3 depicts the three execution paths of the* `ingress` *block (*$|\Pi| = 3$*). We should note that, depending on the incoming traffic and table contents, there are possibly many more execution paths. A successful match will cause an early exit from the lookup and invoke an action, and these also impact performance. To model these paths, we should model lookup algorithms in PRISM as described in Section 3. While we omitted this step from our current work (and in this experiment, we use tables that never match), in [12], we already modeled the DIR-24-8 lookup algorithm probabilistically in the form of a relatively small Markov chain. This can be directly implemented in PRISM as well.*

In Figure 3, the leftmost path (thinly dotted with a blue arrow) is the least expensive (no tables are executed), the rightmost path (dashed with a red arrow) is the most expensive (all tables are executed), and the cost of the middle path is between these two (some tables are executed).

4.2.2. Program Data

We filled the tables of `basic_routing-bmv.p4` with entries up to the given count in Table 4. We intentionally inserted entries that never matched any packets in $\mathbb{I}$ so that a linear lookup (e.g., the ternary lookup in BM) had to go over the complete tables. The only exception was `ipv4_fib`, as this table had only one entry that would match (leading to the execution of `ipv4_fib_lpm`), depending on the packets. The role of this one entry table is to make it easy to check that the PRISM implementation indeed performs the matching.

As our goal was to validate our estimation of θ_X, we would send in possibly the simplest packet stream that triggered each path with an equal likelihood. As described in Table 4, our input space $\mathbb{I}$ will consist of three packets: p_1 triggers the leftmost path (π_1), p_2 triggers the middle one (π_2), and p_3 triggers the rightmost one (π_3). To make a sufficiently expressive plot of our S_X sample as we measured $\widehat{\theta}_X$, the first $\frac{1}{3}$ of our packet stream consisted of p_1 packets, the second $\frac{1}{3}$ consisted of p_2 packets, and the final $\frac{1}{3}$ consisted of p_3 packets. (Note that switches can take advantage of caching when they receive the same packet many times.) Our sample size was $|S_X| = 5000$, and we specified a delay of 50 ms between packets in order to avoid buffering (although predicting anomalies introduced by packet queues is an interesting avenue for future research).

Table 4. $(\Pi, \mathbb{I})$ configuration used for validation $(\widehat{\theta}_X)$.

Π	**Program**	`basic_routing-bmv2.p4`			
	Tables	**Name**	**Entries**	**Keys (bits)**	**Matches**
		`port_mapping`	10,000	9	∅
		`bd`	20,000	16	∅
		`ipv4_fib`	1	44	p_3
		`ipv4_fib_lpm`	50,000	44	∅
		`nexthop`	20,000	16	∅
	Total number of paths ($\lvert\Pi\rvert$)	3			
$\mathbb{I}$	**Packets**	**Name**	**Desc**		
		p_1	Packet failing `hdr.ipv4.isValid`		
		p_2	Packet failing `!ipv4_fib.apply().hit`		
		p_3	Packet satisfying `!ipv4_fib.apply().hit`		
	Sample size ($\lvert S_X \rvert$)	5000 pcs with 50 ms delay + warm-up session			
	Packet distributions ($f_{\mathbb{I}}$)	**Sample**	p_1	p_2	p_3
		S_X	0.33	0.33	0.34

4.2.3. Set-Up

In this case study, we conducted our measurements according to the set-up described in Figure 4, with the HW and SW specifications of the target computer listed in Table 5. There are two notable features in this table. One feature worth mentioning is the caches, as Core i7 processors have inclusive L3 and non-inclusive L1 and L2 cache policies [18]. This means that in the case of an L1 or L2 cache miss, the CPU can query the L3 cache instead of waiting until an L1 or L2 reload (i.e., it only suffers a bandwidth penalty and not a cache miss penalty (unless an L3 cache miss happens as well)). The other feature in the table is that we used the P4C Behavioral Model (BM) software switch [16] for running our P4 programs. This programmable switch was designed as a reference model with respect to the expected behavior of P4 switches, but its performance was expected to be worse compared with the highly optimized, production-grade switches (such as Intel Tofino or the DPDK-based T4P4S). For our current validation purposes this is better, since we can expect the BM's performance to be easier to observe and consider. Additionally, the BM version specified here uses linear searching for ternary lookups. While this is much less efficient than the conventionally used sublinear lookup algorithms (such as cuckoo hashing, DIR-24-8, or prefix trees), it again helps us in validating the cost analysis by making it easier to observe and consider the performance. As of this writing, the BM pre-allocates entries using a C++ `std::vector`, presumably to ensure contiguous storage for cache efficiency reasons.

Table 5. Specifications of hardware and software used in the case study.

$\mathbb{E}$	**Machine specs**	Intel Core i7-7500U @ 4x3.5GHz
		L1: 32KB, L2: 256KB, L3: 4MB, DDR4 RAM: 4GB
		Ubuntu 20.04, tcpdump 4.9.3
	Software switch	P4C-BM 1.14.0

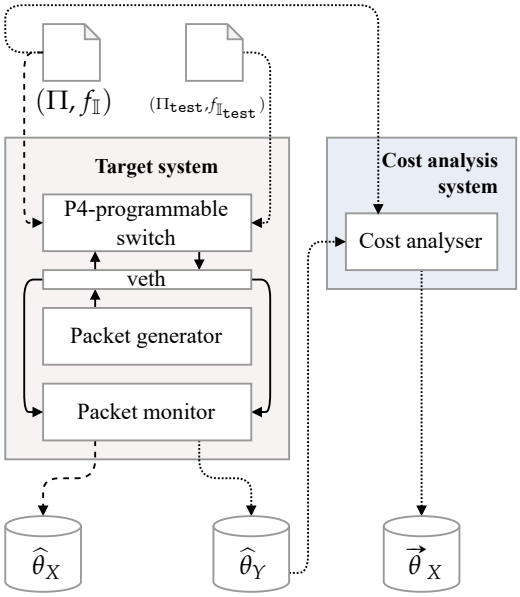


Figure 4. System set-up for validating estimates resulting from static cost analysis.

4.3. Data Collection

Our cost analysis tool estimates the latency (θ_X) of programmable switches. Given a target switch and a P4 program, the cost analysis tool estimates ($\vec{\theta}_X$) how long it will take the switch to execute the P4 program for one input packet. To validate the cost analysis tool, we must compare these $\vec{\theta}_X$ estimates with the actual measurements taken on the target switch ($\widehat{\theta}_X$).

We see the validation process as a sequence of experiments. With each validation experiment, we associate a specific packet stream and a P4 program. On the one hand, we install the P4 program on the target switch, send the packet stream, and measure the latency step by step. On the other hand, we parameterize the cost analysis tool with the packet stream and the P4 program and start the analysis tool. In the final step, we compare the measurements (generated by the former) and the static estimates (generated by the latter).

The system we designed for conducting the experiments is depicted in Figure 4. In the Figure, we denote the components related to the cost analysis with a dotted arrow, those related to benchmarking and validation with a dashed arrow, and those that are participating in both with continuous arrows.

On the left side, the target computer runs a virtual switch, which in turn runs a P4 program (Π or $\Pi_{\texttt{test}}$). The host environment sends packets on an input virtual network interface (according to $f_{\mathbb{I}}$ or $f_{\mathbb{I}_{\texttt{test}}}$), where they are read and processed by the virtual switch. In turn, the virtual switch copies the processed output packet onto the output virtual network interface, from which the host environment reads the packet and duly records the time the switch took to process the packet (i.e., the difference between the timestamp of the packet appearing on the input interface and the timestamp of it appearing on the output interface). The set-up assumes that no packets are dropped, but it can be modified to detect packet drops as well. On this system, we conducted both the bootstrapping measurements (resulting in $\widehat{\theta}_Y$, the cost model we used during cost analysis to calculate $\vec{\theta}_X$) and—in a separate phase—the actual benchmarks ($\widehat{\theta}_X$), which we compared with $\vec{\theta}_X$.

On the right side, a separate workstation runs our static cost analysis tool. The P4 program is input into the tool, and the output is $\vec{\theta}_X$, the cost estimate. Aside from the program code, the tool also needs a cost model ($\widehat{\theta}_Y$) as input. These data contain quantitative information that models the deep characteristics (such as lookup algorithms and machine performance) of the target computer. To assemble $\widehat{\theta}_Y$, we ran special P4 programs on the target computer in order to measure the execution time of the primitive language elements. In real world usage, we envision that the $\widehat{\theta}_Y$ cost model would be produced by the target

switch manufacturer or vendor so users can make $\vec{\theta}_X$ estimates without having to acquire the hardware.

Data Collection for Cost Models ($\widehat{\theta}_Y$)

To predict the performance, we will calculate $\vec{\theta}_X$ with PRISM for the $(\Pi, \mathbb{I})$ configuration described in Table 4. However, for this we need $\widehat{\theta}_Y$ (i.e., to estimate the costs of the primitive program components (Inst $= \{i_1, \ldots, i_n\}$)) independent from specific P4 programs (sets of $(i_{k_1}, \ldots, i_{k_m})$ program paths) in order to compose these costs during the estimation of $\vec{\theta}_X$ for specific Π P4 programs. In this case study, we constructed $\widehat{\theta}_Y$ to model the cost of executing lookup tables of different sizes. We chose to use this model for its simplicity, but the PRISM representation allows more intricate models (with better predictive power) as well, including Markov chains.

The $(\Pi_{\texttt{test}}, \mathbb{I}_{\texttt{test}})$ configuration we used for this is described in Table 6. We used a simple P4 program (not depicted) called `table-benchmark.p4` that performed a lookup in the `test_table` if it received packet p_2 while it avoided the lookup if it received packet p_1. `test_table` had 100,000 entries. We chose this number simply because it was the sum of the entry counts in Table 4. The size of the individual entries to be matched was 48 bits (the size of an Ethernet destination address), comparable to the key size of `ipv4_fib_lpm`. We had two test cases, with each used to estimate the cost of specific primitive program components. (Note that the cost analyzer can accept almost arbitrary components, as per Section 3. The components we identify here are just for illustration.)

Table 6. $(\Pi_{\texttt{test}}, \mathbb{I}_{\texttt{test}})$ configuration used for cost modeling ($\widehat{\theta}_Y$).

$\Pi_{\texttt{test}}$	**Program**	`table-benchmark.p4`			
	Tables	**Name**	**Entries**	**Keys (bits)**	**Matches**
		`test_table`	100,000	48	∅
	Total number of paths ($\lvert\Pi_{\texttt{test}}\rvert$)	2			
$\mathbb{I}_{\texttt{test}}$	**Packets**	**Name**	**Desc**		
		p_1	No table lookup performed		
		p_2	Table lookup performed		
	Sample size ($\lvert S_Y^1\rvert = \lvert S_Y^0\rvert$)	5000 pcs with 50 ms delay + warm-up session			
	Packet distributions ($f_{\mathbb{I}_{\texttt{test}}}$)	**Sample**	p_1	p_2	
		S_Y^1	0	1	
		S_Y^0	1	0	

We used two samples to derive $\widehat{\theta}_Y$. We obtained sample S_Y^1 by sending in 5000 packets of the type p_2, and in each instance, an unsuccessful lookup was performed on `test_table`. As the whole table was read through, we could use these measurements to estimate the cost of performing a lookup on tables of other sizes as well. Unfortunately, the pipeline has steps other than the table lookup (for example, packet parsing), and we should avoid counting the cost of these for the table lookup costs. For this reason, we collected another sample (S_Y^0) by sending in p_1-type packets, triggering the path where only the bare minimum amount of work was performed (i.e., receiving and parsing a packet and forwarding it to the egress port). As such, S_Y^0 measures the baseline cost of running a P4 program. We could correct the statistics derived from S_Y^1 by subtracting S_Y^0.

Figure 5 depicts our measurements of S_Y^1 and S_Y^0 based on the configuration in Table 6. To produce a precise cost model $\widehat{\theta}_Y$ for estimation purposes, we need to take into account the peculiarities of the environment as much as possible. When we conducted the first iteration of the validation and compared S_Y^1 in Figure 5 to the third portion of S_X in Figure 6 (see Section 4.4), we noticed that the former had a much higher average, even though the

total number of entries visited during the table lookups was identical (100,000). We suspect this is because the entry count in itself does not tell us about the actual size of the data moved during the lookup; we need to consider the size of each entry (notably, the key sizes) as well. Tables with large keys (such as `test_table`) will have fewer entries per KB, which means they need more time to move the same number of entries compared with tables with small keys. With Equation (7), we estimated the actual amount of data moved per table lookup as entry count times entry size, where the latter is the key size multiplied with some c table-independent overhead (the work the BM performs to process an entry):

$$\text{size(table)} \cong c * \text{keysize(table)} * \text{entries(table)} \tag{7}$$

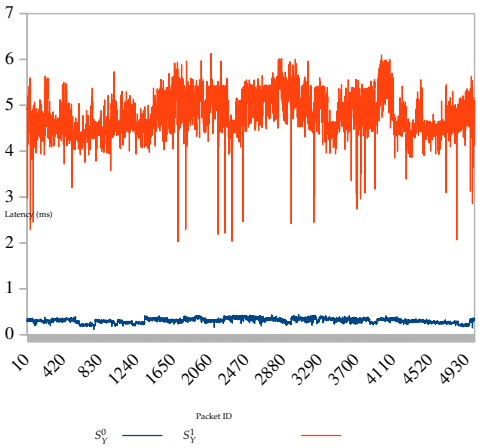


Figure 5. Samples used for cost modeling ($\widehat{\theta}_Y$).

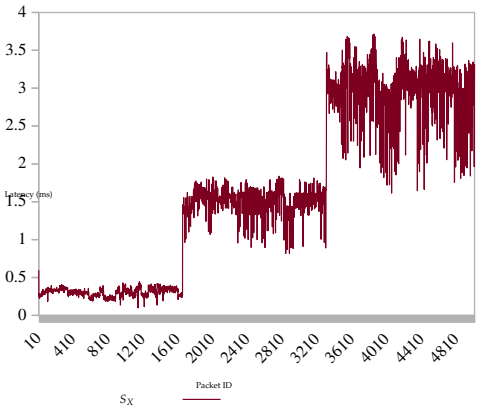


Figure 6. Samples used for validation ($\widehat{\theta}_X$).

To create our cost models, we took the table applications as a primitive instruction together with a special `start` instruction (oo, which accounts for the baseline cost), and as such Inst $= \{$ `start`, `test_table`, `port_mapping`, `bd`, `ipv_fib`, `ipv_fib_lpm`, `nexthop`$\}$. For example, $\widehat{\theta}_Y^{\texttt{max}}x(\texttt{t}, \underline{e})$ denotes the cost of any P4 statement that performs lookups on table t in some worst-case scenario $\underline{e} \in \mathbb{E}^n_{\texttt{max}}$.

To illustrate how we can simply plug in better (or worse) cost models in the static cost analyzer according to our needs, we will derive two different cost models from these measurements. Based on S_Y^1, we constructed a more naive $\widehat{\eta}_Y$ cost model and a more elaborate $\widehat{\theta}_Y$ cost model and examined which one produced better estimates. More specifically, in the $\widehat{\eta}_Y$ cost model (Equation (8)), we assume that table cost grows linearly with the table entries (we calculated the measured cost per entry based on S_Y^1 and used this as a factor to estimate the cost

of executing table t). On the other hand, in the $\widehat{\theta}_Y$ cost model (Equation (9)), we assume that the table cost grows linearly with the actual amount of data moved to perform the table lookup:

$$\widehat{\eta}_Y^{\texttt{max}}(t, \underline{e}) = \begin{cases} \max(S_Y^0(\underline{e})) & \text{if } t = \texttt{start} \\ \text{entries}(t) \cdot \dfrac{\max(S_Y^1(\underline{e})) - \max(S_Y^0(\underline{e}))}{\text{entries}(\texttt{test\_table})} & \text{if } t \text{ is a table} \end{cases} \tag{8}$$

$$\widehat{\theta}_Y^{\texttt{max}}(t, \underline{e}) = \begin{cases} \max(S_Y^0(\underline{e})) & \text{if } t = \texttt{start} \\ \text{size}(t) \cdot \dfrac{\max(S_Y^1(\underline{e})) - \max(S_Y^0(\underline{e}))}{\text{size}(\texttt{test\_table})} & \text{if } t \text{ is a table} \end{cases} \tag{9}$$

Note that in case Equation (7) describes size(t) well, the c overhead constant is eliminated in Equation ((9)). Note also that after the measurements have been performed (in some variable $\underline{e}$ environment), all elements of these equations are known constants. In Section 3.3, we described how we can map Inst to P4 statements and accordingly assign these constant costs to P4 statements in PRISM. To make simple changes in cost models (such as moving from $\widehat{\eta}_Y$ to $\widehat{\theta}_Y$), we only needed to modify a few lines of code in the PRISM representation of the cost model.

4.4. Evaluation

We now attempt to measure S_X and estimate $\widehat{\theta}_X$ (i.e., to dynamically estimate the latency in the $\mathbb{E}$ environment described by Table 5) given the Π program and input space $\mathbb{I}$ described in Table 4. We will use $\widehat{\theta}_X$ to validate our $\vec{\theta}_X$ estimates.

The plot of the S_X sample we collected is depicted in Figure 6. As expected, the plot shows three distinct "steps", corresponding to the paths triggered by the p_1, p_2, and p_3 packet streams. As our device under testing was a general-purpose computer, we speculate that the variance was partly due to the cache behavior. On path π_3, the overall size of the data moved was much more than that on path π_2. This means cache misses were more catastrophic.

Given the program code of `basic_routing-bmv2.p4`, packet distribution $f_{\mathbb{I}}$, and a cost model, PRISM can automatically derive static cost estimates $\vec{\eta}_X$ (based on cost model $\widehat{\eta}_Y$) and $\vec{\theta}_X$ (based on cost model $\widehat{\theta}_Y$). In Appendix A, we discuss how we can check in this simple case that the PRISM indeed returned with correct results. In the scatterplots in Figures 7 and 8, we compare the resulting static estimates with the dynamic estimates ($\widehat{\theta}_X$) we made (based on the S_X sample described earlier) for each statistic defined in Tables 2 and 3.

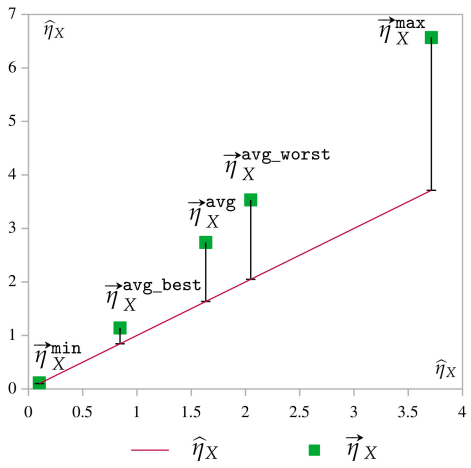


Figure 7. Estimates based on entry counts.

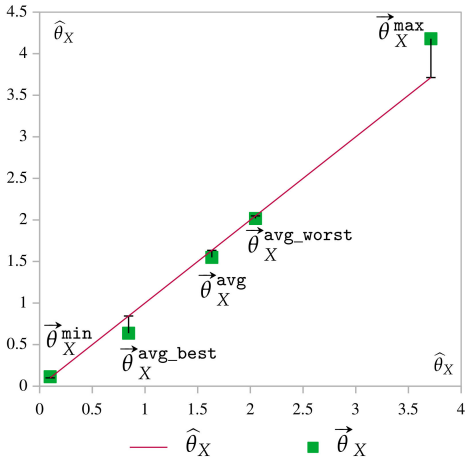


Figure 8. Estimates based on table size.

$\widehat{\eta}_Y$, our naive cost model based on the table entries, did not perform very well. It highly overestimated most of the estimators. As $\widehat{\eta}_Y$ was derived using `test_table`, it was biased toward large-key tables, and consequently, the small-key tables in `basic_routing-bmv2.p4` were consistently overperforming $\vec{\eta}_Y$ estimators.

Comparatively, $\widehat{\theta}_Y$ performed with much fewer errors (e.g., considering the variance in S_X). Due to our reliance on just one sample and the dangers of overfitting, we should still avoid making far-reaching conclusions about the reliability of these estimates. We also had to circumvent the limitations of the (possibly excessively) simple $\widehat{\theta}_Y$ cost model we used in this example in order to keep this study concise. We applied table lookups implemented as linear searching (instead of the more prevalent hash and search tree-based algorithms), we were careful to evade packet buffering, and we only accounted for cost factors aside from table lookups in the simplest possible way. Still, in our opinion, this case study illustrates well that we could plug more elaborate models into the cost analyzer with ease (as most of the analysis is handled automatically), and the resulting estimates reacted positively to new information.

4.5. Additional Capabilities

As seen from the description in Section 3, we only used a very limited subset of the capabilities of PRISM to perform static cost analysis. With very little modification (using a different property), we could transform the tool—using PRISM's non-probabilistic properties—into a conventional model checker for P4 to check the interesting CTL properties or, using PRISM's probabilistic properties, answer questions of a probabilistic nature, such as whether the likelihood that a complex P4 program drops a packet is within acceptable bounds. In the rest of this section, we include a few such examples illustrating the capabilities residing in the tool beyond performance evaluation.

4.5.1. Functional Verification

In this work, our stated goal—performance prediction for P4 programs—pertains primarily to the verification of non-functional requirements. The essence of our solution was to transform P4 into a probabilistic program representation that could be passed to the PRISM probabilistic model checker, which could verify the properties written in PCTL. However, PRISM is also capable of classical model checking (i.e., verifying properties written in CTL). Since CTL is not defined for probabilistic execution models, we need to transform our probabilistic representation into a non-deterministic one, which simply means deleting the probabilities from the transitions. In practice, PRISM automatically performs this transformation.

In P4, when a header is successfully parsed, it is marked as valid. Performing operations over invalid headers is illegal, which is why the P4 source code in Listing 1 contains

a validity check over `hdr.ipv4` before any table is applied. Unfortunately, if the compiler wants to ensure that operations are only performed on previously validated headers, it has to enumerate all program paths leading to the operation, which means this problem has exponential worst-case complexity. This makes model checking a competitive solution for this problem, particularly when the property should be checked against the given program input. Using our representation described in Section 3, we can formalize the above requirement as a straightforward CTL formula:

$$\forall((\square(\mathtt{s}_{833} = 0)) \vee (\square(\mathtt{op} = \mathtt{iread} \wedge 835 \leq \mathtt{z}_1 \leq 963))) \tag{10}$$

The formula in Equation (10) states the validity requirement for the `hdr.ipv4` field of Listing 1. $\square$ is a derived CTL path operator (usually the so-called "global" operator). A path satisfies $\square\Phi$ if all states on the path satisfy Φ. The equation assumes that in the SIL data representation, the bits of `hdr.ipv4` are stored between address 835 and 963 (while 833 and 834 store the validity bit and the header size, respectively). The formula is satisfied if, on all $\pi \in \Pi$ paths, either the validity bit stays $\perp$ all along or the header contents are read only in states where the validity bit is $\top$. As the SIL data representation is known at the compile time, this formula can be extended in a straightforward manner to cover each header in the P4 source file.

4.5.2. Probabilistic Characteristics

Understanding the interplay between protocols and network traffic is useful for optimizing the pipeline. In [12], we examined how the effectiveness of a compiler optimization step—found in the T4P4S compiler—depends on the probability that the size of packet headers is equal to that emitted by a given protocol. Using our current program representation, we can formalize such requirements in PRISM as PCTL formulas that check such probabilities. As an example, we will show the calculation of the probability that a particular statement is being executed. The formula in Equation (11) queries the probability of a path on which table `ipv4_fib_lpm` is eventually applied (supposedly by the SIL instruction having label 40). Given that network traffic is distributed according to Table 4, PRISM returns 0.34, as we expect. A similar PCTL formula or its bounded variant for checking other statements can be generated in a straightforward manner using basic source code analysis:

$$\mathbf{P}_{\Delta}(\Diamond(\mathtt{op} = \mathtt{no\_op} \wedge \mathtt{eip} = 40)) \tag{11}$$

5. Related Work

Automatic performance prediction of P4 switches is a hot topic among P4 researchers. Harkous et al. [8] approached the problem by abstracting P4 programs to some $c_1, \ldots, c_n$ attributes deemed by the authors as potentially important to latency (or other metrics to be estimated), such as the number of parsed headers, modified headers, or tables. In a preparatory phase, similar to our phase that constructs $\widehat{\theta}_Y$, they measured on target-specific testbeds the relationship between the attribute and latency as a $g_i : I_i \to O$ function, where $1 \leq i \leq n$, I_i is the set of possible values of attribute c_i, and O is the measurement space for measuring the effects on latency (e.g., $O = \mathbb{R}^+$). Then, they modeled attribute c_i by constructing the function $f_i : I_i \to O$ that fit the data points in g_i very well. The authors used static analysis (specifically control flow analysis) of the P4 program to select a π program path that was triggered by a specific packet. If the π features attribute c_i with value π_i (e.g., the number of modified headers is π_i), then $f_i(\pi_i)$ is an estimate regarding the effect of attribute c_i on the latency. The complete estimate of the program path π (somewhat comparable to our $\widehat{\theta}_Z$ estimator) was estimated using $(f_1(\pi_1), \ldots, f_n(\pi_n)) \cdot \Delta^\top$, where $\Delta = (\Delta_1, \ldots, \Delta_n)$ is a target-specific vector that weighs the importance of each attribute.

A specific method for modeling g_i was presented by Scholz et al. [19], who first automatically segmented I_i based on using the second derivative of g_i and then constructed f_i using curve fitting over each segment. Through this segmentation, their method is even

capable of modeling deep, target-specific events (that we abstracted as components of $\mathbb{E}$) such as cache misses or target-specific reactions to changes in packet size.

As the earlier parallels to $\widehat{\theta}_Y$ and $\widehat{\theta}_Z$ may have already foreshadowed, we see this approach as complementary to ours. While in our discussion we limited $\widehat{\theta}_Y$ as an estimator over Inst (the set of "instructions"), it would not be too difficult to include the estimators for the attributes selected by these authors into our PRISM cost models either. These estimators could effectively predict the attributes (depending on the deep factors in $\mathbb{E}$) important for performance, while our approach would handle modeling other unknowns probabilistically and inferring program paths based on precise program semantics.

The authors of Flightplan [20] also had to solve a problem similar to calculating the device-dependent $\widehat{\theta}_Y$, although they utilized the results for a different purpose. Flightplan's objective is to use P4 to realize a P4 programmable "one big switch" by segmenting P4 programs into subprograms and allocating them to different devices inside the network. Each program segment requires execution of functions (e.g., performing table lookups, checksums, and parsers). To infer the optimal allocation of segments, Flightplan relies on rule-based inference and formal rules in the form of `constraint` $\xrightarrow{\texttt{device : function}}$ `effect`, with `constraint` describing the conditions (e.g., for the input traffic rate) to determine whether the device named `device` can be used for performing a function `function` and `effect` describing how utilizing the device changes the optimized parameters (e.g., output traffic rate, latency, or power consumption). The authors estimate `constraint` and `effect` by running `function` on `device` under a specific workload. As the result depends on the workload, the estimate is expected to have some error. Flightplan and our tool both build on very similar data: predictions on how specific devices execute specific units of computations. As we do not have constraints on the abstraction level of the "instructions" contained by Inst, we see an opportunity for collaboration here as well.

In this work, we delegate the heavy lifting of cost analysis to PRISM [9], a probabilistic model checker specializing in solving the problem as efficiently as possible using matrix-based computations. We reviewed other approaches for static cost analysis as well. Wegbreit [10], possibly one of first authors on the topic, syntactically transformed Lisp programs into difference equations that could be solved as closed-form performance formulas. A modern take using Wegbreit's approach is [21], where the authors analyzed standard JVM bytecode and used single static assignment to transform this into cost equations and size relations. The former are recursive equations in the form of, for example, $C_p(x_1, x_2) = \sum\{T_b\}_{b \in \text{stmt}(p)} + C_q(y)$, telling us that the cost of executing a program beginning with block p (depending on some x_1 and x_2 attributes of the previous program state) equals the sum of the cost of p's statements plus the cost of executing the next block q (depending on a certain y attribute of the program state). The effect of the statements (the relation between x and y) is abstracted into size relations. For example, size relation $\{y = x_1 + x_2\}$ can express that after performing the concatenation of an x_1-length and an x_2-length list, the resulting list will have a length y. By utilizing a computer algebra system to solve the arising recurrence equations (i.e., to remove free variables using the size relations), the authors derived a closed-form performance formula depending only on the size abstraction of the program input. We note the similarity between $\{T_b\}_{b \in \text{stmt}(p)}$ and our $\widehat{\theta}_Y$. The authors also rely on target-specific profiling to estimate the cost of individual bytecode statements.

P4 shares similarities with workflow languages applied in Service-Oriented Computing—such as Business Process Execution Language (BPEL)—in the sense that it is a high-level description of packet switching workflows, and in many cases, the exact implementation of an individual task is application-specific. Gao. et al. [6] verified the functional and non-functional requirements (e.g., evaluated performance) of service-based systems described as BPEL workflows using PRISM. This analysis can be used for evaluating services and selecting those that are most optimal for the business process. Their approach and ours follow a similar outline: as BPEL cannot represent non-functional, quantitative user requirements (e.g., costs and reliability), the authors transformed BPEL into a formalism called a

Probabilistic Reward Labeled Transition System (PRLTS), which allowed the users to also specify non-functional requirements. The authors automatically generated the verification properties in the PRCTL based on threshold analysis and let users customize the properties on a graphical interface. Finally, the PRLTS model was transformed into PRISM so that it could be checked against the generated verification properties.

6. Conclusions

This paper presented a framework that can automatically estimate the performance of programmable network switches based on P4 [1] source code and probabilistic models of the switch internals (i.e., hardware and software execution environments). As the problem at hand has many domain-specific factors and components, we introduced a formal notation to be able to refer to these factors and components unambiguously in the text. The core idea of the solution is to compile P4 to a Markov chain-based representation used by the PRISM probabilistic model checker [9]. PRISM performs the heavy weight calculations required to predict performance while taking into account the complete semantics of P4. In this way, on the one hand, we can successfully handle input-dependent behavior (such as the P4 switch performing different lookup table operations based on the packet it receives), and on the other hand, we gain a representation that can be extended with probabilistic models of the execution environment. To show the framework in action, we described the results of a case study where we used the tool to estimate the performance of a simple P4 program running on the P4C-BM reference switch [16] and compared these results to estimates gathered using conventional benchmarking over the same use case. With this example, we also illustrated that the framework allows for incrementally adding into it extra information (in the form of more concrete models of the environment) in order to improve the precision of the estimation.

While our method is now complete in the sense that it covers the whole process of inferring estimates from source code and basic information about the environment and expected traffic, there are several open questions about making the framework more powerful and more usable. One avenue of future research is to improve the environmental models used by the tool. In the case study, we used a simplistic linear model for table lookups, but in real life, a table lookup is usually implemented sublinearily. Our earlier work [12] already contains an example on how the DIR-24-8 lookup algorithm (also used by DPDK) can be modeled using Markov chains, which means it should be possible to integrate it into our PRISM-based representation, and this way, we could model DPDK-based targets (e.g., T4P4S [2]) as well. A similarly interesting question is how more complex hardware behavior could be modeled into the tool. While questionable on the feasibility side, in theory, we could integrate complete, well-defined hardware models automatically by relying on tools like P4-to-VHDL [22] to first create deep, target-specific representations of the P4 pipeline and then compile this representation to PRISM for very high-precision performance prediction. In this work, we discussed packet processing as a sequential process, but industrial-grade switches such as the Intel Tofino series are capable of processing multiple packets in parallel. At the same time, PRISM is capable of modeling concurrent processes out of the box. As such, performance prediction of concurrent P4 packet processing also seems to be a lucrative path to explore.

Author Contributions: Problem formulation, solution, and validation, D.L.; review and supervision, G.P. and M.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the ÚNKP-21-4 New National Excellence Program of the Ministry for Innovation and Technology from the National Research, Development and Innovation Fund. This research was supported by the project "Application Domain Specific Highly Reliable IT Solutions", implemented with the support of the NRDI Fund of Hungary and financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme. This research was supported in part by project no. FK_21 138949, provided by the National Research, Development and Innovation Fund of Hungary.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Appendix A

We intentionally designed the case study to be simple so that the correctness of the estimates could be checked manually. In this appendix, we show how to check the correctness of $\vec{\theta}_X^{\texttt{avg\_best}}$ in Figure 8. In Table A1, we include the numeric results from the bootstrapping measurements depicted in Figure 5. Based on this data, we also calculated the factors in Equations (8) and (9).

Table A1. Measurements and factors for Equation (9).

g	`max`	`avg`	`min`
$S_Y^0(\underline{e})$	0.438	0.309394	0.114
$S_Y^1(\underline{e})$	6.13	4.810818	2.031
$\frac{g(S_Y^1(\underline{e}))-g(S_Y^0(\underline{e}))}{entries(\texttt{test\_table})}$	0.000057	0.000045	0.000019
$\frac{g(S_Y^1(\underline{e}))-g(S_Y^0(\underline{e}))}{size(\texttt{test\_table})}$	0.009714	0.007682	0.003272

Based on Equations (7)–(9), we then calculate the instruction costs $\widehat{\theta}_Y^{\texttt{min}}(t, \underline{e})$ for each t table. We collected the results of this calculation in Table A2.

Table A2. Instruction costs of lookup tables.

t	size(t)	$\widehat{\theta}_Y^{\texttt{min}}(t, \underline{e})$
`port_mapping`	11	0.0359
`bd`	39	0.1275
`ipv4_fib`	0	0
`ipv4_fib_lpm`	269	0.8799
`nexthop`	39	0.1275

Finally, since we are looking for $\vec{\theta}_X^{\texttt{avg\_best}}$, we want to calculate the best-case expected time over the paths in our use case. As we described in the main text, by using the source code in Listing 1 and the packet distribution in Table 4, we can infer that the second path (having only `port_mapping`, `bd`, and `nexthop` tables) is executed with a probability of 0.33, while the third path (with all the tables, including `ipv4_fib_lpm`) is executed with a probability of 0.34. On the first path, no tables are executed, but even here (as well as on the other two paths), we have to account for the overhead cost (0.114). The expectation is then given by Equation (A1).

$$\begin{aligned} & 0.114 \\ + \quad & 0.33 * (0.0359 + 0.1275 + 0.1275) \\ + \quad & 0.34 * (0.0359 + 0.1275 + 0.8799 + 0.1275) \\ \cong \quad & 0.608 \end{aligned} \tag{A1}$$

We can now compare $\vec{\theta}_X^{\texttt{avg\_best}}$ in Figure 8 with this value to check whether the state-based calculation by PRISM returns a plausible result.

References

1. Bosshart, P.; Daly, D.; Izzard, M.; McKeown, N.; Rexford, J.; Talayco, D.; Vahdat, A.; Varghese, G.; Walker, D. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 87–95. [CrossRef]
2. Laki, S.; Horpácsi, D.; Vörös, P.; Kitlei, R.; Leskó, D.; Tejfel, M. High Speed Packet Forwarding Compiled from Protocol Independent Data Plane Specifications. In Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16, Florianopolis, Brazil, 22–26 August 2016; ACM: New York, NY, USA, 2016; pp. 629–630. [CrossRef]
3. Dang, H.T.; Wang, H.; Jepsen, T.; Brebner, G.; Kim, C.; Rexford, J.; Soulé, R.; Weatherspoon, H. Whippersnapper: A P4 Language Benchmark Suite. In Proceedings of the Symposium on SDN Research, SOSR '17, Santa Clara, CA, USA, 3–4 April 2017; Association for Computing Machinery: New York, NY, USA, 2017; pp. 95–101. [CrossRef]
4. Cicioglu, M.; Calhan, A. A Multi-Protocol Controller Deployment in SDN-based IoMT Architecture. *IEEE Internet Things J.* **2022**, 1. [CrossRef]
5. Kim, H.; Reich, J.; Gupta, A.; Shahbaz, M.; Feamster, N.; Clark, R. Kinetic: Verifiable Dynamic Network Control. In Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, NSDI'15, Oakland, CA, USA, 4–6 May 2015; USENIX Association: Berkeley, CA, USA, 2015; pp. 59–72.
6. Gao, H.; Miao, H.; Liu, L.; Kai, J.; Zhao, K. Automated Quantitative Verification for Service-Based System Design: A Visualization Transform Tool Perspective. *Int. J. Softw. Eng. Knowl. Eng.* **2018**, *28*, 1369–1397. [CrossRef]
7. Gao, H.; Zhang, Y.; Miao, H.; Barroso, R.J.D.; Yang, X. SDTIOA: Modeling the Timed Privacy Requirements of IoT Service Composition: A User Interaction Perspective for Automatic Transformation from BPEL to Timed Automata. *Mob. Netw. Appl.* **2021**, *26*, 2272–2297. [CrossRef]
8. Harkous, H.; Jarschel, M.; He, M.; Kellerer, W.; Priest, R. P8: P4 With Predictable Packet Processing Performance. *IEEE Trans. Netw. Serv. Manag.* **2020**, *18*, 2846–2859. [CrossRef]
9. Kwiatkowska, M.; Norman, G.; Parker, D. PRISM 4.0: Verification of Probabilistic Real-time Systems. In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11), Snowbird, UT, USA, 14–20 July 2011; Gopalakrishnan, G., Qadeer, S., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6806, pp. 585–591.
10. Wegbreit, B. Mechanical Program Analysis. *Commun. ACM* **1975**, *18*, 528–539. [CrossRef]
11. Sapio, A.; Baldi, M.; Pongrácz, G. Cross-Platform Estimation of Network Function Performance. In Proceedings of the 2015 Fourth European Workshop on Software Defined Networks, Bilbao, Spain, 30 September–2 October 2015; pp. 73–78. [CrossRef]
12. Lukács, D.; Pongrácz, G.; Tejfel, M. Control flow based cost analysis for P4. *Open Comput. Sci.* **2020**, *11*, 70–79. [CrossRef]
13. Baier, C.; Katoen, J.P. *Principles of Model Checking (Representation and Mind Series)*; The MIT Press: Cambridge, MA, USA, 2008.
14. P4 Language Consortium. $P4_{16}$ Language Specification, Section 6.7. Calling convention: Call by Copy in/Copy out. 2022. Available online: https://p4.org/specs/ (accessed on 31 May 2022).
15. Gao, H.; Dai, B.; Miao, H.; Yang, X.; Barroso, R.J.D.; Walayat, H. A Novel GAPG Approach to Automatic Property Generation for Formal Verification: The GAN Perspective. *ACM Trans. Multimed. Comput. Commun. Appl.* 2022. *Just Accepted*. [CrossRef]
16. P4 Language Consortium. The Reference P4 Software Switch. 2012. Available online: https://github.com/p4lang/behavioral-model/ (accessed on 31 May 2022).
17. P4 Language Consortium. Basic_routing-bmv2.p4, Official P4 Reference Compiler Test Case, P4C. 2022. Available online: https://github.com/p4lang/p4c/blob/master/testdata/p4_16_samples/basic_routing-bmv2.p4 (accessed on 31 May 2022).
18. Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual, Section E.4.4 Cache and Memory Subsystem. 2022. Available online: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf (accessed on 31 May 2022).
19. Scholz, D.; Harkous, H.; Gallenmüller, S.; Stubbe, H.; Helm, M.; Jaeger, B.; Deric, N.; Goshi, E.; Zhou, Z.; Kellerer, W.; et al. A Framework for Reproducible Data Plane Performance Modeling. In Proceedings of the Symposium on Architectures for Networking and Communications Systems, ANCS '21, Layfette, IN, USA, 13–16 December 2021, Association for Computing Machinery: New York, NY, USA, 2021; pp. 59–65. [CrossRef]
20. Sultana, N.; Sonchack, J.; Giesen, H.; Pedisich, I.; Han, Z.; Shyamkumar, N.; Burad, S.; DeHon, A.; Loo, B.T. Flightplan: Dataplane Disaggregation and Placement for P4 Programs. In Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), Online, 12–14 April 2021; USENIX Association: Berkeley, CA, USA, 2021; pp. 571–592.
21. Albert, E.; Arenas, P.; Genaim, S.; Puebla, G.; Zanardini, D. Cost Analysis of Java Bytecode. In *Programming Languages and Systems*; De Nicola, R., Ed.; Springer: Berlin/Heidelberg, Germany, 2007; pp. 157–172.
22. Benácek, P.; Pu, V.; Kubátová, H. P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers. In Proceedings of the 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Washington, DC, USA, 1–3 May 2016; pp. 148–155. [CrossRef] .