P4CKET: P4 Program Optimization through Online Profiling

Darshil D. Kaneria
Carnegie Mellon University

Sai Jaydeep Kudumula
Carnegie Mellon University

Abstract

Modern network programmability requires sophisticated optimization techniques to address complex hardware constraints. The increase in deployment of programmable network devices has exposed critical bottlenecks in resource allocation and performance efficiency.

This paper introduces a novel algorithm that leverages online profiling to dynamically guide P4 program optimizations during the compilation phase. It is made possible by adaptively instrumenting the P4 program based on select runtime statistics which involve heat maps and hot path information. Our approach allows the P4C compiler to make intelligent optimization decisions such as consolidating multiple tables into a single stage and offloading infrequently accessed tables to the controller. We reduce the involvement of an administrative entity by tightening the information exchange pipeline between the controller, controller, and the deployment pipeline. Experimental evaluations show that our proposed optimization technique can reduce the overall stage allocation memory by upto 40% while mostly maintaining the performance expectation of the P4 program.

1 Introduction

The network landscape has changed with the increased deployment of programmable network devices. This has brought proposals for static compilation strategies and hardware-specific code transformations. However, these approaches often fail to capture the dynamic nature of traffic, highlighting the need for adaptive instrumentation techniques that can identify runtime information and optimize program execution by adjusting the memory mapping of logical stages to physical hardware, and by removing redundant constructs in P4 programs.

Current optimization techniques rely on static methods like table consolidation, stage reordering, and offloading infrequently used functionality to the controller. A common shortcoming of these approaches is their inability to capture the variability in network traffic where traffic patterns, performance requirements, and workload characteristics may change rapidly. We consider runtime information critical for compilers of P4 programs. Instrumentation is as important as information gathered via static analysis. However, inflexible instrumentation techniques can be detrimental to P4 program execution, especially in latency-sensitive environments like data centers.

We introduce P4CKET, a closed-loop framework that incorporates runtime information into P4C's platform-independent optimization pipeline in an online manner. The framework captures runtime characteristics such as (a) path a packet takes through the ingress pipeline, (b) aggregate hit-rate of tables and actions over time, (c) variability of traffic over time. Characteristics (a) and (b) help identify dependencies between tables and actions. This information is forwarded to the compiler as an interference graph and used to adjust control flow in the apply blocks of the ingress pipeline, which may change hardware-specific stage allocation to reduce memory footprint. Characteristic (c) determines the frequency and intervals for program recompilation and redeployment, automatically tuned through controller feedback in a closed-loop system, reducing the need for an administrative entity to intervene. P4CKET preserves program semantics by rearranging tables instead of transforming them. This approach reduces compiler pass complexity but limits potential optimizations like table consolidation, reduction, and elision. The framework's goal is to conserve memory on silicon while preserving program semantics and meeting switch performance requirements. We explore the following high level question: *What is the cost of adaptive instrumentation, and how does this extra dimension of information allow the compiler to make better optimization decisions?*

The paper is structured as follows: We explore related works in §2, followed by an introduction to P4 and the open-source compiler P4C (§3), which we incorporate into our framework. Next, we discuss the framework's design (§4), split into analysis and optimization phases. The analysis phase describes adaptive instrumentation, generating a heat-map of table accesses and identifying hot paths. The optimization phase explains how this information is passed to the compiler and used in mid-end passes to restructure control statements. We then present our experimental setup and evaluation results (§5). The future works section (§6) outlines optimization possibilities through instrumentation information. Finally, we provide closing remarks (§7) and discuss surprises we encountered during the project.

2 Related Works

This section examines prior research in P4 program optimization, and runtime instrumentation, highlighting the limitations that motivate our proposed approach. We analyze existing techniques in profile-guided optimizations to contextualize the contributions of the P4CKET framework.

Our research acknowledges the crucial optimizations done as part of previous work. P5 [2] and P4CGO [7] both propose optimizations that include external inputs in the form of control plane policies, but they fail to capture changes to these policies or traffic pattern. P2GO [8] focuses on analyzing the pre-captured traffic patterns (offline profiling) to determine the dependency between the tables and optimize the stage memory allocation by consolidating tables into fewer stages. However, since it relies on offline profiling, it does not adapt itself to varying traffic.

The approach mentioned in this paper is inspired by Morpheus [6], which introduces the adaptive instrumentation technique to software data planes. It tracks map access patterns by instrumenting the program to monitor how and where data structures (referred to as maps in eBPF/DPDK) are accessed during runtime and uses this information to pinpoint heavy hitters. We bring this concept to P4 switches and improve the recompilation process by considering different heuristics.

3 A Primer on P4 and the Compiler

P4 [3] [5] is a domain-specific language for programming packet-processing pipelines (and more recently, even NICs). More recently, it has seen widespread adoption due to an increased demand in forwarding pipeline configurability combined with improvements at the silicon level. P4 provides constructs for defining packet headers, parsers for extracting these headers, match-action tables for processing, and control blocks to sequence operations.

The P4C compiler is the reference implementation for translating P4 programs into target-specific configurations or runtime code. P4C provides multiple backends to help debug target-specific constraints. In our experimentation, we make use of two such backends – BMv2 and Tofino. BMv2 is a software switch implementation which is used in preliminary testing to check the correctness of the program while the Tofino backend is used to get accurate stage allocation information for table placement based on actual hardware specifications. P4C operates in three stages: a front-end that parses the P4 source code and validates its syntax and semantics, a mid-end that performs target-independent optimizations, and a back-end that generates target-specific code. It uses the visitor model to traverse and transform the abstract syntax tree representing the program. The preorder and postorder methods of the visitor model are used to traverse the AST in a specific sequence. The preorder method processes a node before its children, which is useful for initializing state or making decisions based on the parent node. In contrast, the postorder method processes a node after its children, typically used for aggregating results or doing transformations based on the descendants. In context of this paper, we are mostly interested in the way our mid-end optimizations affect the table placement during the back-end optimization phase for Tofino backend. So, we insert multiple mid-end passes which define several of these methods to gather information about the dependency between tables, and transform conditional statements or insert annotations such that the compiler is able to correctly identify the true dependency between tables. A more detailed description of how P4C performs table placement for the Tofino backend is provided in section §4.

4 Design Overview

The implementation of P4CKET is split into two phases: the analysis phase and the optimization phase (followed by the recompilation and deployment phase, but we consider these to be part of the same logical phase).

4.1 Analysis Phase

A common requirement for all the optimizations mentioned in this paper is the need for runtime information to adapt to varying traffic patterns. The minimum amount of information that can be gathered from the P4 program is achieved by instrumenting all available actions for the tables in the ingress pipeline. The high-level approach is to create a header (which we will call *tcount*) where the keys are the tables, and the value for each field is a bit vector, with each position representing an action that could be taken by the table. Each field's value is updated based on the action applied to the packet at any given point in the pipeline. Before the packet exits the switch through its designated port, the header is inserted into it. We make the assumption here that all packets will have the Ethernet header as the top-level header, and thus we place the *tcount* header below the Ethernet header. We have created a custom pass called `addTcountHeader`, which performs the operations described above.

A naive approach would be to add these headers at the beginning of every action and never change them. The obvious downside stems from the fact that most of these switches operate in a latency- and bandwidth-sensitive environment (e.g., data centers). Packets across all paths through the pipeline must indiscriminately bear the cost of the instrumentation instructions, which translates into consistently higher latency and worse bandwidth. To mitigate this cost, we propose an adaptive instrumentation approach that conditionally instruments the actions for a table based on whether the table is part of a hot path or not.

4.1.1 Adaptive Instrumentation

To assuage the effects of instrumentation on high-level performance, we selectively instrument paths of the pipeline. When the program is first deployed, all the actions for all the tables are instrumented. This initial full coverage helps us identify the first hot path. The controller samples these packets at a lower rate to reduce the pressure on the switch's crossbar. For every packet that it samples, the controller maintains an element-wise sum per field of the tcount header. The information is stored in the form of a 2D array. Each row in the array corresponds to a field in the header (which, by extension, refers to a table in the switch). Each element of a row corresponds to the number of times an action was applied to the packets. After some time has elapsed (i.e., after the refresh interval, which is discussed later), the controller computes the heat map for the switch and determines the hit rate for each table. It also computes the table interference graph, which is crucial in determining the true table dependency for a particular traffic profile. The heart of the adaptive instrumentation algorithm lies in how we use this information to elide instrumentation. The group of tables with the highest aggregate hit rate forms the hot path (Figure 1). Based on the initial table dependency identified by the compiler, we infer the order in which the packets traverse the tables in the hot path. For tables included in the hot path, we elide the instrumentation on the actions that are most frequently applied. For tables not part of the hot path, we retain all the instrumentation. The reasoning behind this is that packets traversing the hot path frequently trigger these actions, incurring a small cost that accumulates by the time the packet reaches the egress pipeline. Moreover, the main use of instrumentation is to help the controller find the appropriate hot path for a particular traffic pattern and detect changes in the hot path. By retaining instrumentation across the cold path, we are able to detect changes in the path the packets take when we see an uptick in the number of actions triggered in tables on the cold path. During the next interval, a new aggregate hit rate is computed, and if it exceeds the hit rate for the current hot path (which has been frozen), this new path is declared the hot path. The instrumentation in the former path is restored, followed by the elision of instrumentation from the new hot path.

4.1.2 Insertion of tcount Headers

During the first compilation, the P4CKET adds mid-end passes to the P4C compiler to modify the program in the following ways:

Table	Hit Percentage
ipv4_lpm	100.0%
udp_acl	11.81%
dhcp_acl	15.45%

Table 1: Hit Percentages for acl_test.p4 with pseudorandom traffic generation

1. The compiler generates information about the number of tables and actions in the program and converts into a form which the next pass is able to digest. This pass is defined in the `countActionTables.cpp`
2. Information is transferred between two passes using a shared data struct which stores the action count and table count defined in `sharedData.h`
3. This information is consumed by the second pass which is `addTcount.h` which does the following:
    (a) Insert the header declaration
    (b) Create corresponding fields in the metadata structure which is updated throughout the ingress pipeline
    (c) Insert the instructions to send packets to the controller based on a sampling rate calculated based on the refresh interval and the traffic variability.
    (d) Insert instructions to emit the header under the ethernet header.
    (e) Insert update instructions which record the actions applied on the packet inside the appropriate field in the tcount header.

All the passes, except the last one, fall under the `Inspector` category, which allows a pass to read information but not modify it. As we shall see in the next subsection, the other passes are of type `Transform`, which allow a visitor method to update the underlying AST by modifying an existing node or inserting/removing a new one.

4.2 Optimization Phase

The next phase in P4CKET involves sending feedback from the controller to the compiler through a shared channel. This feedback is then used by the compiler passes to guide its behavior, annotating and restructuring the control statements that control the application of tables in the ingress pipeline.

4.2.1 Primer on RMT

Currently, P4CKET only supports switches that implement the Reconfigurable Match-Action Table architecture [4]. An

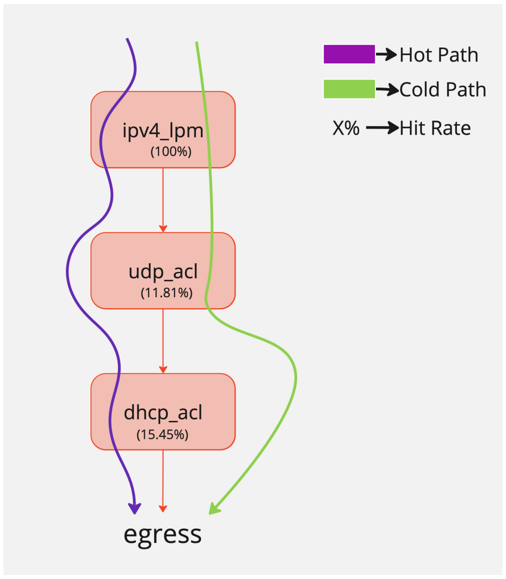

Figure 1: Hot paths based on the hit rate for each table combined with the table interference information.

```
apply {
    hdr.tcount.setValid();
    if valid(ipv4) {
        ipv4_lpm.apply();
        if valid(udp) {
            udp_acl.apply();
            if valid(dhcp) {
                dhcp_acl.apply();
            }
        }
    }
}
```

Figure 2: P4 Program Control Flow

```
header tcount_t {
    bit<32> ipv4_lpm;
    bit<32> udp_acl;
    bit<32> dhcp_acl;
}
```

Figure 3: tcount_t Header

```
action ipv4_forward(dstAddr, port) {
    // Instrumented line below
    md.ipv4_lpm = md.ipv4_lpm | 1;
    stdmd.egress_spec = port;
    ether.srcAddr = ether.dstAddr;
    ether.dstAddr = dstAddr;
    ipv4.ttl = ipv4.ttl - 1;
}
```

Figure 4: Action Instrumentation

RMT switch consists of multiple stages, each with configurable match-action units that perform packet processing tasks such as header parsing, classification, and modifications. Packets pass through a sequence of these stages, where they are matched against rules stored in TCAM or SRAM, and actions are applied based on the matches.

4.2.2 Reducing the Memory Footprint

A compiler with no information about true table dependencies might naively place tables in separate stages, even when the tables are mutually exclusive for a given traffic pattern (i.e., if the actions of table A and table B never act on the same packet). Table placement in Tofino uses simple rules to make the best use of Match-Action Unit (MAU) resources for different situations. The two main rules are:

1. Always prioritize the order of dependencies.
2. Balance between dependencies and resource usage.

These rules also allow for backtracking to fix issues when an impossible dependency chain is found. Table placement decisions are made in the `preorder(IR::BFN::Pipe *)` method, which allocates tables to logical tables and resources, storing decisions in `Placed` objects for backtracking. After this method completes, the transform rewrites tables to match the `Placed` list, splitting tables as needed.

This process uses a greedy allocator, maintaining a work list of `TableSeq` objects. For each table, a new `Placed` object is created and linked to the front of the `done` list. Possible placements are compared using the `is_better` method to determine the best choice.

Considering the example from Figure 2, 3, and 4 we annotate the tables as follows:

```
// IPv4 LPM Table
@stage(1)
table ipv4_lpm {
    ...
}

// UDP ACL Table
@stage(2)
```

```
table udp_acl {
   ...
}

// DHCP ACL Table
@stage(2)
table dhcp_acl {
   ...
}
```

We also perform control flow manipulation to structure the conditionals in such a way as to explicitly indicate to the compiler the lack of true table dependency. Considering the example mentioned in Figure 2, the `udp_acl.apply()` statement would be moved inside the conditional which checks for the validity of the dhcp header. The resulting conditional is `!udp_acl.apply().hit && valid(dhcp)`. This hints the compiler with the absence of true dependence between the `udp_acl` and `dhcp_acl` tables, and allows it to perform more efficient stage allocation *if physically possible*.

4.2.3 Offloading Functionality to Controller

. A much simpler optimization is to identify the tables with the lowest hit rate for a specific traffic pattern. If their hit rate does not exceed a certain threshold, they are offloaded to the controller. The offloaded table is replaced with a placeholder table that punts packets to the controller using an action that is a superset of the original actions in the offloaded table. This approach reduces memory usage and makes more memory available to other tables that could benefit from it. If the hit rate increases, the functionality is brought back to the switch.

4.2.4 Adjusting the Refresh Interval

The refresh interval controls:

1. The packet sampling frequency for the controller
2. The recompilation frequency for the P4 program

For now, it is based on the single heuristic of traffic variance – If the traffic variance is high, the refresh interval is higher to capture potential changes in the hot path. If the traffic does not vary as much, we reduce the refresh interval. The packet sampling frequency is directly proportional to the refresh interval.

5 Experimental Setup and Evaluation

To validate our initial hypothesis, we set up a 2-host-1-switch configuration in a Mininet [1] environment. The underlying binary for the switch is the BMv2 `simple_switch_grpc` software switch. The controller is created outside the Mininet environment but is connected to the switch via the dedicated CPU port (Figure 5). Once the controller starts, it performs the following steps:

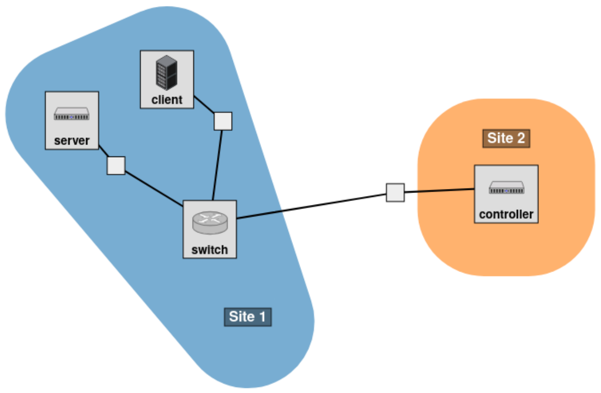

Figure 5: Mininet setup for evaluation

1. Set up the Mininet topology by reading from the config file.
2. Invoke the compiler to create the initial runtime binary and deploy it to the switch.
3. Add entries to the tables based on the test being evaluated.
4. Every refresh interval, pass information to the shared channel and invoke the compiler *if the hit rate information has been updated*.
5. Re-deploy the P4 program if needed.

5.1 Efficient Stage Allocation

As of the writing of this paper, we do not have access to Tofino hardware and therefore cannot generate accurate memory utilization and power consumption metrics. Instead, we infer these metrics from the stage allocation logs shown in Figure 6, which indicate a reduction in the number of stages allocated by an average of 40%.

5.2 Cost of Adaptive Instrumentation

The adaptive instrumentation algorithm used by P4CKET aims to minimize its impact on latency and bandwidth. During our experiments, we configured the links such that, under optimal conditions, the RTT for a packet from Host 1 to Host 2 through the switch is 20 ms, and the maximum supported bandwidth for each link is 100 Mbps. We capture the average latency and throughput across 10 runs for each test case with and without the instrumentation. From Figure 7, we observe a significantly lower average throughput across all test cases,

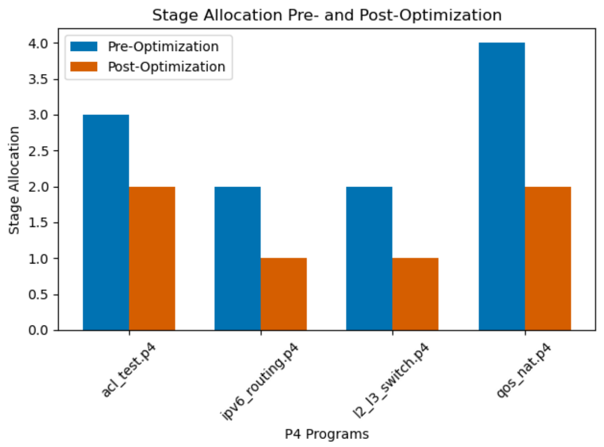

Figure 6: Stage allocation comparison for benchmark P4 programs. A reduction in stage allocation indicates compaction of multiple tables into a single stage which makes additional stages available for other functionality

which is attributed to the overhead of the BMv2 software switch. However, we observe that latency remains unaffected across all test cases, while bandwidth is significantly impacted. Investigating this issue is left as future work due to time constraints.

6 Future Work

While we have demonstrated the potential benefits of adaptively instrumenting P4 programs, the impact on throughput highlights the need for further improvements in hot path detection heuristics and exploration of alternative instrumentation techniques. Future work could also evaluate this approach on real hardware and analyze differences in power consumption.

Additional profile-specific optimizations, such as dynamically reducing table memory based on action invocation, are also worth exploring.

7 Conclusion

7.1 Discussion

In this paper, we introduce P4CKET, a framework for optimizing P4 programs by utilizing traffic profile information through online profiling techniques such as adaptive instrumentation. We evaluate the efficacy of our approach using four different test cases. The evaluation demonstrates an improvement in the number of stages allocated for Match-Action Units with negligible impact on end-to-end latency. However, we also observe a significant reduction in average throughput and leave the investigation of this issue for future work. Towards the end of the study, we noticed that the P4C community has open-sourced the Tofino backend for the P4C compiler, allowing us to generate the stage allocation reports for this specific switch hardware.

7.2 Distribution of Total Credits

We would prefer to equally split (50%-50%) the credits earned for this research project.

References

[1] Mininet.

[2] Abhashkumar, A., Lee, J., Tourrilhes, J., Banerjee, S., Wu, W., Kang, J.-M., and Akella, A. P5: Policy-driven optimization of p4 pipeline. In *Proceedings of the Symposium on SDN Research* (New York, NY, USA, 2017), SOSR '17, Association for Computing Machinery, p. 136–142.

[3] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev. 44*, 3 (July 2014), 87–95.

[4] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding metamorphosis: fast programmable match-action processing in hardware for sdn. *SIGCOMM Comput. Commun. Rev. 43*, 4 (Aug. 2013), 99–110.

[5] Budiu, M., and Dodd, C. The p416 programming language. *SIGOPS Oper. Syst. Rev. 51*, 1 (Sept. 2017), 5–14.

[6] Miano, S., Sanaee, A., Risso, F., Rétvári, G., and Antichi, G. Domain specific run time optimization for software data planes. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2022), ASPLOS '22, Association for Computing Machinery, p. 1148–1164.

[7] Wen, C., Li, Z., Jafri, S. U., Qiu, X., and Rao, S. P4cgo: Control plane guided p4 program optimization. In *Proceedings of the 2024 SIGCOMM Workshop on Formal Methods Aided Network Operation* (New York, NY, USA, 2024), FMANO '24, Association for Computing Machinery, p. 1–7.

[8] Wintermeyer, P., Apostolaki, M., Dietmüller, A., and Vanbever, L. P2go: P4 profile-guided optimizations. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2020), HotNets '20, Association for Computing Machinery, p. 146–152.

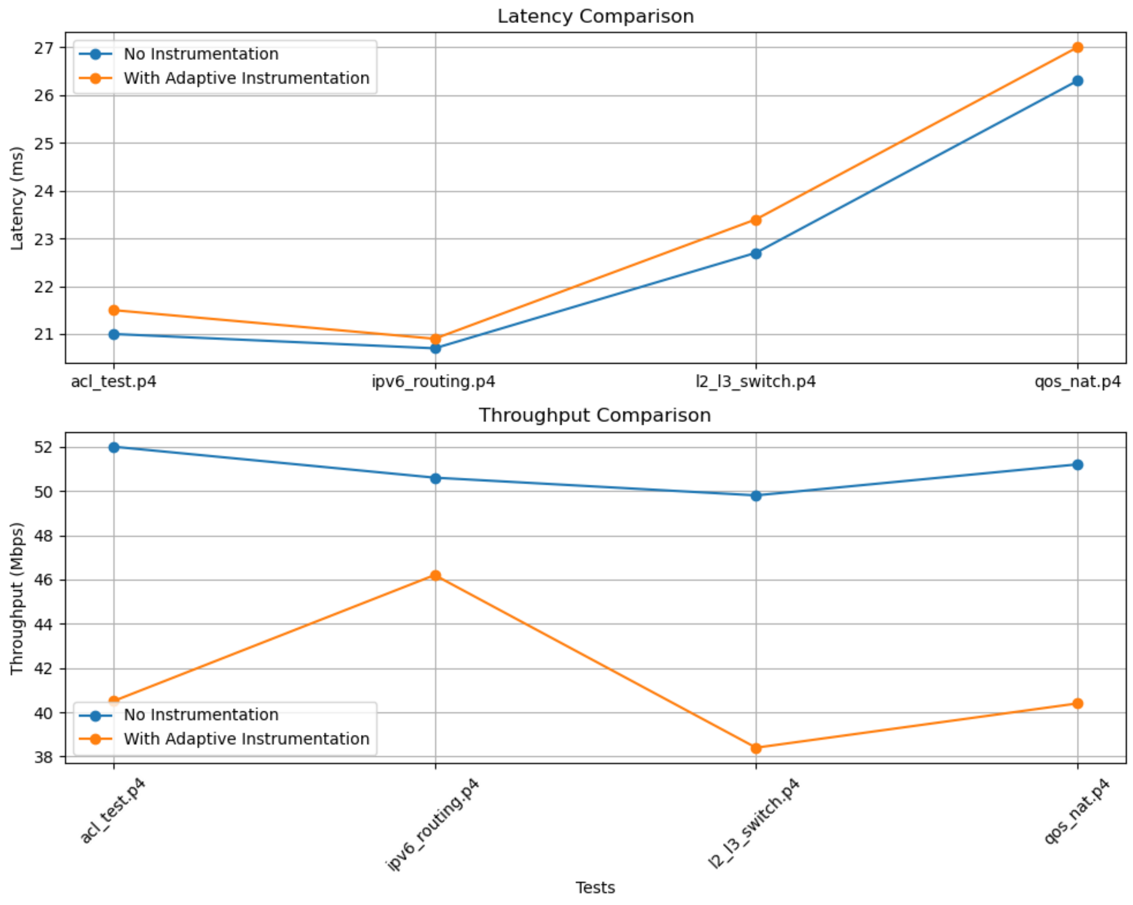

Figure 7: Effect of Adaptive Instrumentation on the average latency and throughput.