Newton: Intent-Driven Network Traffic Monitoring

Yu Zhou[†§], Dai Zhang[†], Kai Gao[⋆], Chen Sun[§], Jiamin Cao[†],
Yangyang Wang[†], Mingwei Xu[†], Jianping Wu[†]
[†]Institute for Network Sciences and Cyberspace, BNRist, Tsinghua University
[§]Alibaba Group [⋆]Sichuan University

ABSTRACT

Monitoring network traffic based on operators' intents is essential to today's networks. As the bandwidth and size of networks increase steeply, monitoring systems shall fulfill the requirements of on-demand network monitoring for ever-growing traffic volumes. However, existing monitoring systems either cannot satisfy operators' intents on demand or introduce substantial monitoring overheads. In this paper, we present *Newton*, an intent-driven traffic monitor that enables specifying operators' intents with traffic monitoring queries and supports dynamic and scalable network-wide queries. Specifically, *Newton* 1) empowers operators to dynamically create, remove, and update on-data-plane queries without interrupting normal packet forwarding, 2) conducts systematic optimizations to achieve precise network traffic monitoring, and 3) executes network-wide queries with high resilience to dynamic network status. Evaluation results show that *Newton* improves the flexibility, scalability, and resource efficiency of traffic monitoring, demonstrating its great potential to be deployed in large-scale programmable networks.

CCS CONCEPTS

• **Networks** → **Network monitoring**; **Programmable networks**.

KEYWORDS

Network traffic monitoring, programmable switch, P4

ACM Reference Format:
Yu Zhou, Dai Zhang, Kai Gao, Jiamin Cao, Yangyang Wang, Mingwei Xu, Jianping Wu. 2020. *Newton*: Intent-Driven Network Traffic Monitoring . In *The 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20), December 1–4, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3386367.3431298

1 INTRODUCTION

Traffic monitoring plays a key role in a variety of network operation tasks. Network operators need to capture heavy-hitter flows for traffic engineering to maximize bandwidth utilization [1, 2] and track abnormal traffic changes to detect attacks [3, 4] or to troubleshoot failures [5, 6]. Traditional traffic monitoring systems, *e.g.* sFlow [7] and NetFlow [8], provide coarse-grained monitoring information, which cannot satisfy operators' intents.

The current advance of programmable switches [9–12] opens new opportunities to monitor traffic based on operators' intents. Operators can implement traffic monitoring intents with domain-specific programming languages, such as and NPL [12], POF [13], and P4 [14], and then run programs that realize intents on data planes at line rate. The above procedure is referred to as *intent-driven traffic monitoring*. Owing to increased network scale and device bandwidth as well as operation demands on fine-grained monitoring data, building an efficient traffic monitoring system shall satisfy two fundamental requirements.

- **1. Deploy traffic monitoring intents on demand without disturbing normal packet forwarding**. As operators need to deploy new monitoring intents continuously in running networks, monitoring systems shall support dynamic installation of new monitoring tasks. Furthermore, operation requirements change over time [15–18], which drives continuous update of monitoring intents. For example, operators need to update monitoring tasks to drill down into sources of anomaly traffic when detecting DDoS attacks. Thus, monitoring systems should support *dynamic operations* (*e.g.* update and remove) of monitoring tasks to ensure reactiveness to network events and evolved intents. The premise of supporting on-demand monitoring intents lies in not disturbing normal packet forwarding, including connectivity and forwarding performance.
- **2. Make monitoring overheads low for scalability**. Monitoring overheads, including bandwidth for data transmission and CPU cores for analysis [19–21], should not increase severely as network sizes and traffic volume increase. Because large bandwidth consumption could impede the performance of normal traffic, and large CPU consumption could incur capital and operation investment.

Unfortunately, existing intent-driven traffic monitoring systems cannot satisfy the above requirements simultaneously. First of all, Sonata [19] and Marple [22] abstract networks as a database. They specify operators' intents by *traffic monitoring queries* (*e.g.* based on SQL and Spark API [23]) over the database and load queries into switches as the form of P4 programs. Then, the switches export exact information required by intents with small overheads. But Sonata and Marple only support static operations of monitoring tasks, and updating queries needs to reboot switches, which makes switches halt for seconds. Second, TurboFlow [24] and *Flow [17] execute queries on software analyzers to provide on-demand monitoring intents and aggregate monitoring information on data planes. But their monitoring overheads increase fast with network sizes and traffic volume, restricting the scalability.

CoNEXT '20, December 1–4, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7948-9/20/12…$15.00
https://doi.org/10.1145/3386367.3431298

This paper proposes *Newton*, an intent-driven NEtWork Traffic mONitoring system. *Newton* uses a widely-used high-level query API, Spark API [23] (alike Sonata [19]), to express traffic monitoring intents and compiles the queries to switch configurations, which frees operators from trivial underlying implementations. On data planes, *Newton*'s queries perform accurate monitoring data exportation for high scalability and can be dynamically and safely reconfigured. On control planes, a centralized controller compiles and deploys queries. As for *Newton*'s realization, we shall address three challenges.

The first challenge is to **realize a variety of queries on programmability limited data planes**. A query could consist of various primitives (*e.g. map*, *reduce*, and *filter*) and parameters [25]. On-demand query operations require great flexibility of query reconfiguration, which is hard for data planes. Existing data plane virtualization proposals [26–28] provision such flexibility but severely compromise performance. We observe that the query primitives share a common set of processing steps, including *selecting operation keys*, *calculating hash values*, *conducting state transitions*, and *processing state results*. Based on the observation, *Newton* proposes to *decompose query primitives into four basic modules*, each implementing one processing step. Moreover, the modules are designed with runtime reconfigurability, so that *Newton* can use table rules to configure their behaviors and to construct query primitives dynamically. Based on query primitive decomposition and reconfigurable modules, *Newton* achieves good generality with no performance penalty for on-demand data plane queries.

The second challenge is to **provide accurate query results with limited hardware resources**. Data planes have limited resources (*e.g.* stages and SRAM) and might not be able to accommodate multiple queries or even one complex query [19]. Moreover, the accuracy of stateful query primitives (*i.e. reduce* and *distinct*) depends on the amount of available stateful memory. *Newton* proposes several optimizations to alleviate the limitation. First, *Newton* optimizes module layout on data planes to improve resource utilization. Second, *Newton* proposes an effective query compilation algorithm for reducing the query resource consumption. Third, *Newton* introduces *cross-switch query execution* that enables queries to utilize resources across switches with small bandwidth overheads.

The last challenge is to **run network-wide queries in the presence of network dynamics.** The centralized controller should guarantee that monitoring queries are deployed on the forwarding paths of target traffic. Moreover, the controller should timely response to network dynamics, such as failures, which can change the traffic forwarding paths. Whereas, detecting network dynamics itself is a hard problem that can take a long time, let alone determining the affected traffic and the corresponding forwarding paths. The cross-switch query execution in *Newton* further adds up the difficulty, as one query could run in different switches. To overcome this challenge, we propose a resilient query placement algorithm that redundantly places queries into switches along all the possible paths. Meanwhile, the algorithm multiplexes table rules, which ensures that redundant queries do not introduce unacceptable overheads. With the query placement algorithm, *Newton* queries own great resilience to network dynamics with moderate data plane resource consumption.

Newton can be readily deployed on P4-programmable switches. We build a prototype of *Newton* and implement 9 queries upon the prototype. We evaluate *Newton* with real-world packet traces and Tofino switches [11]. Evaluation results shows that *Newton* supports flexible query operations which can be completed within dozens of milliseconds without disturbing packet forwarding. Moreover, *Newton* reduces monitoring overheads by two orders of magnitude when compared with TurboFlow [24] and *Flow [17]. For all the queries, *Newton* reduces module usage by over 42.4% and stage usage by over 69.7%.

2 BACKGROUND AND MOTIVATION

2.1 Background

In this section, we briefly introduce the background on traffic monitoring queries and programmable data planes .

Traffic monitoring intents and queries. Traffic monitoring plays a key role in network operation [5, 29, 30]. There can be various traffic monitoring intents, such as getting the IP addresses of the victim hosts under DDoS attacks and capturing the top-100 flows that contribute to most of the traffic. Existing network monitoring systems support specifying operators' intents as traffic monitoring queries which are composed of various primitives. In this paper, we adopt four widely-used stream processing primitives, including *filter*, *map*, *distinct*, and *reduce*, which are also used by Sonata [19] to develop network monitoring queries.

Programmable data planes [9–11] enable operators to flexibly customize their networks according to ever-changing operation requirements. There are three reconfigurable components on programmable data planes. The first one is *match-action table* whose matching fields and actions can be programmed. The second is *stateful memory (*i.e. *registers and counters)* that stores states and supports line-rate transactional ALU [31]. The last is *control flow* that composes the above two components in a switch pipeline. P4 [14], a domain-specific language, facilitates programming the above components. Resources on programmable data planes are evenly sliced into physical stages, each of which can support a amount of concurrent tables and stateful memory. Programmable data planes have enlightened many perspectives of network researching [32–40], especially for network monitoring [19–22, 41–46].

Reconfigurable data planes. There are two types of data plane reconfigurability. The first type is *non-runtime reconfigurability*, referring to the data plane configurations that cannot be done at runtime. P4 belongs to the first type, and reloading P4 programs needs to reboot running switches and to interrupt normal packet forwarding. The second is *runtime reconfigurability*. Match-action table rules belong to the second type, and operators can update table rules in running switches. Achieving on-demand monitoring queries needs runtime reconfigurability, which has two approaches. The first approach is to develop a proprietary ASIC module dedicated to traffic monitoring operations. The representative technique is BroadScan [47, 48] in Broadcom's BCM56275 chip. The second approach is to leverage general match-action tables to build a traffic monitoring module with runtime reconfigurability. *Newton* explores the second approach for good portability and little coupling with specific ASIC implementations.

2.2 Related Works

Network monitoring on programmable switches. As switches are becoming open and programmable, more and more powerful network monitoring systems are developed for specific purposes. FlowRadar [20] and LossRadar [21] provide network-wide per-flow counters or packet loss statistics. HashPipe [43] is an efficient data structure to detect large flows entirely on data planes. ElasticSketch [41] employs memory-efficient sketches to support rich traffic monitoring metrics, *e.g.* heavy hitters and entropy. HyperSight [46] detect abnormal packet behaviors. BeauCoup [49] supports counting distinct items among related packets. Orthogonal to the above proposals, *Newton* provides a general-purpose system for a variety of traffic monitoring intents.

On-data-plane traffic monitoring queries. Motivated by ever-changing operators' demands on network monitoring, Marple [22] proposes query-based data plane monitoring. Operators can specify their network monitoring intents with a high-level language, then Marple compiles queries into switches pipelines, enabling a language-directed hardware design. Sonata [19] integrates switching ASIC and CPU to jointly realize monitoring queries. Sonata dynamically refines the traffic monitoring scope for better accuracy but still falls short of supporting dynamic query operations, such as installing a new query.

Dynamic traffic monitoring queries. To overcome problems incurred by on-data-plane monitoring queries, *Flow [17] and TurboFlow [24] propose dynamic monitoring queries. Their insight is to export generic data (grouped packet vectors for *Flow and flow records for TurboFlow) from data planes to CPU-based analyzers and to perform query logic on software. However, both of them lead to too much monitoring traffic and require a considerable amount of CPU cores for data analysis when networks scale out. *Newton* explores a different direction, *i.e.* running dynamic monitoring queries directly on data planes. Compared with *Flow and TurboFlow, *Newton* implements query logic on data planes to minimize monitoring overheads.

3 OVERVIEW OF *NEWTON*

Architecture. Figure 1 shows the architecture of *Newton*. First, centralized *Newton controller* compiles queries that are developed with stream processing API. Unlike existing intent-driven traffic monitoring systems, the controller only generates *table rules* for *Newton* modules to implement different queries. On data planes, the switch pipeline could have multiple suites of *Newton modules*.

Workflow. The overall workflow of *Newton* is consistent with the top-down network telemetry paradigm [50]. At the initialization time, operators should add *Newton* module layout into the P4 program (§4.1 and §4.2), and load the P4 program into the switch pipeline. At runtime, operators realize their intents with the query API. Next, based on query rule composition (§4.3), *Newton* controller compiles queries into table rules instead of P4 programs, which differs from Sonata and Marple. Then, *Newton* controller installs query rules in appropriate switches according to resilient rule placement (§5.2). After rule installation, queries can run in switches jointly (§5.1) and export traffic monitoring data according to operators' intents.

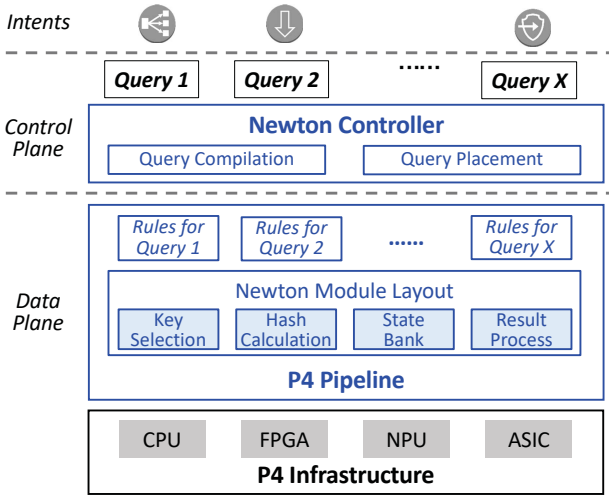


Figure 1: Architecture of *Newton*. The blue boxes identify the contributions of *Newton*.

3.1 Design Goals of *Newton*

Newton is an intent-driven network traffic monitoring system that possesses the four design goals.

On-demand monitoring queries. As new monitoring requirements continuously arise, operators should be able to install new queries into running switches. For example, operators might create queries to zoom on the anomalous traffic [15, 16]. Furthermore, as network status (*e.g.* failures) and traffic characteristics change over time, the running queries should be updated accordingly [17]. Besides, cloud providers could offer network monitoring as services for tenants, such as Amazon CloudWatch [18] and Google Andromeda [51], which requires on-demand query provisioning. Thus, supporting on-demand monitoring queries is essential to ever-increasing network operation demands.

On-data-plane monitoring queries. Monitoring queries can run either on CPU cores or data planes. For on-CPU monitoring queries, monitoring data should be transmitted from data planes to CPU for further analysis. For scalability, on-CPU queries must keep the overheads of data transmission and processing under control when networks scale out. First, some systems reduce overheads by exporting coarse-grained monitoring data from data planes via sampling or filtering [5, 8], but they fail operators' monitoring intents due to lack of fine-grained data. Second, some systems export full data of all packets and reduce overheads via compression [52] or aggregation [17, 24]. Whereas, overheads of *full data exportation* grow proportionally with network sizes and traffic volume. For example, *Flow needs 8 CPU cores to fully process data from a 640Gbps switch [17] . For a network with 1K switches, *Flow requires at least 8,000 CPU cores for monitoring. Therefore, alike Sonata [19] and Marple [22], *Newton* adopts on-data-plane queries and only exports desired data to satisfy operators' intents with high scalability. As presented in §6.1, on-data-plane queries can reduce overheads by *over two orders of magnitude.*

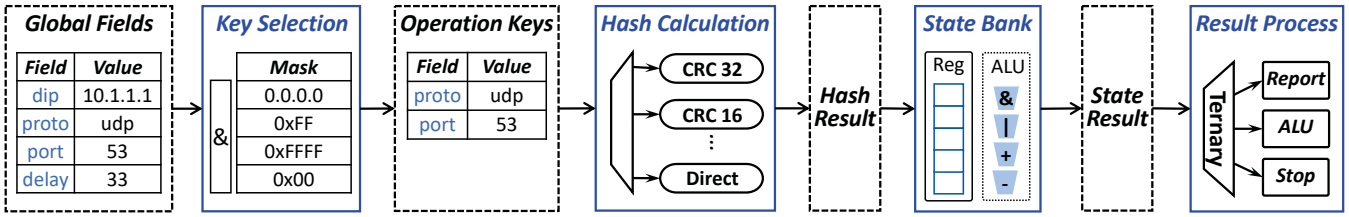


Figure 2: Design of key selection, hash calculation, state bank, and result process.

Resource-efficient monitoring queries. Due to space and power constraints of ASIC, data planes are inevitably limited in resources. As data plane resources determine the capability of supporting complex queries and the precision of monitoring results, optimizing resource usage is of great importance. *Newton* effectively optimizes resource of monitoring queries via the compact module layout, module rule composition, and cross-switch.query execution.

Network-wide monitoring queries. When considering a network, queries must be deployed on forwarding paths of the monitored traffic to provide complete network-wide monitoring results. Existing systems concentrate on device-level monitoring and do not consider network-wide monitoring queries. *Newton* provides an efficient solution for deploying network-wide monitoring queries with strong resilience to network dynamics.

Table 1 summarizes the design goals and corresponding techniques of *Newton*. Next, we will present how to run *Newton* at device level (§4) and how to deploy *Newton* at network-wide (§5) respectively.

4 QUERIES AT DEVICE LEVEL

This section introduces on-demand traffic monitoring queries in *Newton*, and concentrates on the following two problems: *how to make monitoring queries dynamically reconfigurable without disturbing other functions* (§4.1), and *how to fully utilize limited data plane resources during the initialization time (§4.2) and the runtime (§4.3).*

4.1 Query Primitive Decomposition

Newton supports on-demand monitoring queries and makes four query primitives reconfigurable, including *map*, *filter*, *reduce*, and *distinct*. *Newton* focuses on the four primitives because they can satisfy a wide range of significant monitoring intents, as revealed by Sonata [19]. To reconcile the programmability of switching ASIC and the complexity of query primitives, we come up with two ideas.

Design Goals	Techniques	
On-demand On-data-plane	Query primitive decomposition	§4.1
Resource-efficient	Compact module layout	§4.2
	Module rule composition	§4.3
	Cross-switch query execution	§5.1
Network-wide	Resilient module rule placement	§5.2

Table 1: Summary of *Newton* designs.

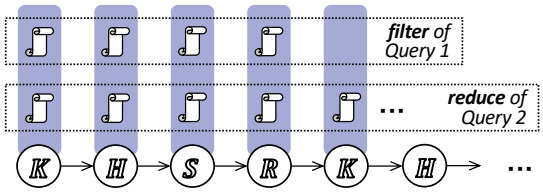


Figure 3: Query primitive rules. Primitives belonging to different queries can share the same module. Note that *reduce* could leverage several module suites to implement a multi-array CM.

- **Modular decomposition of query primitives**. The query primitives share a similar set of processing steps. For example, most primitives select some header fields from a global header fields set, and the stateful primitives do atomic operations on registers. Based on the similarity, we realize the common processing steps with four modules: *field selection*, *hash calculation*, *state bank*, and *result process*. In *Newton*, a module is composed of corresponding P4 tables, table rules, registers and control flow logic.
- **Reconfigurable modules**. Query reconfigurability requires updating query logic via changing table rules instead of modifying P4 programs. Due to the diversity of primitives and primitive parameters, it is non-trivial to achieve the goal. As we decompose query primitives into a set of modules whose logic is simple enough to be configured by rules, we propose to make the modules reconfigurable.

Next, we introduce the four modules shown in Figure 2 and show how they compose the query primitives.

Key selection (𝕂). The first step of most query primitives is to select a set of header fields as the operation keys for subsequent steps. Thus, we design 𝕂 to attain operation keys. 𝕂 takes a list of global fields as input. Due to the hardness of doing list operations (*e.g.* add or remove an element) on programmable data planes, we choose the *bit-mask* action, *i.e.* &, to conceal unneeded fields (*e.g. dip* in Figure 2). Furthermore, the *bit-mask* action could support flexible logic, *e.g.* getting the IP prefix and discretizing the delay.

Hash calculation (ℍ). ℍ conducts hash actions over the operation keys to generate a *hash result*. The reconfigurable elements of ℍ includes the hash algorithms as well as the range of the hash result. Then, the hash result can be used as the register index in the next step. Besides, ℍ supports the *direct* mode. Under this mode, ℍ directly uses a key as the hash result. Whereas, the direct mode comes

at the cost of additional compound actions whose number is proportional to the number of global fields. Fortunately, programmable switches have sufficient VLIW [9] to meet such requirement.

State bank (𝕊). *Newton* adopts the sketch-based implementation of stateful primitives, *e.g.* using Bloom Filter (BF) [53] for *distinct* and Count-Min Sketch (CM) [54] for the sum function of *reduce*. We propose 𝕊 to realize stateful primitives. 𝕊 comprises two parts, *i.e.* a register array and stateful ALUs that can be executed over each register. As shown in Figure 2, *Newton* supports four types of ALU. As BF needs | and CM needs +, the supported ALUs are sufficient. 𝕊 uses the hash result as the register index and supports configuring the ALU to be executed over the registers. Furthermore, with the adjustable range of the hash result, 𝕊 supports flexible register allocation among different queries. 𝕊 outputs the state result of stateful ALUs. Furthermore, 𝕊 can also output the hash result as the state result.

Result process (ℝ). We provide ℝ for processing the state result from 𝕊. ℝ conducts *ternary matching* over the state result and supports three types of operations, as shown in Figure 2. The first one is *report* that uploads the metadata set to analyzers via mirroring. The second one executes some ALUs, *e.g.* + and -, over the result and passes the result to the next step. We can flexibly reconfigure the matching range and the actions to be executed.

The above modules can jointly compose query primitives, and Figure 3 shows how to compile query primitives into module rules. Take *filter(pkt.srcUdpPort==53)* as an example. First, *Newton* configures 𝕂 to select the source UDP port (as shown in Figure 2). Then, *Newton* uses ℍ to set the result to be the UDP port and uses 𝕊 to transmit the hash results to the state result. Next, ℝ matches the state result. If the state result is 53, ℝ continues. Otherwise, ℝ stops the query. Besides, Table 3 (§6.2) presents resource consumption of the four modules and the composed primitives respectively.

Concurrency. As shown in Figure 3, *Newton* multiplexes modules among multiple concurrent queries. Different queries might monitor the same traffic or different traffic. *Newton* chains the queries monitoring the same traffic. For the queries monitoring different traffic, we introduce a *newton_init* table that conducts ternary matching on 5-tuple (including two IP addresses, a protocol type, and two TCP/UDP ports) and TCP control flag to classify and dispatch traffic for concurrent queries. The capacity of *Newton* for supporting concurrent queries is determined by both available data plane resources (including the table size of all modules and the register memory size of 𝕊) and monitoring intents (including the query rule number and register memory consumption).

Expressibility. *Newton* has the same expressibility of Sonata [19] but supports dynamic query operations. To be exact, all Sonata queries can be re-implemented with *Newton*'s decomposed query primitives. For the query primitives that are beyond the capability of data planes, *Newton* shares the same insight with Sonata and lets them run on CPU. Note that query primitive decomposition comes with additional costs, because it consumes more data plane resources, *i.e.* mainly tables and stages. Next, we will demonstrate how to optimize query primitive decomposition during the initialization time and the runtime to render the resource usage of *Newton* comparable to Sonata.

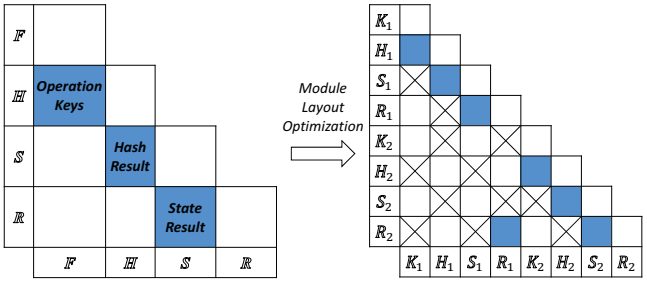


Figure 4: Module dependencies. The blue block denotes there exists write-read dependency between the two modules. The blocks with crosses denote the two modules are placed in the same stage in the compact module layout.

4.2 Compact Module Layout

Newton optimizes the resource used by traffic monitoring queries in two aspects. First, *Newton improves the utilization of data plane resources* by designing a compact module layout while complying with module dependencies. The module layout refers to how to place modules in P4 pipeline and is loaded into P4 pipeline during the initialization time. The module layout of *Newton* should be compact and accommodate as many modules as possible in the switch pipeline. Second, *Newton optimizes the resource consumption of queries* by optimizing the query compilation to reduce the number of used modules during the runtime. Next, this part will introduce the first dimension, while the other one is at §4.3.

Module dependencies. Each *Newton* module takes up one table that can be accommodated by one physical stage, and different modules occupy different types of resources. For example, 𝕊 occupies SRAM mainly, while ℍ occupies Hash functions. Besides, modules have write-read dependencies. For example, ℍ read the operation keys whose values are written by 𝕊. Modules with write-read dependencies are inherently prevented from being placed in the stages [55], and the left part of Table 4 presents module dependencies in *Newton*. Then, to fully utilize all types of resources in each stage, *Newton* should have a *compact module layout*.

Compact module layout. A naïve module layout is to sequentially place one module per stage in P4 pipeline. The naïve module layout yields low resource utilization. For example, 𝕊 could only use 25% registers of P4 pipeline at most.

We design a compact module layout shown in Figure 5 to fully utilize all types of resources on each stage. However, module dependencies hinder placing all the four modules in the same stage, like Stage 4 in Figure 5. To address this problem, we propose to *eliminate module dependencies*. We design two independent metadata sets and two sets of modules that use different metadata sets. For example, red 𝕂 outputs to the hash result of the second set. Thus, modules using the two sets can be placed in the same stage. The right part of Figure 4 shows the dependencies for the two sets of modules. Moreover, red modules and blue modules are not isolated completely. We use a new field named as the *global result* and extend ℝ to match and update the global result. In essence, the compact module layout improves the utilization of other resources at the cost of accommodating an additional metadata set and the global result

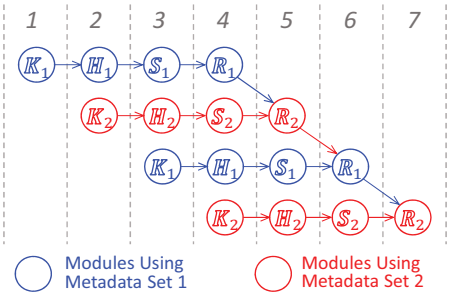


Figure 5: Compact module layout. Arrows denote queries can continue to execute the next module after the current one. A metadata set is composed of operation keys, a hash result, and a state result.

with PHV [9]. Table 3 presents the comparison between the naïve module layout and the compact module layout. The compact module layout enables queries to use more data plane resources than the naïve module layout with the same stage number. Besides, the compact module layout brings new chances for optimizing query compilation (see §4.3).

4.3 Module Rule Composition

Compiling queries in *Newton* involves two steps. The first step is to compile each query primitive into the rules of each module. *Newton* uses one or several suites of modules to realize query primitives, and Figure 3 shows an example for two query primitives. In this section, we focus on the second step, *module rule composition*, which is about *how to place module rules belonging to the same query in the compact module layout.*

An intuitive module composition approach is to simply chain modules according to the primitive sequence in the query. For the example query shown in Figure 6, the intuitive approach occupies up to 20 modules and 20 stages. Thus, the approach comes with a flaw: requiring a large number of tables and stages which programmable switches cannot afford (*e.g.* Tofino has 12 stages per pipeline [56]). To reduce tables and stages consumed by queries, we optimize module rule composition with the following designs.

Opt.1: Replacing front filters with *newton_init*. *Newton* employs *newton_init* (see §4.1) to dispatch traffic for different queries. We observe that many queries first execute *filter*, which uses five tuples and TCP control flag to extract the concerned traffic, which is functionally similar to *newton_init*. To be exact, *newton_init* can equally express the logic of some *filter* primitives in the front of queries. Thus, we can replace filters with *newton_init*. As shown in §6.4, we can perform front filter replacement for 8 out of 9 queries.

Opt.2: Removing unneeded modules. We can optimize module rule composition via removing the two types of unneeded modules. The first one is for unused modules. Some primitives have particular settings that some modules of them are unused. For instance, *map(pkt=>pkt.dip)* only needs $\mathbb{K}$, while the other modules are unused and can be removed without violating the correctness. The second one is for redundant modules. Contiguous primitives might have the same operation keys. Thus, they have the same configuration for $\mathbb{K}$, and $\mathbb{K}$ is redundant, as selected fields can be passed to the subsequent module. For instance, *map(pkt=>pkt.dip)* and *reduce(keys=(pkt.dip),f=sum)* have the same operation keys. We only need one $\mathbb{K}$ for the two primitives.

Opt.3: Composing module rules horizontally and vertically. Until now, *Newton* composes module rules 'horizontally', *i.e.* executing modules sequentially. In this case, *Newton* only uses one module per stage and still needs a large number of stages. Considering *Newton*'s module layout and two separate result sets, we try to compose modules 'vertically' to further optimize the stage number. In the vertical module composition, we implement contiguous primitives with the modules that use different result sets. Thus, modules

Algorithm 1: Module composition algorithm

Input: A module list (L_M) and a primitive list (L_P)
Output: Module composition for each stage (C)

```
L'_M ← [ ];
θ ← none;
Replace the front filters with newton_init if they rely on five
  tuples and TCP flag to classify traffic ;                    /* Opt.1 */
foreach m in L_M do
    if m.is_used() then
        if m is 𝕂 and m.oper_keys ≠ θ then
            θ ← m.oper_keys;
            L'_M.append(m) ;                                   /* Opt.2 */
        else if m is not 𝕂 then
            L'_M.append(m); ;                                  /* Opt.2 */
θ_1, θ_2, p' ← none, none, none ;
foreach p in L_P do
    Remove all m from p.modules when m ∉ L'_M;
    if p' ≠ none and p'.label = 1 then
        if p.oper_keys ≠ θ_2 then
            Restore 𝕂 for p if removed;
        θ_2 ← p.oper_keys;
        p.label ← 2 ;                                          /* Opt.3 */
    else
        if p.oper_keys ≠ θ_1 then
            Restore 𝕂 for p if removed;
        θ_1 ← p.oper_keys;
        p.label ← 1 ;                                          /* Opt.3 */
    p' ← p;
C ← [ ]; s ← 1;
while ∃x ∈ L'_M, x is unassigned do
    S ← [ ];
    foreach p in L_P do
        m ← the first unassigned module of p;
        if !(stage s cannot accommodate m
          or S contains p.modules[0]
          or dependencies of m are satisfied) then
            S.append(m);
            Label m as assigned;
    s ← s + 1;
    C.append(S) ;                                  /* module composition */
```

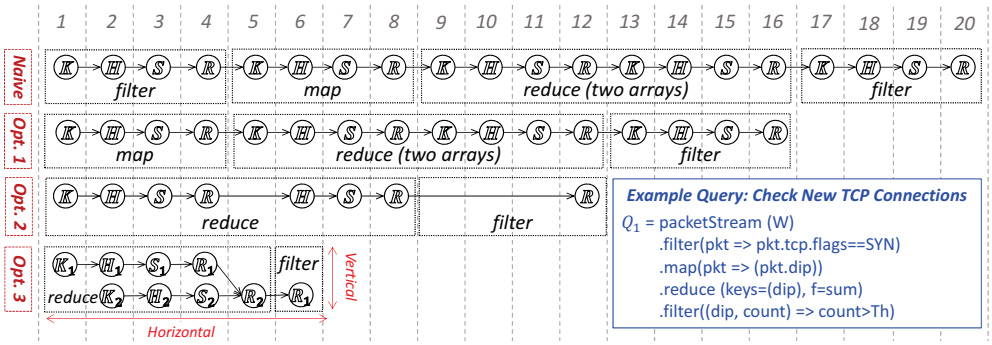


Figure 6: Module rule composition for finding SYN flooding victims [19]. The modules surrounded by rectangles are allocated to accommodate query primitive rules.

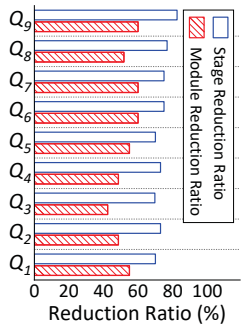


Figure 7: Query optimization ratio.

of contiguous primitives can share physical stages. Furthermore, *Newton* uses $\mathbb{R}$ to merge results from different primitives.

Based on the above ideas, *Newton* provides Algorithm 1 for module rule composition. Figure 7 summarizes the optimization ratios for 9 queries. The optimization ratio represents the percentage of modules and stages that can be reduced from the nïve module composition. Overall, *Newton* can reduce modules by more than 42.4% and stages by more than 69.7%. §6.4 demonstrates a detailed analysis on the effect of module rule composition.

Next, we present an example to demonstrate how a compiled traffic monitoring query works. At first, when a packet enters into the switch pipeline, it matches the *newton_init* table to get the desired query. Assume the packet *pkt* needs to execute Q_1 shown in Figure 6. Then, in the first stage, *pkt* executes $\mathbb{K}_1$ to get operation keys for the following modules whose index is 1. Next, in the second stage, *pkt* executes $\mathbb{H}_1$ and $\mathbb{K}_2$ simultaneously, and *pkt* continues to go through the other modules. Notably, in the fifth stage, $\mathbb{R}_2$ will extracts the minimum value between the global result from $\mathbb{R}_1$ and the hash result from $\mathbb{S}_2$. At last, $\mathbb{R}_1$ in the sixth stage will check whether the global result is larger than *Th*. If so, the switch shall report the operation keys, hash results, state results and the global result to the software analyzer.

5 QUERIES AT NETWORK-WIDE

Queries running in single switches is not enough for practical network monitoring, because operators should correctly deploy the queries on the forwarding paths of monitored traffic and have network-wide monitoring requirements as well, *i.e.* profiling network-wide load imbalance. Thus, *Newton* shall support network-wide monitoring queries. Network-wide monitoring queries should be efficient and resilient, which needs to address two problems. The first one is *how to execute queries in different switches* (§5.1). The second one is *how to place network-wide queries (§5.2).*

5.1 Cross-switch Query Execution

The current practice for network-wide queries is *sole query execution model*, where all queries run independently in switches. However, the limited data plane resources make it hard to implement complex queries within one switch because a query might occupy too many tables and stages that are beyond what switches

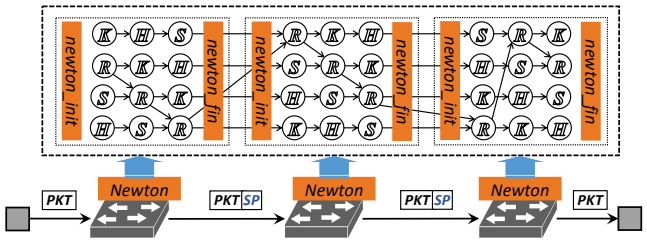


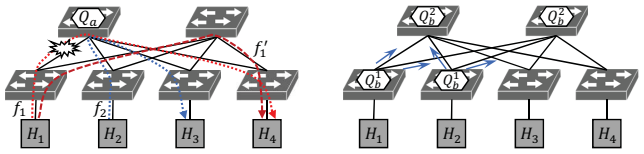
Figure 8: Cross-switch query execution with the result snapshot header.

support. Due to ASIC constraints and performance requirements, it is hardly possible for future programmable switches to be satisfactory. One of the solutions is to push more query logic into the software analyzer [19], which is prone to be scalability bottlenecks.

Cross-switch query execution. *Newton* explores a novel direction that supports query implementation across switches. We propose *cross-switch query execution (CQE)*. CQE makes switches work together to compose a large pipeline of modules, and a query can use all modules in the pipeline. CQE brings two benefits. First, CQE increases the number of available stages, so that more primitives and complex logic are allowed in one query. Second, CQE enables one query to use the memory of many switches, which effectively reduces the impact of data plane resource fragmentation and improves the accuracy of *reduce* and *distinct*.

CQE imposes no change on packet forwarding paths and utilizes switches along the path to realize queries jointly. We employ parallelism concepts of distributed machine learning [57, 58] to demonstrate how *Newton* differs from the existing solutions. On the one hand, existing solutions are the same as *data parallelism* [59] that slices data among workers (switches), while all workers implement the same model. Whereas, *Newton* is identical to *model parallelism* [60] that slices the model among workers, while the workers should sequentially process data. Model parallelism is more appropriate for queries than data parallelism because querying models might be too large to be implemented in switches while switches can process data at fast speed.

Result snapshot protocol. Although CQE seems complex, we manage to design a low-overhead implementation, called *result*



(a) Query placement problem (b) Query placement example

Figure 9: Network-wide query placement. Q_a and Q_b monitor the flow f_1 and f_2. Q_a needs one switch, while Q_b is partitioned into two parts (Q_b^1 and Q_b^2) and run on two switches.

snapshot (SP) protocol. To implement CQE, all we need to do is to transmit results along the forwarding path. Thus, as shown in Figure 8, *Newton piggybacks a snapshot of module execution results in packets*. Then, we use a table, named *newton_fin*, to do the snapshot operation and re-design the parser to decode the SP header and to initialize result sets. Switches will remove the SP header before packets arrive at the destination end-hosts. With SP, a query can be flexibly implemented across multiple switches. As for cost, we only need to reserve 12 bytes for SP and incur less than 1% bandwidth overhead (assume 1500 bytes per packet), when packets need to execute queries cross switches.

For network-wide queries, there remain many realistic problems, one of which is how to place queries at network-wide while being resilient to network failures and forwarding path update. In the next section, we will provide the solution to the above question.

5.2 Resilient Module Rule Placement

When considering network-wide monitoring queries, *how to place queries in multiple switches* turns out be a hard problem. CQE further adds up the difficulty of the query placement problem. As *Newton* does not influence traffic forwarding paths, the queries should be correctly deployed along forwarding paths for all monitored flows. However, computing forwarding paths inherently is NP-hard [61], and it is prohibitively expensive to calculate all forwarding paths between any two hosts according to forwarding rules. Besides, the situation goes worse when the monitored traffic includes a number of flows that have different forwarding paths.

More importantly, forwarding paths are mutable and change over time due to failures, routing protocol updating events, and so on. For example, we deploy Q_a in Figure 9(a) to monitor flow f_1 and f_2. When there is a link failure, f_1 is rerouted to f_1'. In this case, Q_a cannot monitor f_1' and generate the wrong monitoring results. Considering the above factors, query placement is an intractable problem. Thus, our idea is to simplify the problem by *placing queries in switches along all the possible paths without considering forwarding rules*. Figure 9(b) shows a concrete example to place a query with CQE. The idea introduces redundant query rules, because it might install query rules into the switches where there is no monitored traffic. Fortunately, the redundant query rule cost is bounded.

Resilient module rule placement. Based on the above idea, we propose Algorithm 2 that is computationally efficient and yields strong resilience. The algorithm guarantees that the query logic can work correctly with any forwarding path updating event. Furthermore, the algorithm supposes that stages of queries are sequential

Algorithm 2: Module rule placement

Input: Module composition C from Algorithm 1
Input: Query placement $\mathbb{P}$ at each switch
Each switch has N module stages. S_e contains edge switches that are the monitored traffic's first hop;
Slice C into M parts, c_1, ..., c_M, and M is $\lceil |C|/N \rceil$;
$\mathbb{P}$ contains the query slices of each switch;
foreach *s in* S_e **do**
    topo_dfs (*s*, 1);
function *topo_dfs (s, d)*
    if $d \leq M$ **then**
        if c_d *is not in* $\mathbb{P}$ *[s]* **then**
            $\mathbb{P}[s]$.append(c_d);
        Label *s* as discovered;
        foreach *neighbor* s' *of s* **do**
            if s' *is undiscovered* **then**
                topo_dfs (s', d + 1);
        Label *s* as undiscovered;

and each switch has the same number of stages. For example, a query with 10 stages needs 4 3-stage switches to complete, and the query has 4 partitions. Algorithm 2 conducts a depth-first search over the topology. Meanwhile, the algorithm will assign the query partition for each switch, stored by $\mathbb{P}[s]$, and different flows and forwarding paths can reuse $\mathbb{P}[s]$, restricting the number number of module rules. Then, according to $\mathbb{P}[s]$, *Newton* controller installs module rules into each switch. For example, Q_b in Figure 9(b) monitors traffic between H_1 and H_2. *Newton* should place the two partitions of Q_b as shown in the figure: Place Q_b^1 in all the edge switches connected to H_1 and H_2, and place Q_b^2 in all core switches.

Algorithm 2 does not answer an important problem: *what if the query requires more switches than the hop count along the forwarding path*. For example, a query could requires two switches, but the forwarding path of its monitored traffic only has one hop. For this case, *Newton* defers the remaining part of the query to the software analyzer, *e.g.* Spark. The switches will report the current execution status, and the software analyzer will continue executing the query.

6 EVALUATION

Setup. We deploy *Newton* on a testbed composed of three switches and two servers. The switches [62] are equipped with 3.2T Tofino ASIC [11] and Intel Pentium 1.60GHz CPU. The servers are equipped with Intel Xeon 12-core 2.4GHz CPU and Intel XL710 NIC. The testbed topology is alike Figure 8, and all links are 40Gbps. We use two real-world packet traces from CAIDA [63] and MAWI [64] to evaluate *Newton*.

Queries. We use 9 queries (Q_1 to Q_9 in Table 2) from the open-source repository [25]. Figure 6 shows the code of Q_1. These queries involve diverse monitoring intents. Some queries monitor flow characteristics (*e.g.* Q_3), while some detect network attacks (*e.g.* Q_4 and Q_6). The time window of stateful query primitives spans 100ms, and values of *reduce* and *distinct* are evaluated and reset

	Descriptions
Q_1	Monitor new TCP connections
Q_2	Monitor hosts under SSH brute attacks
Q_3	Monitor super spreaders
Q_4	Monitor hosts under port scanning
Q_5	Monitor hosts under UDP DDoS attacks
Q_6	Monitor hosts under SYN flood attacks
Q_7	Monitor completed TCP connections
Q_8	Monitor hosts under Slowloris attacks
Q_9	Monitor hosts that do not create TCP connections after DNS

Table 2: Evaluation queries [25].

every 100ms. Note that not all queries can be fully implemented on data planes [19], and we only consider the data plane parts throughout the evaluation.

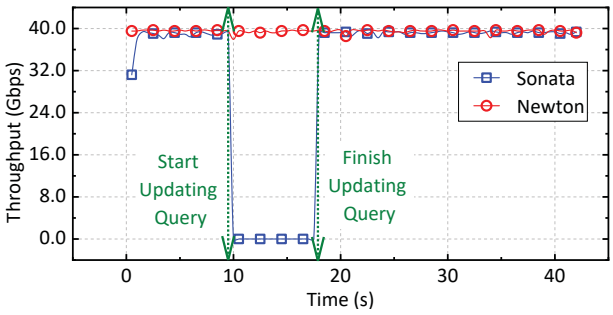
Result summary. In this paper, we evaluate *Newton* in five perspectives, including comparison of *Newton* and existing intent-driven network monitoring systems, data plane resource usage, comparison of cross-switch query execution (CQE) and the sole switch query execution model, query compliation, and query placement. We summarize the evaluation result highlights as follows.

- **Query operations:** *Newton* can install or remove a query within 20ms and does not interrupt normal packet forwarding. (§6.1)
- **Resource efficiency:** *Newton* can provide more data plane resources to traffic monitoring queries via packing more modules in stages. (§6.2)
- **CQE efficiency:** CQE can effectively reduce monitoring overheads and improve the monitoring accuracy for network-wide queries. (§6.3)
- **Query compliation:** *Newton* can reduce consumed modules by 42.4% and consumed stages by 69.7% for query compilation. (§6.4)
- **Network-wide query placement:** Resilient query placement does not bring large overheads and can easily scale out to large networks. (§6.5)

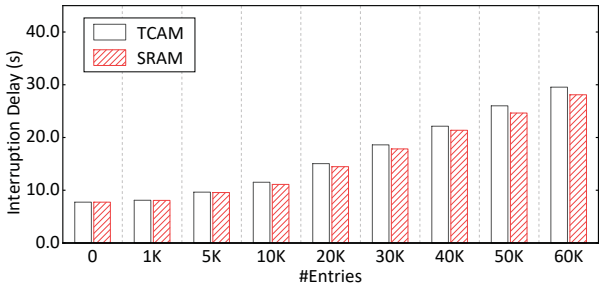
6.1 Benefits of *Newton*

Interruption delay of updating queries in the existing system. First, we evaluate how much time Sonata takes to update running queries. In the experiment, we use switch.p4 [65] to forward packets and change the number of table rules (TCAM or SRAM) required to recover traffic forwarding. Moreover, we use the time when the switch stops forwarding packets during query updating as the interruption delay. As shown in Figure 10(a), after updating queries, Sonata brings about 7.5s outage, *i.e.* the switch throughput reduces to zero, while *Newton* does not affect the switch throughput at all. As given in Figure 10, with the number of table entries increasing, interruption delay of Sonata grows linearly, up to 0.5 minutes with 60K table entries, which degrades network reliability and capacity.

Delay of operating queries in *Newton*. In this part, we show the delay of two *Newton*'s query operations, *i.e.* installing and removing queries. In essence, query installation is to install some table rules, while query removal is to remove some rules. Thus, we measure the delay of manipulating rules for the 9 queries. Figure 11 shows



(a) Throughput degradation



(b) Interruption delay of Sonata

Figure 10: Interruption brought by Sonata.

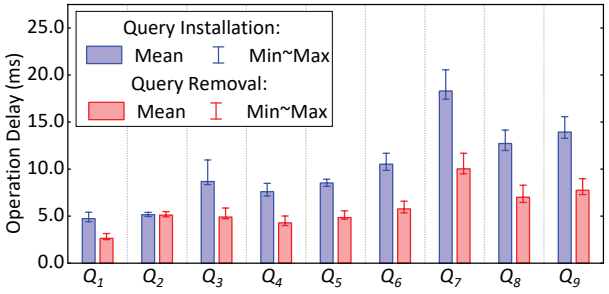


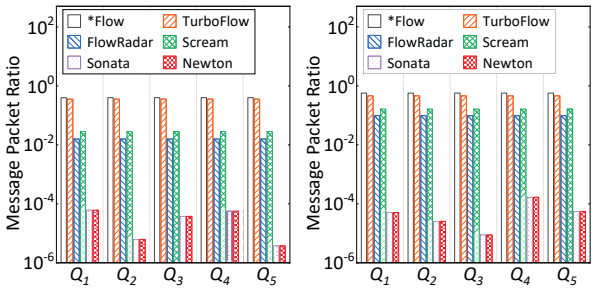
Figure 11: Query installation and removal delay.

the results of the experiments that are repeated for 100 times. Operations of all queries take no more than 20ms. For Q_1, the delay of query installation can even be as low as 5ms.

Monitoring overheads. We compare *Newton* with 5 countermeasures, *i.e.* *Flow [17], FlowRadar [20], TurboFlow [24], Scream [66], and Sonata [19], in terms of monitoring overheads under different queries and packet traces. We use the ratio of the number of monitoring messages against the number of raw packets as the monitoring overheads. As shown in Figure 12, Sonata and *Newton* incur the least monitoring overheads which are two orders of magnitude smaller than the others as a result of accurate data exportation. Furthermore, overheads of TurboFlow and *Flow are proportional to traffic volume. As for FlowRadar whose register

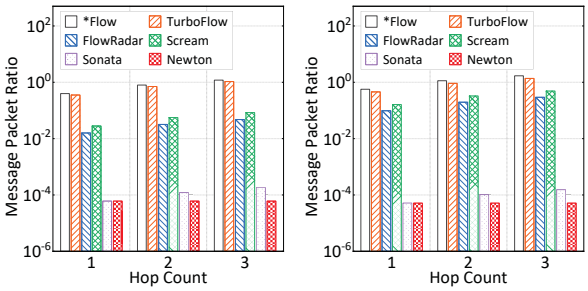
Category	Metric	Crossbar	SRAM	TCAM	VLIW	Hash Bits	SALU	Gateway
Per-stage	Baseline	1.189%	1.232%	1.613%	4.225%	1.222%	1.389%	0.357%
	Compact Module Layout	4.756%	4.929%	6.451%	16.90%	4.889%	5.555%	1.428%
Per-module	Field Selection	0.243%	0.704%	0.0%	3.521%	1.100%	0.0%	1.428%
	Hash Calculation	2.682%	0.352%	0.0%	0.704%	1.589%	0.0%	0.0%
	State Bank	1.219%	3.521%	2.150%	2.112%	2.200%	5.555%	0.0%
	Result Process	0.609%	0.352%	4.301%	10.56%	0.0%	0.0%	0.0%
Per-primitive	filter(pkt.tcp.flags==2)	0.0186%	0.00193%	0.0252%	0.066%	0.0191%	0.217%	0.0056%
	map(pkt=>(pkt.dip))	0.0186%	0.00193%	0.0252%	0.066%	0.0191%	0.217%	0.0056%
	reduce(keys=(pkt.dip),f=sum)	0.0371%	0.0385%	0.0504%	0.132%	0.0382%	0.0434%	0.0112%
	distinct(keys = (pkt.dip, pkt.sip))	0.0557%	0.0578%	0.0756%	0.1980%	0.0573%	0.0651%	0.0167%

Table 3: Hardware resources consumed by *Newton*. The values are normalized by the resource usage of switch.p4



(a) CAIDA (b) MAWI

Figure 12: Monitoring overhead evaluation.



(a) CAIDA (b) MAWI

Figure 13: Network-wide monitoring overhead for Q_1.

array size is 4096, the overhead is about 1%. When networks scale out, FlowRadar consumes a large number of servers [20].

6.2 Data Plane Resource Efficiency

We demonstrate resource utilization of *Newton* at three levels, *i.e.* stage level, module level, and primitive level. The involved resources include table resources (*e.g.* SRAM and TCAM) and control flow resources (*e.g.* gateway [9] for if-else expressions). In the experiments, we configure each module to accommodate 256 rules.

Stage resource utilization. For stage resource utilization, the baseline uses one stage to accommodate only one module (refer to the naïve module layout in §4.2), and we compare the baseline with the compact module layout. As shown by the per-stage category in Table 3, module layout optimization can improve per-stage utilization. The baseline is mostly 25% of the compact module layout because the baseline modules are spread out over 4 stages. In summary, compared with the baseline, the compact module layout can increase data plane resources available to queries.

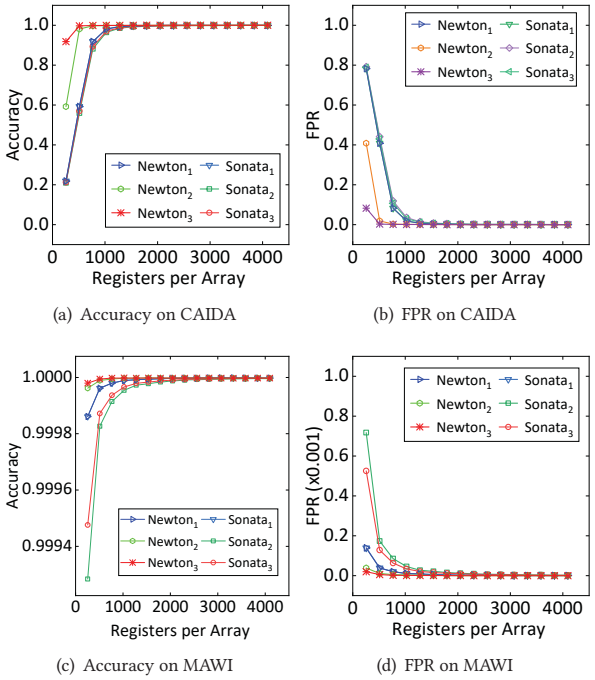
Module resource utilization. As shown by the per-module category in Table 3, each module takes a small amount of data plane resources, which enables us to incorporate tens of modules in one switch. Moreover, *Newton* could reserve considerable resources for other switch functions. Besides, resource utilization is skewed among modules. Thus, the compact module layout that puts four modules in the same stage enables a balanced utilization of per-stage resources.

Primitive resource utilization. As each module supports up to 256 queries, each of the 256 queries can amortize the module resources. We show the resource utilization of the four example primitives by adding up their amortized resources. As shown in Table 3, each primitive occupies a minor amount of data plane resources, and *Newton* enables memory-efficient concurrent queries.

6.3 Benefits of CQE

We evaluate cross-switch query execution (CQE) with regard to scalability and accuracy of Q_1 in the testbed. The testbed topology is simple but representative to show the benefits of CQE. We configure each switch to accommodate three register arrays, each of which has varied numbers of registers. As for *Newton*, we deploy Q_1 in all switches, thus Q_1 can utilize registers among all switches.

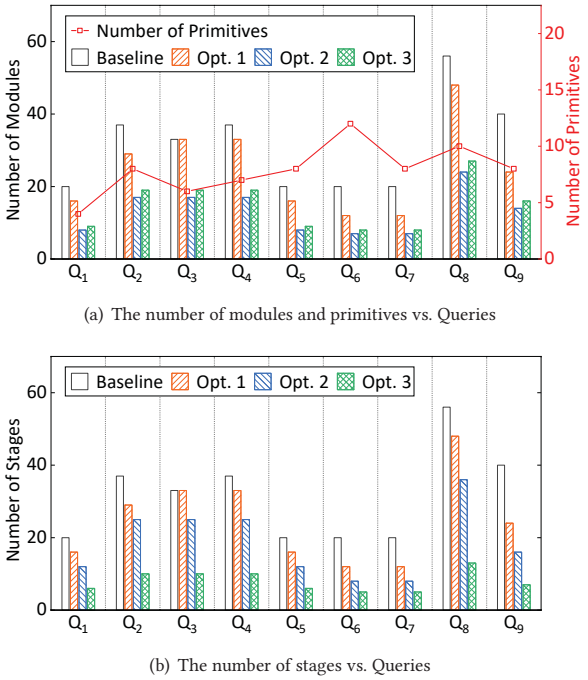
Scalability. In Figure 13, the overheads of all the systems other than *Newton* are linearly proportional to the hop count of packet forwarding paths. Because they treat switches as independent entities and deploy the same querying logic on switches. Thus, switches report monitoring results independently, and the overheads grow linearly with the forwarding path length. Differing from the existing systems, *Newton* treats all switches as a consolidated entity and only reports monitoring data once. Thus, monitoring overheads



(a) Accuracy on CAIDA (b) FPR on CAIDA

(c) Accuracy on MAWI (d) FPR on MAWI

Figure 14: Monitoring accuracy and errors.



(a) The number of modules and primitives vs. Queries

(b) The number of stages vs. Queries

Figure 15: Evaluation of query compilation.

of *Newton* are inherently agnostic to the forwarding path length, yielding good scalability against network sizes.

Accuracy. We measure the accuracy and errors (false positive rates, FPR) of Q_1 on Sonata and *Newton* when changing the number of available registers per array from 256 to 4096. Furthermore, we also change the forwarding path length, *e.g.* $Newton_1$ represents that the hop count is 1 for *Newton*. As shown in Figure 14, *Newton* can improve the accuracy, *e.g.* about 350% improvement over Sonata when there are 256 registers. The above results illustrate that *Newton* can efficiently utilize memory scattered in switches to provide much better accuracy than existing solutions.

6.4 Evaluation of Query Compilation

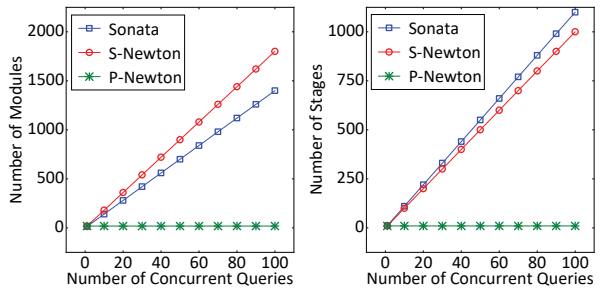
To evaluate query compilation, we use the baseline that places one module in one physical stage as the *baseline*. Then, we apply the query optimization step by step, *i.e. Opt.1*, *Opt.2*, and *Opt.3* (refer to §4.3), over the baseline. Figure 15 displays the number of primitives, modules , and stages for the baseline and each optimization step, and Figure 6 shows overall reduction ratios for all queries.

Module and stage usage. Before diving into query optimization, we analyze the number of primitives and modules. An intuition is that more primitives lead to more modules and stages. However, when it turns to *Newton*, the intuition seems incorrect. For example, Q_6 (12 primitives) consumes much fewer stages and modules than Q_8 (10 primitives). This is because Q_6 has multiple parallel sub-queries, while *Newton* enables resource multiplexing among sub-queries. As shown in Figure 15, resource multiplexing can effectively optimize the usage of modules and stages.

Query compilation optimization. Figure 15 shows that *Newton* can effectively optimize query compilation. The baseline needs over 50 stages for some queries, which is impossible for off-the-shelf programmable switches. With query compilation optimization, *Newton* occupies no more than 10 stages for all the 9 queries. Particularly, for Q_6, *Newton* only needs 5 stages that are even smaller than the number of primitives.

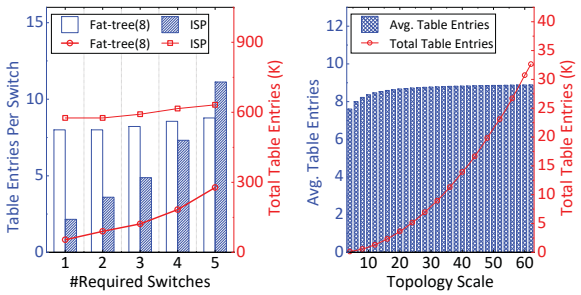
Query-level resource efficiency. Last, we compare *Newton* with Sonata in terms of consumed tables and stages. Figure 15 shows numbers of the logical tables and estimated stages (according to [55]) of 5 queries upon Sonata. The baseline of *Newton* consumes comparable tables to Sonata. Whereas, when applying the query compilation optimization, *Newton* even has lower stage consumption than Sonata.

Concurrent queries. Last, we compare *Newton* with Sonata in terms of resource efficiency when there are multiple concurrent queries. In the experiments, we assume that all the queries have the same logic with Q_4. Furthermore, Sonata sequentially chains these queries. *S-Newton* denotes that the queries in *Newton* monitor the same traffic, and *P-Newton* denotes that the queries in *Newton* monitor the different traffic. As shown in Figure 16, the module number and the stage number of both Sonata and *S-Newton* are linearly proportional to the number of queries. Whereas, *P-Newton* multiplexes resources and yields small resource usage even with 100 queries. In summary, *Newton* can amortize resource consumption among queries, which improves query-level resource efficiency.



(a) Module multiplexing (b) Stage multiplexing

Figure 16: Resource multiplexing over Q_4.



(a) Fat-tree(8) and ISP (b) Require 3 switches

Figure 17: Network-wide query placement of Q_4.

6.5 Evaluation of Query Placement

In this part, we evaluate the network-wide query placement to show its scalability and overheads. In the experiments, we deploy Q_4 in the fat-tree topology when changing the topology scale from tens of switches to thousands of switches, and we demonstrate the scalability of network-wide query placement via the total table entries as well as the average number of entries, *i.e.* entries per switch. We assume that each switch can have 10, 5, 4, 3, 2 stages, then, Q_4 occupying 10 stages and 19 modules (also 19 table entries) requires 1 to 5 switches respectively for cross-switch query execution.

Figure 17(a) shows the number of required table entries when deploying Q_4 in an 8-ary fat-tree topology and a classic backbone topology from the top-tier North America ISP [67] with different numbers of required switches. For the fat-tree topology, *Newton* monitors traffic emitted into the top-of-rack switches from servers, and for the ISP topology, *Newton* monitors the traffic emitted from California. As can be seen, both the average and total table entries increase with the number of required switches, while the growth is more obvious on the ISP topology. Figure 17(b) shows the table entries with different fat-tree topology scales. With the topology scale increasing, the total table entries grow linearly, while the average table entries stabilize to a constant value, indicating that the network-wide query placement comes with acceptable overheads and can easily scale to large networks.

7 DISCUSSION AND LIMITATIONS

Data plane virtualization. There is a line of work, including Hyper4 [26], HyperV [68], and P4Visor [69], which explores virtualizing data planes to make data plane support runtime reconfiguration. They aim at providing on-demand operations of general data plane functions, such as forwarding and firewall. Due to large resource consumption, data plane virtualization can hardly be deployed on hardware P4 targets. Alike data plane virtualization, *Newton* can be view as virtualizing network monitoring queries, but *Newton* presents a practical design that can run on hardware switches with high resource efficiency.

Deployment. *Newton* is implemented as a component for the P4 program and can coexist with other components, such as routing and load balancing. Moreover, the number of stages allocated to *Newton* can be flexibly adjusted based on the resource requirements of other components. Furthermore, *Newton* controller can work as a module of the centralized network controller or be deployed as an independent process. CQE needs the Result Snapshot header to transmit result sets among *Newton*-enabled switches and removes the header before packets entering into servers. Besides, *Newton* supports partial deployment, and CQE only works in adjacent *Newton*-enabled switches.

Limitations. *Newton* has its limitations. First, alike Sonata, *Newton* only supports a few query primitives running on data planes, while the other complex primitives (like *join*) should to be executed by CPU, because the capability of *Newton* is inherently limited by data planes. Second, this paper does not design the solution for scheduling concurrent queries to optimally utilize data plane resources, which is left as the open question for our future work and other interested researchers. Third, as for CQE, states in stateful query primitive could lose in dynamic scenarios where forwarding paths are dynamically altered, and the solo switch query execution model has the same limitation. State loss could lead to ASIC reporting inaccurate monitoring results, but CPU can alleviate the inaccuracy.

8 CONCLUSION

Compared with the state of the arts, *Newton* simultaneously meets the requirements of today's network traffic monitoring systems: on-demand, low overhead, resource efficiency, and network-wide. The main novelty of *Newton* lies in modular decomposition of query primitives, the compact module layout, query compilation optimization, cross-switch query execution, and resilient query placement. The design of *Newton* is low-cost and readily deployable on existing programmable switches. Our evaluation demonstrates that *Newton* brings significant benefits in scalability, dynamics, and accuracy at a cost of acceptable data plane resources and bandwidth consumption.

ACKNOWLEDGEMENT

We thank all reviewers and our shepherd for their constructive comments. We thank Zhaowei Xi, Yunsenxiao Lin, Weibin Meng, Yimin Jiang, and Ya Su, for their insightful suggestions. This research is supported by National Key R&D Program of China (2019YFB1802504 and 2017YFB0801701) and the National Science Foundation of China (61625203, 61832013, and 61872426). Prof. Mingwei Xu is the corresponding author.

REFERENCES

[1] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceddings of NSDI*, 2010.

[2] T. Benson, A. Anand, A. Akella, M. Zhang, T. Benson, A. Anand, A. Akella, M. Zhang, T. Benson, and A. Anand. Microte: Fine grained traffic engineering for datacenters. *Proceedings of CoNEXT*, 2011.

[3] Barefoot Networks. In-network ddos detection. Website. https://goo.gl/WfQUur.

[4] Robert T. Schweller, Ashish Gupta, Elliot Parsons, and Chen Yan. Reversible sketches for efficient and accurate change detection over network data streams. In *Proceeding of IMC*, 2004.

[5] Yibo Zhu, Ben Y. Zhao, Haitao Zheng, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, and Ming Zhang. Packet-level telemetry in large datacenter networks. In *Proceedings of SIGCOMM*, 2015.

[6] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *Proceedings of NSDI*, 2013.

[7] sFlow. sflow. Website, 2019. https://sflow.org/.

[8] Benoit Claise. Cisco systems netflow services export version 9. Website. http://www.rfc-editor.org/rfc/rfc3954.txt.

[9] Pat Bosshart, Glen Gibb, Hun-seok Kim, George Varghese, Nick Mckeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of SIGCOMM*, 2013.

[10] Sharad Chole, Isaac Keslassy, Ariel Orda, Tom Edsall, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, and Shang-Tse Chuang. drmt: Disaggregated programmable switching. In *Proceedings of SIGCOMM*, 2017.

[11] Barefoot Networks. Tofino. Website, 2019. https://www.barefootnetworks.com/products/brief-tofino/.

[12] Broadcom. Broadcom's new trident 4 and jericho 2 switch devices offer programmability at scale. Website. https://www.broadcom.com/blog/trident4-and-jericho2-offer-programmability-at-scale.

[13] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of HotSDN*, 2013.

[14] Pat Bosshart, Glen Gibb, Hun-seok Kim, George Varghese, Nick Mckeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. P4: Programming protocol-independent packet processors. *SIGCOMM CCR*, 44(3), 2014.

[15] Minlan Yu, Lavanya Jose, and R Miao. Software defined traffic measurement with opensketch. In *Proceedings of NSDI*, 2013.

[16] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Dream: Dynamic Resource Allocation for Software-defined Measurement. In *Proceedings of SIGCOMM*, 2015.

[17] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. Scaling Hardware Accelerated Monitoring to Concurrent and Dynamic Queries With * Flow. In *Proceedings of ATC*, 2018.

[18] Amazon. Cloud watch. Website, 2019. https://aws.amazon.com/cloudwatch/.

[19] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven network telemetry. In *Proceedings of SIGCOMM*, 2018.

[20] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *Proceedings of NSDI*, 2016.

[21] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In *Proceedings of CoNEXT*, 2016.

[22] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *Proceedings of SIGCOMM*, 2017.

[23] Apache Spark. Spark: Lightning-fast unified analytics engine. Website. http://spark.apache.org.

[24] John Sonchack, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings of EuroSys*, 2018.

[25] Priceton University. Query-driven streaming network telemetry. Website, 2019. https://github.com/Sonata-Princeton.

[26] David Hancock and Jacobus van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of CoNEXT*, 2016.

[27] Zhang Cheng, Jun Bi, Zhou Yu, Abdul Basit Dogar, and Jianping Wu. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In *Proceedings of ICCCN*, 2017.

[28] Cheng Zhang et al. Mpvisor: A modular programmable data plane hypervisor. In *Proceedings of the Symposium on SDN Research*, SOSR. ACM, 2017.

[29] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of SIGCOMM*, 2015.

[30] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying Datacenter Network Debugging with PathDump. In *Proceedings of OSDI*, 2016.

[31] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM, pages 15–28. ACM, 2016.

[32] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of SOSP*, 2017.

[33] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *Proceedings of FAST*, 2019.

[34] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *Proceedings of NSDI*, 2018.

[35] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of SIGCOMM*, 2017.

[36] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast Connectivity Recovery Entirely in the Data Plane. In *Proceedings of NSDI*, 2019.

[37] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of SOSR*, 2016.

[38] Amedeo Sapio et al. In-network computation is a dumb idea whose time has come. In *Proceedings of HotNets*, 2017.

[39] Jiamin Cao, Jun Bi, Yu Zhou, and Cheng Zhang. Cofilter: A high-performance switch-assisted stateful packet filter. In *Proceedings of SIGCOMM Posters and Demos*, 2018.

[40] Repositories for hypertester with strict anonymity. Website, 2019. https://github.com/hypertester/.

[41] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and Fast Network-wide Measurements. In *Proceedings of SIGCOMM*, 2018.

[42] Qun Huang, Patrick P. C. Lee, and Yungang Bao. Sketchlearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference. In *Proceedings of SIGCOMM*, 2018.

[43] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, SOSR, pages 164–176. ACM, 2017.

[44] Changhoon Kim et al. In-band network telemetry via programmable dataplanes. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR, 2015.

[45] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed Network Monitoring and Debugging with SwitchPointer. In *Proceedings of NSDI*, 2018.

[46] Yu Zhou, Jun Bi, Tong Yang, Kai Gao, Jiamin Cao, Dai Zhang, Yangyang Wang, and Cheng Zhang. Hypersight: Towards scalable, high-coverage, and dynamic network monitoring queries. *IEEE Journal on Selected Areas in Communications*, 2020.

[47] Broadcom. Bcm56275: 244 gb/s programmable multilayer switch. Website. https://docs.broadcom.com/docs/56275-PB.

[48] Ramakrishnan Durairajan and Reza Rejaie ABSTRACT. Cedar: A reconfigurable data plane telemetry system. Website. https://cs.uoregon.edu/directed-research-project/cedar-reconfigurable-data-plane-telemetry-system.

[49] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of SIGCOMM*, 2020.

[50] Minlan Yu. Network Telemetry: Towards A Top-Down Approach. *SIGCOMM CCR*, 49(1), 2019.

[51] Amin Vahdat. Enter the andromeda zone - google cloud platform's latest networking stack. Website, 2019. http://goo.gl/smN6W0.

[52] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proceedings of NSDI*, 2014.

[53] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7), July 1970.

[54] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[55] Lavanya Jose, Lisa Yan, George Varghese, and Nick Mckeown. Compiling packet programs to reconfigurable switches. In *Proceedings of NSDI*, 2015.

[56] Kun Qian, Sai Ma, Mao Miao, Jianyuan Lu, Tong Zhang, Peilong Wang, Chenghao Sun, and Fengyuan Ren. Flexgate: High-performance heterogeneous gateway in data centers. In *Proceedings of APNet*, 2019.

[57] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proceedings of OSDI*, 2014.
[58] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.
[59] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of NIPS*, 2012.
[60] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Proceedings of NIPS*, 2012.
[61] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. *IEEE/ACM ToN*, 22(2), 2014.
[62] Edgecore Networks. Wedge 100bf-32x. Website, 2019. https://bit.ly/2TfkI2p.
[63] The caida ucsd anonymized internet traces - chicago 2014-03-20. Website. https://goo.gl/s35dwY.
[64] WIDE Project. Mawi working group traffic archive. Website. http://mawi.wide.ad.jp/mawi/.
[65] The P4 Language Consortium. Consolidated switch repository. Website, 2019. https://github.com/p4lang/switch.
[66] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. SCREAM: Sketch Resource Allocation for Software-defined Measurement. In *Proceedings of CoNEXT*, 2015.
[67] At&T. Next-generation ip mpls backbone. Website. https://www.att.com/Common/merger/files/pdf/wired-network/Domestic_0C-768_Network.pdf.
[68] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit, and Jianping Wu. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In *Proceedings of ICCCN*, 2017.
[69] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *Proceedings of CoNEXT*, pages 98–111, 2018.