Newton: Intent-Driven Network Traffic Monitoring

Zhaowei Xi, Yu Zhou, Dai Zhang, Kai Gao, Chen Sun, Jiamin Cao, Yangyang Wang, Mingwei Xu, *Member, IEEE*, and Jianping Wu, *Fellow, IEEE*

***Abstract*— Network monitoring systems are designed to fulfill operators' intents and serve as essential tools to modern networks. As a result of rapidly increasing network bandwidth and scale nowadays, network monitors should satisfy on-demand network monitoring for continuously growing traffic volumes. However, existing monitoring systems either cannot satisfy flexible intents on demand or produce significant overheads. In this paper, we present *Newton*, an intent-driven traffic monitor that is able to specify operators' intents with traffic monitoring queries and conduct dynamic and scalable network-wide queries deployment. *Newton* enables operators to customize and modify queries dynamically without interrupting the network workflow. Besides, *Newton* proposes systematic optimizations at device level and network-wide level to reduce resource consumption while deploying queries. *Newton* can combine the resources across switches to deploy complex queries with high resilience to dynamic network status. Evaluations prove that *Newton* is of high flexibility, scalability, and resource efficiency, which demonstrates *Newton* is promising to be deployed in large-scale programmable networks.**

***Index Terms*— Network monitoring, programmable networks.**

I. Introduction

NETWORK monitoring is an essential part of various network operations. For instance, network operators need to 1) capture heavy-hitter flows for traffic engineering to enhance bandwidth utilization [1], [2], 2) spotlight abnormal traffic patterns to detect anomalies [3], [4], 3) collect flow-level or packet-level data to troubleshoot failures [5], [6]. Aforementioned operations bring out various tasks of network monitoring. Besides, the monitoring tasks may change frequently in practical. However, traditional traffic monitors, e.g., sFlow [7] and NetFlow [8] can only provide coarse-grained monitoring data due to substantial overheads, which cannot satisfy the monitoring tasks.

The development of programmable switches [9]–[12] brings new insights for operators to reconfigure monitoring tasks into data planes with domain-specific languages, e.g., NPL [12], POF [13], and P4 [14]. If monitoring tasks are compiled into a P4 program, the job can be realized at line rate in the data planes [14]. Two fundamental principles are vital when designing a practical and efficient traffic monitor.

Traffic monitors should be capable to customize and modify data planes dynamically without interrupting network workflow. The monitoring jobs that the operators hope to deal with change frequently [15]–[18], and supporting a wide range of on-demand tasks in one tool is significant. We use the word *intent* to represent the jobs that are required in the practical network, where the on-demand jobs are not fixed and the change happens all the time. Thus, for traffic monitors, it is essential to consider how to deploy new intents in a practical network. When there is a firmware upgrade or device replacement, the intents can be deployed *statically* while the device is off and network workflow is halted, but this is infrequent. In most cases (e.g., operators may need specific information about certain flows when network monitors detect anomalies or attacks), operators still need to adjust their monitoring intents *dynamically* while they cannot halt the devices or the workflow, which is critical under the consideration of service level agreement (SLA) and cost. Therefore, traffic monitors should support *dynamic customization of monitoring intents* to guarantee no influences on network workflow, including halting/rebooting the switch, changing the packet forwarding logic and leading to a significant loss on performance.

Traffic monitors should achieve low overheads for scalability. The scale and bandwidth of network boom rapidly in recent years. Therefore, monitoring overheads, including bandwidth for data transmission and CPU resources for analysis [19]–[21], should not increase severely when the traffic monitor is deployed in a large commodity network.

Nevertheless, existing traffic monitors cannot obey two aforementioned principles simultaneously. Sonata [19] and Marple [22] export exact information required by monitoring tasks with small overheads, as these works abstract the network as a database and specify monitoring tasks by traffic monitoring queries inspired from SQL and Spark API [23]. The jobs are transformed to monitoring primitives and then

Manuscript received January 24, 2021; revised July 9, 2021 and November 13, 2021; accepted November 14, 2021; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor C. Wu. Date of publication November 30, 2021; date of current version April 18, 2022. This work was supported in part by the National Key Research and Development Program of China under Grant 2019YFB1802504 and Grant 2017YFB0801701; in part by the National Science Foundation of China under Grant 61625203, Grant 61832013, and Grant 61872426; and in part by the Tsinghua University-China Mobile Communications Group Company Ltd., Joint Institute. *(Corresponding authors: Mingwei Xu; Yangyang Wang.)*

Zhaowei Xi, Dai Zhang, Jiamin Cao, Yangyang Wang, Mingwei Xu, and Jianping Wu are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, also with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, and also with the Beijing National Research Center for Information Science and Technology (BNRist), Beijing 100084, China (e-mail: xizw19@mails.tsinghua.edu.cn; zhangd15@mails.tsinghua.edu.cn; jammyc97@gmail.com; wangyy@cernet.edu.cn; xumw@tsinghua.edu.cn; jianping@cernet.edu.cn).

Yu Zhou and Chen Sun are with Alibaba Group, Alibaba Infrastructure Service, Hangzhou 311121, China (e-mail: yuzhou.zy@alibaba-inc.com; qichen.sc@alibaba-inc.com).

Kai Gao is with the College of Cybersecurity, Sichuan University, Chengdu 610065, China (e-mail: kaigao@scu.edu.cn).

Digital Object Identifier 10.1109/TNET.2021.3128557

deployed into switches as the P4 program. Consequently, Sonata and Marple use queries to represent the monitoring jobs and modify jobs through building a new P4 program and downloading the program into the data planes, which causes switch reboot and halts the normal workflow. Therefore, Sonata and Marple are capable of a wide range of tasks, but not the task switching in the practical networks, which is the key obstacle to use Sonata and Marple in practical. Meanwhile, Turboflow [24] and *Flow [17] aggregate monitoring data on data planes and execute queries on software analyzers to satisfy monitoring intents, thus providing dynamic operations without bothering network workflow. But the overheads grow rapidly when network scale and traffic volume expand, leading to poor scalability.

We present *Newton*, an intent-driven NEtWork Traffic mONitor that supports dynamic traffic queries efficiently. *Newton* uses a widely-used query-based API, Spark API [23] (alike Sonata) to represent traffic monitoring intents, and a centralized controller compiles the queries to *Newton* modules. On data planes, *Newton* modules can be dynamically reconfigured through table entry updating and the update does not require halting or interrupting the workflow. *Newton* exports precise intent-oriented data with low overheads and high scalability. The design of *Newton* is not trivial due to three challenges.

The first challenge is **supporting dynamic traffic queries with limited programmability in data planes**. Though called programmable, data planes support limited operations to guarantee high performance [14]. Therefore, supporting traffic queries in data planes without compromising performance can be challenging, particularly when we require dynamically updating the traffic queries. Existing data plane virtualization works [25]–[27] may be general solutions to dynamic traffic queries but severely influence performance. *Newton* observes that queries primitives can be formulated as a set of processing steps, which can be summarized as selecting operation keys, calculating hash values, executing state transitions, and processing state results. Therefore, *Newton* deals with the challenge by decomposing query primitives into four basic *Newton* modules. The *Newton* modules are designed to be runtime reconfigurable, thus *Newton* uses table rules to reconfigure the data planes and realizes intents dynamically.

The second challenge is **supporting various and complex intents with limited resources in network devices**. Resources (e.g., stages and SRAM) are significant when considering whether multiple queries or a complex query can be deployed or not. Moreover, the accuracy of query primitives (e.g. *reduce* and *distinct*) is highly relevant to the amount of stateful memory. Consequently, it is vital for traffic monitors to be resource-efficient to alleviate the limitations. Firstly, *Newton* designs a compact module layout on data planes to improve the utilization of stages. Secondly, *Newton* presents a query compilation algorithm to reduce the usage of modules and stages. Thirdly, *Newton* introduces dynamic register allocation to meet the accuracy bound efficiently. Fourthly, *Newton* proposes network-wide query deployment to combine the capability of multiple devices with small bandwidth overheads.

The third challenge is **supporting network-wide deployment with network dynamics**. *Newton* allows a query to be partitioned across different switches to extend the capability of supporting queries. Therefore, traffic monitors need strategies on how to place monitoring queries in the path of target traffic. The strategies should be robust towards network dynamics, which refer to the conditions that changes the forwarding path, e.g., switch fault or link failure. Unfortunately, detecting network dynamics is a hard problem, especially determining the corresponding forwarding paths associated with the network dynamics. Therefore, *Newton* proposes a resilient query placement algorithm by redundantly placing queries into switches. The rule multiplexing ensures the redundancies do not lead to unacceptable overheads. The query placement algorithm promises strong resilience to network dynamics.

Newton can be deployed on P4-programmable switches. We build a prototype of *Newton* and implement 9 queries upon the prototypes. The evaluations of *Newton* with real-world packet traces reveal that *Newton* provides dynamic intent customization without interrupting the normal workflow. Compared with Marple and Sonata which need seconds to statically update queries, query update in *Newton* can be finished in dozens of milliseconds. Compared with TurboFlow and *Flow, *Newton* reduces monitoring overheads by two orders of magnitude. Meanwhile, through compact module layout and rule composition, *Newton* reduces module usage by over 42.4% and stage usage by over 69.7% compared to the naive query deployment.

II. Background and Related Works

A. Background

We give a brief introduction to the background on traffic monitoring queries and the reconfigurability of programmable data planes.

Traffic Monitoring Queries: Traffic monitoring plays a key role in network operations. Various traffic monitoring intents have been proposed and then specified as traffic monitoring queries, which are defined to be composed of various primitives by existing works [19], [22]. In this paper, we adopt four stream processing primitives: *filter*, *map*, *distinct*, and *reduce*, which are also used by Sonata [19] to develop network monitoring queries in data planes. For instance, if operators find that a victim host may be under DDoS attacks and hope to filter and block attack flows, the traffic monitors may design the queries as: 1) use *filter* to select packets whose destination IP is the victim host; 2) use *map* to map packet key (i.e. flow 5-tuples) to value (i.e. 1 as the number of packet); 3) use *reduce* to aggregate the value (i.e., add the register by 1); 4) if the reduce result exceeds the threshold, drop the corresponding packets and report the packet digest. Using the traffic monitoring queries, traffic monitors can transform monitoring intents to processing logic in the network.

Reconfigurability of Programmable Data Planes: Programmable data planes allow operators to customize network logic. The reconfigurability comes from: 1) *match-action table*. Match-action tables are the basic blocks in the architecture of programmable data planes, users can define the matching

fields, matching methods, and actions. The structure of match-action tables needs to be defined in a P4 program, while the parameters are given by table rules at runtime. 2) *stateful memory*. Stateful memory (i.e., registers and counters) can store states and support line-rate transactional ALUs [28]. 3) *control flow*. How to organize the match-action table and the stateful memory in a switch pipeline is called control flow, which can be also designed in a P4 program [14].

Therefore, the reconfigurability of programmable data planes can be divided into two dimensions. The first is *non-runtime reconfigurability*, referring to the data plane configurations that cannot be customized at runtime. Among aforementioned components, configurations that need to be clarified in the P4 program (e.g. the structure of match-action table, transactional ALU of stateful memory, control flow) belong to non-runtime reconfigurability, which means operators cannot change the configurations without interrupting network workflow. The second is *rumtime reconfigurability*. Configurations that are defined by control plane (e.g. match-action table rules, direct access to stateful memory) belong to the runtime reconfigurability. To achieve dynamic traffic monitoring queries without interrupting the normal workflow, traffic monitors can only leverage the runtime reconfigurability to update queries. Though developing a proprietary ASIC module dedicated to traffic monitoring is a feasible way, e.g. BroadScan [29] in Broadcom's BCM56275 chip [30], *Newton* presents a traffic monitoring module design that leverages general match-action tables to achieve good portability.

B. Related Works

Traffic Monitoring on Programmable Data Planes: Many traffic monitors for specific purposes have been proposed as network devices become more and more programmable and powerful. FlowRadar [20] presents a design of network-wide per-flow counters, while LossRadar [21] builds a per-flow loss map following the same routine. HashPipe [31] detects large flows entirely on data planes using an efficient data structure. ElasticSketch [32] supports rich traffic monitoring metrics including heavy hitters and entropy through deploying sketches dynamically. PRECISION [33] uses partial recirculation to find top flows and explores action stacking to reduce the usage of stages. HyperSight [34] detects abnormal packet behaviors through Bloom Filter Queue. BeauCoup [35] can count distinct items among related packets. Compared with above works, *Newton* is designed to be a general-purpose system for various traffic monitoring intents.

On-Data-Plane Traffic Monitoring Queries: Since intents change continuously in commodity networks, works aiming to support general traffic monitoring demands are proposed. Marple [22] presents language-directed hardware design. Operators specify traffic monitoring intents with a high-level language, which parses intents to monitoring queries and then compiles the queries into switch pipelines. Sonata [19] combines CPU with switching ASIC to implement monitoring queries and achieves better accuracy. But both Marple and Sonata falls short of supporting dynamic traffic monitoring queries since they need to download a new P4 program to update monitoring intents, leading to halting the normal workflow. Although Sonata and Marple support general queries instead of specific purposes, it can be hard to enjoy the benefit in a practical network.

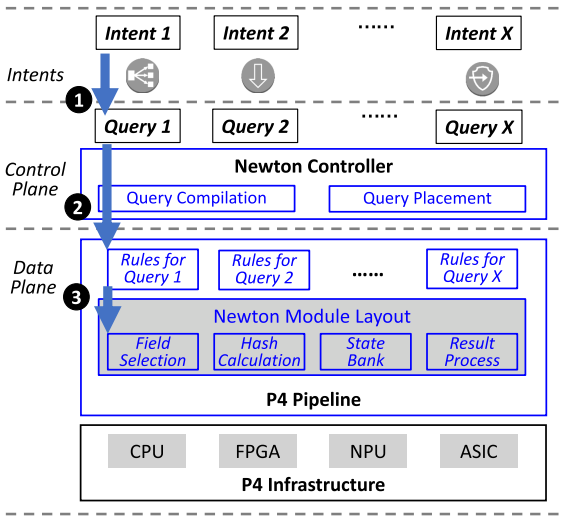


Fig. 1. Architecture of *Newton*. The blue boxes identify the contributions of *Newton*. *Newton* realizes the compact module layout (grey background) at the initialization time while queries are compiled and rules are dispatched at the runtime (white background). Blue arrows represent the workflow: 1) intents are transformed into query primitives, 2) the controller compiles primitives into rule entries, 3) rules are downloaded into the data planes.

Dynamic Traffic Monitoring Queries: To support dynamically updating traffic queries, *Flow [17] and TurboFlow [24] propose dynamic monitoring queries. Both *Flow and TurboFlow export generic data (grouped packet vectors for *Flow and flow records for TurboFlow) and analyze dynamic query logic on software. Consequently, the above works lead to too much monitoring information and significant CPU resource consumption [19]. Works like UnivMon [36] and LEAN [37] explore more efficient mechanisms of dynamic traffic monitoring and make great progress, but the generic data is still much larger compared to intent-oriented data. *Newton* explores a different routine to implement dynamic monitoring queries directly on data planes. Therefore, *Newton* can export intent-oriented data rather than generic data to decrease monitoring overheads.

III. Overview of Newton

A. Architecture and Workflow

Figure 1 shows the architecture of *Newton*. *Newton* is composed of a centralized controller and data planes deployed with *Newton* modules. The controller generates table rules from traffic queries compiled from intents and dispatches rules to each switches. The P4 pipeline realizes the queries and exports monitoring data. Specifically, the workflow of *Newton* is consistent with the top-down network telemetry paradigm, and can be divided into two phases: initialization and runtime.

Initialization: At the initialization time, operators add *Newton* module layout into the P4 program, and compile the P4 program into the switch pipeline. The *Newton* module

is independent with queries, and is able to implement query primitives through table rules.

Runtime: At the runtime, operators' intents are transformed to traffic monitoring queries. Furthermore, queries are compiled into table rules rather than the P4 program, which differs from Sonata and Marple. Therefore, *Newton* allows operators to change intents dynamically. Then *Newton* controller installs table rules in certain switches.

B. Spotlights of Newton

Through designs in IV and V, *Newton* manages to achieve four spotlights that make *Newton* a promising traffic monitor:

Dynamic (Update Intents Without Interrupting the Normal Workflow): It is important for traffic monitors to update monitoring queries dynamically without interrupting the normal workflow, as operators change intents continuously when network circumstances or requirements change. Moreover, the rise of cloud networks gives tenants opportunities to raise dynamic monitoring queries to their traffic [18]. Therefore, we regard dynamic as an essential principle for traffic monitors to be practically deployed. *Newton* proposes query primitive decomposition (IV-A) that decomposes primitives into four modules: *field selection*, *hash calculation*, *state bank*, *result process*, and queries can be implemented on the data planes by downloading table rules into *Newton* modules.

Intent-Oriented (Export Intent-Oriented Data Rather Than General Data): Network virtualization works [25]–[27] or traffic monitors such as Turboflow [24] and *Flow [17] provide dynamic traffic queries, but are of poor scalability. Some works choose coarse-grained sampling or filtering to reduce overheads but lose accuracy and even fail to satisfy certain intents. Other works explore to improve scalability through compression and aggregation, which is still insufficient. For instance, *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. By implementing queries on data planes (IV-A), *Newton* exports intent-oriented data dynamically, rather than general data to overcome the scalability problems.

Efficient (Optimize Resource Usage to Support Multiple and Complex Queries): Traffic monitoring is sensitive to the usage of resources (e.g. stages and stateful memory). On the one hand, the amount of stages serves as the limitation of deploying traffic query logic. On the other hand, the accuracy of queries is highly relevant to memory allocation. *Newton* conducts compact module layout (IV-B) and rule composition (IV-C) to optimize the number of modules and stages, and query memory allocation (IV-D) to enhance the utilization of memory. Besides, for too many queries or too complex queries that exceed the capability of the single switch, *Newton* proposes switch combination for query execution (V-A) to expand the resources.

Resilient (Provide Strong Resilience to Network Dynamics): Network dynamics refer to the conditions that influence the forwarding path of flows, e.g., modify forwarding rules, switch fault and link failure. To coordinate switches to complete traffic queries, network dynamics should be considered. For instance, if the former part of a query is deployed in switch A, and the latter in switch B. When B is down, the traffic is forwarding to other switches and thus the query becomes invalid due to the lack of the latter part. Existing works concentrate on device-level monitoring and leverage single-device capability to avoid coordination among switches. Data are then combined by the controller to analyze network events. *Newton* combines switches to partition traffic queries to further support complex queries, thus *Newton* needs to solve with network dynamics. Consequently, *Newton* provides resilient rule placement (V-B) for network-wide queries with strong resilience to network dynamics.

TABLE I
SUMMARY OF NEWTON SPOTLIGHTS AND CORRESPONDING DESIGNS

Spotlights	Techniques	
Dynamic Intent-oriented	Query primitive decomposition	IV-A
Efficient	Compact module layout (1)	IV-B
1.Stage-efficient	Rule composition (1)	IV-C
2.Memory-efficient	Query memory allocation (2)	IV-D
3.Expansion-efficient	Switch combination (3)	V-A
Resilient	Resilient rule placement	V-B

Table I gives a summary of the spotlights of *Newton* and corresponding techniques. The designs are organized as two dimensions, device level and network level.

IV. QUERIES AT DEVICE LEVEL

This section discusses how *Newton* implements queries dynamically and exports intent-oriented data with efficient usage of resources in the programmable switch. IV-A proposes the structure of *Newton* modules that support dynamic traffic queries and export data specified by the queries. IV-B presents how to pack *Newton* modules into physical stages efficiently at the initialization phase. IV-C introduces how to reduce the number of modules and stages at the runtime phase.

A. Query Primitive Decomposition

Newton focuses on four query primitives proposed by Sonata [19], including *filter*, *map*, *reduce*, and *distinct*, because aforementioned primitives are proved to be capable of a series of significant monitoring intents.

The key insight is that we find query primitives share a similar set of processing steps: 1) select some header fields; 2) generate a key by using fields directly or hash functions; 3) do atomic operations on the stateful memory; 4) further process based on the state result. Therefore, *Newton* realizes the processing steps as four modules: *field selection* ($\mathbb{F}$), *hash calculation* ($\mathbb{H}$), *state bank* ($\mathbb{S}$), and *result process* ($\mathbb{R}$). Each module is a P4 subprogram of match-action tables, registers, and control flow logic. In a stage of P4 pipeline, all tables of four modules are defined, so the stage is able to be applied as any module using table rules. Consequently, we can define a chain of modules to realize query primitives.

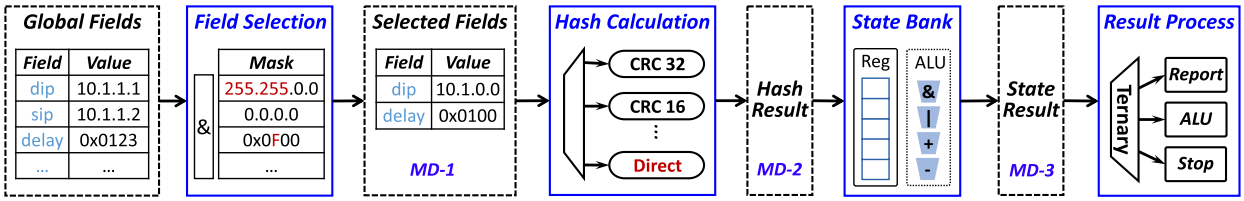


Fig. 2. Design of field selection, hash calculation, state bank, and result process. The MD stands for metadata in P4 pipeline.

Figure 2 shows the structure of four modules. We then discuss the details of *Newton* modules and highlight the dynamically reconfigurable parts.

Field Selection ($\mathbb{F}$): The first step of query primitives is to select packet fields as input for subsequent steps. Due to the difficulty of list operations such as add or remove a field, we perform a bit-mask logic to all optional fields. As shown in Figure 2, operators use rules to define the masks, and then conduct AND with fields to obtain desired fields or bits (e.g. first 16 bits of sip).

Hash Calculation ($\mathbb{H}$): $\mathbb{H}$ represents operations that generate a key by hashing from masked fields. Besides, in many circumstances, the key can be simply a certain field. Therefore, $\mathbb{H}$ supports *hash* mode and *direct* mode. In the *hash* mode, operators can reconfigure the type of hash function and the range of the hash result. In the *direct* mode, $\mathbb{H}$ directly chooses a field as the key. To support the *direct* mode, $\mathbb{H}$ should add actions corresponding to the certain field, thus the number of actions is proportional to the number of global fields. Fortunately, sufficient VLIW [9] in programmable switches makes it possible to cover widely-used fields.

State Bank ($\mathbb{S}$): To support stateful primitives (*reduce* and *distinct*), *Newton* adopts the sketch-based implementation, using Count-Min Sketch (CM) [38] for *reduce* and Bloom Filter (BF) [39] for *distinct*. $\mathbb{S}$ maintains a register array and four types of stateful ALUs to conduct executions to each register. As CM needs $+$ and BF needs $|$, the ALUs are competent for query primitives. Besides, registers can be read and written from the control plane by certain APIs in the switch CPU (e.g., *register_write* in the PTF-test). *Newton* applies the APIs to clear the register when there is a task switching (One monitoring time window of the task finishes) or memory allocation (New queries come and the memory need to adjust). The time cost of writing registers from the control plane is negligible. Therefore, $\mathbb{S}$ supports flexible register allocation among different queries. The output of $\mathbb{S}$ is the state result, which can be the hash result directly as well.

Result Process ($\mathbb{R}$): After obtaining the state result, query primitives may react to the result. While the reactions can be varied, $\mathbb{R}$ realizes three essential behaviors. For other complex behaviors, we believe it is more appropriate to defer them as a series of new primitives. The first is *report*, which uploads the metadata collection to the control plane via mirroring. The second is *ALU*s (e.g. $+$, $-$, and $=$) performing certain calculation on state result. The third is *stop* that stops querying the packet.

With aforementioned modules, *Newton* can decompose query primitives into module rules. Take *filter (pkt.srcIP&0xFFFF0000 == 10.1.0.0)* as an example as shown in Figure 2. Firstly, *Newton* controller configures the mask of source IP as 0xFFFF0000, so $\mathbb{F}$ reserves the highest 16 hits. $\mathbb{H}$ then directs the selected field as the key, and $\mathbb{S}$ does nothing but uses the key as the state result. At last, $\mathbb{R}$ stops the query procedure if the state result is not 10.1.0.0.

One query is a chain of query primitives, thus it can be deployed neutrally in P4 pipeline through query primitive decomposition. Whereas, how to support multiple queries needs extra designs. For queries monitoring the same traffic, *Newton* chains them as a series of queries. Queries monitoring different traffic cannot affect each other and thus need to be parallel in control logic. Therefore, *Newton* introduces the *newton_init* table which conducts ternary matching on 5-tuples to classify traffic at the front of all queries. The traffic is then dispatched to corresponding queries. As for the capacity, it is decided by data plane resources (*how much to offer*, e.g. table size of all modules and register memory size of $\mathbb{S}$ cannot exceed the limitation.) and monitoring intents (*how much to require*, e.g. the number of rules and consumption of registers).

All Sonata [19] queries can be re-implemented with decomposed query primitives. For queries beyond the capability of data planes (e.g., *join*), *Newton* shares the same idea to bypass them to run on CPUs. Though *Newton* owns the same expressibility of Sonata theoretically, it is our responsibility to remind that query primitive decomposition comes with additional costs, as it consumes more data plane resources. Therefore, we present how to optimize query primitive decomposition during the initialization time (IV-B) and the runtime (IV-C) to make the resource usage of *Newton* comparable to Sonata.

B. Compact Module Layout

The stage in P4 pipeline is capable of all four modules as revealed above, and thus sustains resources of all modules. Thus, if naively loading one module into one stage at the initialization time, the resource utilization can be low. An intuitive example is that only $\mathbb{S}$ can access stateful memory, thus the utilization of stateful memory is less than the ratio of $\mathbb{S}$.

Therefore, *Newton* presents compact module layout to accommodate as many modules as possible in the switch pipeline. Unfortunately, module accommodation cannot be arbitrary due to the write-read dependencies between certain modules. We discuss the module dependencies and then propose compact module layout.

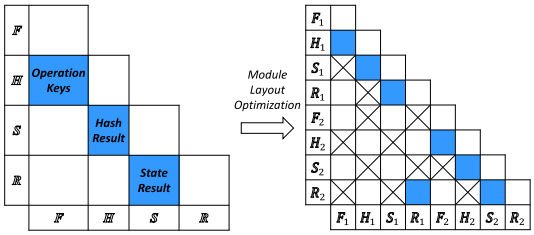


Fig. 3. Module dependencies. The blue blocks refer to write-read dependency between the two modules. The blocks with crosses represent the two modules are placed in the same stage in compact module layout.

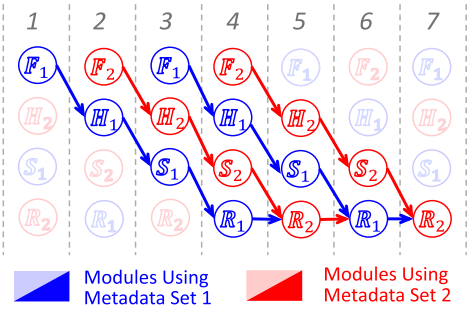


Fig. 4. Compact module layout. Arrows denote queries con continue to execute the next module after the current one. A metadata set is composed of the selected fields, the hash result, and the state result.

Module Dependencies: Each type of module takes up one table and different types of resources. But modules may have write-read dependencies due to the shared values. For instance, $\mathbb{S}$ reads the key written by $\mathbb{H}$. Modules with write-read dependencies are not possible to be placed in the same stage. The left part of Figure 3 introduces the dependencies. To enhance the utilization of resources in each stage, *Newton* comes up with compact module layout.

Compact Module Layout: Figure 4 introduces the compact module layout. Each column stands for one stage in the switch. four modules as a set of ($\mathbb{F}$, $\mathbb{H}$, $\mathbb{S}$, $\mathbb{R}$) are deployed in the stage. The module consists of several match-action tables and a series of metadata fields, and $\mathbb{S}$ owns a certain number of registers. Once table entries are downloaded, the corresponding modules conduct operations to realize the query.

Due to the dependencies, four modules cannot be deployed in the same place. The key insight of compact module layout is to use two independent metadata sets and two sets of modules that use different metadata sets. For instance, though $\mathbb{F}_1$ and $\mathbb{H}_1$ have the write-read dependency, $\mathbb{H}_2$ can be placed with $\mathbb{F}_1$ because they do not use the same metadata set (selected fields, hashed key, state result). The accommodation of two sets of modules is revealed in the right part of Figure 3. In brief, the stage can be regarded as two types: one allows controller to download rules of ($\mathbb{F}_1$, $\mathbb{S}_1$, $\mathbb{H}_2$, $\mathbb{R}_2$), the other allows rules of ($\mathbb{F}_2$, $\mathbb{S}_2$, $\mathbb{H}_1$, $\mathbb{R}_1$). Moreover, *Newton* uses a new field named *global_result*, and extends $\mathbb{R}$ to match and update the global result. Therefore, the results of primitives using different metadata sets can coordinate together.

Compact module layout improves the utilization of resource usage in each stage by involving an additional metadata set and the global result. Consequently, compact module layout enables more query primitives than naive deployment. Besides, compact module layout brings new insight for optimizing query compilation (opt.3 in IV-C).

C. Module Rule Composition

At the runtime, the compilation of queries in *Newton* can be partitioned into three steps: 1) compile queries to query primitives, which can be done the same as Sonata; 2) compile query primitives to rules of *Newton* modules, which is introduced in IV-A; In this section, we mainly discuss 3) module rule composition, which means placing modules rules into the compact module layout. Module rule composition decides how many modules and stages are required to implement the query, thus it is highly relevant to the efficiency of the resource usage.

A naive idea is to simply use a module suite ($\mathbb{F}$-$\mathbb{H}$-$\mathbb{S}$-$\mathbb{R}$) for a primitive and chain the modules according to the primitive sequence. We discuss an example query in Figure 5. We can learn the naive approach occupies 20 modules and 20 stages, which is wasteful and not affordable (e.g., Tofino has 12 stages per pipeline). To reduce consumption of modules and stages, *Newton* optimizes the rule composition in three dimensions.

Opt.1 (Replacing Front Filters With Newton_Init): As mentioned in IV-A, *Newton* employs *newton_init* to dispatch traffic for different queries. In other words, *newton_init* is a natural *filter* to select flows. We notice that for many queries the most front primitive is *filter*. Therefore, if the front *filter* of the query uses only 5-tuples and TCP control flag to select traffic, it can be functionally replaced by *newton_init*. For other *filter*, Opt.1 conducts nothing, otherwise the primitive before the *filter* will only process traffic filtered by the *filter* after them, which makes severe mistakes. As shown in Figure 11a, we can perform the replacement for 8 out of 9 queries.

Opt.2 (Removing Unneeded Modules): We observe two types of modules are unneeded. The first type is unused modules. Given the specific parameters of primitives, some modules can be unused. For instance, as Figure 5 illustrates, *map(pkt→pkt.dip)* only needs $\mathbb{F}$, thus we can remove other modules without violating the correctness. The second type is redundant modules. Given the former primitives, we may already get the metadata we need thus the modules aiming to get the metadata can be redundant. In Figure 5, the *map* and *reduce* share the same key, thus we only need one $\mathbb{F}$ for the *map* and the two-array *reduce*. For the same reason, we do not need ($\mathbb{F}$, $\mathbb{H}$, $\mathbb{S}$) in latter *filter* to get the dip and count. We only need $\mathbb{R}$ to stop/continue the query based on the count.

Opt.3 (Composing Module Rules into Two Metadata Sets): Remember we propose compact module layout to enhance the utilization of resources, which means we have a two-metadata-set based module layout. Therefore, we do not need to chain all primitives as they are in the queries, and we can compose modules "vertically" as shown in the example. Accordingly, the number of stages are further reduced.

The optimizations of module rule composition are summarized as Algorithm 1. Figure 5 shows the reduction ratio of modules and stages for the 9 queries used in this paper (table II), and VI-C conducts a further analysis on the effect of module rule composition.

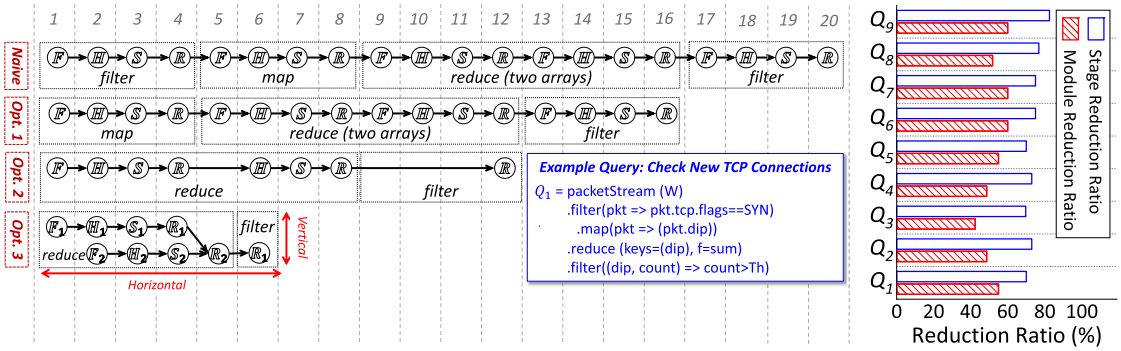


Fig. 5. Module rule composition of finding SYN flooding victims (left part) and the optimization ratio of 9 queries used in this paper (right part).

Algorithm 1 Module Composition Algorithm

Input: A module list (L_M) and a primitive list (L_P)
Output: Module composition for each stage ($\mathcal{C}$)

$L'_M \leftarrow [\,]$;
$\theta \leftarrow none$;
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* $\mathbb{F}$ ***and*** $m.oper\_keys \neq \theta$ **then**
      $\theta \leftarrow m$.oper_keys;
      L'_M.append(m) ; /* Opt.2 */
    else if m *is not* $\mathbb{F}$ **then**
      L'_M.append(m); ; /* Opt.2 */
$\theta_1,\ \theta_2,\ p' \leftarrow none,\ none,\ none$;
foreach p *in* L_P **do**
  Remove all m from $p.modules$ when $m \notin L'_M$;
  if $p' \neq none$ ***and*** $p'.label = 1$ **then**
    if $p.oper\_keys \neq \theta_2$ **then**
      Restore $\mathbb{F}$ for p if removed;
    $\theta_2 \leftarrow p.oper\_keys$;
    $p.label \leftarrow 2$; /* Opt.3 */
  else
    if $p.oper\_keys \neq \theta_1$ **then**
      Restore $\mathbb{F}$ for p if removed;
    $\theta_1 \leftarrow p.oper\_keys$;
    $p.label \leftarrow 1$; /* Opt.3 */
  $p' \leftarrow p$;
$\mathcal{C} \leftarrow [\,]$; $s \leftarrow 1$;
while $\exists x \in L'_M,\ x$ *is unassigned* **do**
  $\mathcal{S} \leftarrow [\,]$;
  foreach p *in* L_P **do**
    $m \leftarrow$ the first unassigned module of p;
    if *!(stage s cannot accommodate m*
      or *$\mathcal{S}$ contains $p.modules[0]$*
      or *dependencies of m are satisfied)* **then**
      $\mathcal{S}$.append(m);
      Label m as assigned;
  $s \leftarrow s + 1$;
  $\mathcal{C}$.append($\mathcal{S}$) ; /* module composition */

D. Query Memory Allocation

Besides stages, the stateful memory (e.g. registers) is another vital type of resources that determines the query accuracy. Though the amount of registers in one state bank cannot change without the reconfiguration, *Newton* is able to dynamically adjust the allocation to queries by changing the parameters in *Newton* modules. For instance, if Q_1 uses 512 registers and the user hopes to extend the size to 1024, *Newton* changes the hash range of Q_1 and the memory offset of other queries, and then *Newton* clears the registers from the control plane as mentioned in State bank ($\mathbb{S}$) in IV-A. Therefore, new results of Q_1 will be recorded in more registers and are still isolated with other queries.

Considering the query concurrency and limited registers, we can benefit from dynamic memory allocation to use registers efficiently and enhance the accuracy of queries. Dynamic memory allocation consists of two significant questions: 1) when to reallocate for queries, 2) how to reallocate for queries.

When to Reallocate for Queries: *Newton* collects stateful results to the control plane at a fixed frequency (e.g., 100ms) and estimates the accuracy of each query in the control plane. It is trivial that there is no need to change the allocation if no query suffers low accuracy and we have sufficient remaining registers for new queries. Therefore, *Newton* reallocates registers when the accuracy bound of a certain query is missed or the resource is over-provisioned. *Newton* borrows algorithms of Scream [40] to estimate the accuracy. The algorithms of Scream applies probabilistic bounds on individual counters and tightens the bounds by separating large item errors from small item errors [40]. Scream algorithms can estimate the accuracy without a prior knowledge of monitored traffic. Consequently, the intent-oriented data of *Newton* provides sufficient information for the estimation. Take Count-Min sketch (CM) as an example, *Newton* exports the size of detected heavy hitters at the end of each monitoring window, and counters in *newton_init* record the traffic volume of each query. Combining aforementioned data with the register size and the threshold, *Newton* can estimate CM accuracy according to Equation 5 in Scream [40].

How to Reallocate for Queries: How to reallocate refers to the strategies of deciding new size of registers for queries. Alike Scream, *Newton* allocates registers at the block granularity, while *Newton* does not adopt the multiplicative increase and multiplicative reduce (MM) used by Scream. Register allocation should converge as fast as possible and guarantee stability, because slow convergence compromises query accuracy and unstable register allocation imposes a large

TABLE II
EVALUATION QUERIES [41]

	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

amount of control channel overheads. Therefore, *Newton* adopt aggressive increase and conservative reduce (AC) for register allocation. VI-D demonstrate that *Newton* achieves better convergence and stability than Scream.

Aggressive Increase: Scream chooses to increase allocated registers by a factor of two when the accuracy bound fails. Therefore, the allocation epoch may cross multiple windows, and the convergence rate can be low. *Newton* directly increases allocated registers to the number that can satisfy the accuracy bound according to the accuracy estimation algorithm at the beginning of each window.

Conservative Reduce: Multiplicative reduce leads to unstable accuracy and results in sparse register allocation. Instead, *Newton* chooses more conservative reduce than Scream. *Newton* conducts addictive reduce if the result of subtracting the accuracy bound from the estimated accuracy is larger than the threshold, otherwise *Newton* makes no reduce. Conservative reduce performs a slow reaction to register over-provisioning, but achieves much better stability. In traffic monitoring, guaranteeing accuracy bound is of high priority when considering a memory-accuracy balance. Therefore, *Newton* claims that the slow reaction is reasonable.

V. QUERIES AT NETWORK LEVEL

Though queries may be compiled together and data may be collected together in a centralized controller, existing works run queries independently in switches. The limited resources make it hard to implement some complex queries within one switch. Meanwhile, pushing complex queries to software analyzer can cause scalability problems.

Newton provides a highly decoupled abstraction on queries to support dynamic intent update, which makes it possible for *Newton* to split one query into multiple switches and combine the resources in the switches (V-A.) Implementing queries across multiple switches should be resilient against network dynamics (e.g. failures), thus *Newton* discusses how to place network-wide queries (V-B).

A. Switch Combination for Query Execution

Newton proposes cross-switch execution (CQE) to support query deployment across switches. As shown in Figure 6, CQE makes upstream and downstream switches work together as "a large pipeline" of *Newton* modules. After executing the logic of upstream modules, the switch obtains the result sets

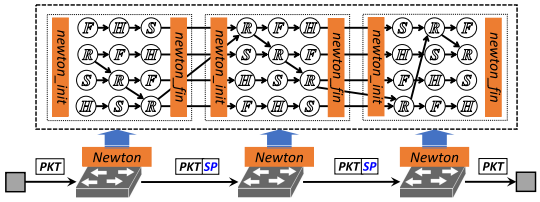


Fig. 6. Cross-switch query execution with the result snapshot header.

and transmit through the result snapshot header (SP) in the packet. The snapshot header is added at the first CQE-enable switch and removed at the last one similar to in-band network telemetry. Using the snapshot header, downstream switches can continue the monitoring process. As a result, *Newton* achieves the detachment of monitoring logic and thus can leverage the resources among different switches to complete the query.

CQE aims to solve to problem of deploying complex queries by chaining the primitives across switches in the forwarding path (Figure 6), not to do the queries in parallel (more discussions in VII).. CQE increases the number of available stages so that one query can have more primitives and complex logic. Besides, we can use CQE to expand memory to improve the accuracy of *reduce* and *distinct* (e.g., perform multi-array *reduce*).

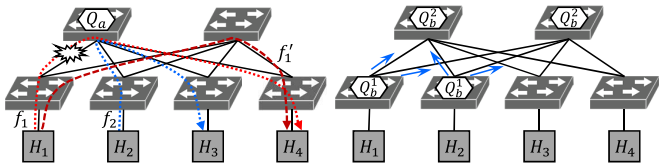
CQE combines switches without imposing any change on packet forwarding paths, which make it hard to place rules. The reason is that CQE needs the information of forwarding paths but finding paths for all flows is a costly NP-hard problem. Besides, the situation can be worse if a flow may own more than one path due to the strategies of load balance or traffic engineering.

More importantly, even if we get all paths at a certain time and the network has no policy that allows multi-path flows, forwarding paths still change due to network dynamics, such as failures, routing protocol updating, and so on. We explain the problem in Figure 7a that Q_a is a query that tends to monitor f_1 and f_2 and can be deployed in one switch. When there is a link failure and f_1 is rerouted to f_1', Q_a cannot monitor f_1' and thus exports wrong data.

Considering the above facts, we propose resilient query placement (Algorithm 2) to tolerate network dynamics without involving unbounded redundant rules. Our idea is to simplify the problem by placing queries in switches along all possible paths without considering forwarding rules. Algorithm 2 supposes the partition of queries are sequential and conducts a depth-first search over the topology from the edge switches. When the switch is discover at k-depth (the maximum is M), *Newton* deploys the kth partition of module rules into the switch. Figure 7b shows how to conduct the query Q_b that needs two switches. Following the logic of algorithm 2, *Newton* places Q_b^1 in two edge switches connected to H_1 and H_2, and places Q_b^2 in two core switches.

B. Resilient Query Placement

Query placement algorithm cannot satisfy the query if the query requires more switches than the hop count along the traffic forwarding path. For this case, *Newton* allows to defer



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

Fig. 7. Network-wide query placement. The left part describes the problem of network dynamics in even the simplest situation(Q_a needs one switch). The right part gives an example of resilient query placement of Q_b which needs to be partitioned into two parts (Q_b^1 and Q_b^2).

Algorithm 2 Resilient Query Placement

Input: Module composition $\mathcal{C}$ 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 $\mathcal{C}$ into M parts, $c_1, \ldots, c_M$, and M is $\lceil |\mathcal{C}|/N \rceil$;
$\mathbb{P}$ 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;

the remaining part to the software analyzer, e.g. Spark. The last-hop switch reports current execution status for the analyzer to continue the query execution.

VI. Evaluation

We develop a prototype of *Newton* on a testbed composed of three switches and two servers, and the P4 program is in [42]. The switches 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 topology is alike Figure 6, and all links are 40Gbps. We use two real-world packet traces from CAIDA [43] and MAWI [44] to evaluate *Newton*. As for the monitoring queries, we use 9 queries (Table II) from the open-source repository [41], and give analysis to Q_1 specifically in Figure 5. Selected queries involve various monitoring intents, such as monitoring flow characteristics(e.g. Q_3), detecting network attacks(e.g. Q_4, Q_5 and Q_6). Values of stateful query primitives, e.g., *reduce* and *distinct* are uploaded and reset every 100ms. Besides pushing the values through digests, Tofino allows users to pull the register from the control plane. Therefore, users can decide the interval of uploading to adjust the overhead of messages. Note not all queries can have a full implementation on data planes, we only consider the evaluation of data planes in this section.

We evaluate *Newton* following the spotlights in III-B to demonstrate that *Newton* manages to achieve a dynamic, intent-oriented and network-wide traffic monitor with high efficiency.

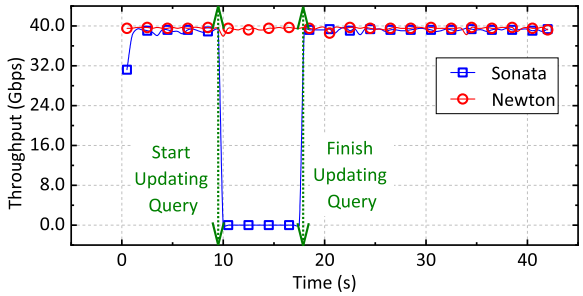
- *Dynamic and Intent-Oriented: Newton* can install or remove a query within 20ms and does not interrupt the normal workflow.
- *Efficiency of Compact Module Layout: Newton* enhances the efficiency of resource usage via compacting more modules in stages.
- *Efficiency of Module Rule Composition: Newton* reduces modules by 42.4% and stages by 69.7% during query compilation.
- *Efficiency of CQE:* CQE reduces monitoring overheads and ameliorate the accuracy of monitoring queries.
- *Network-Wide Query Placement:* Resilient query placement will not bring large overheads and can scale out to large networks.

A. Performance of Newton

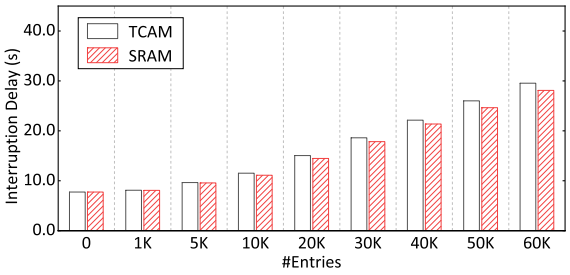
Interruption Delay of Updating Queries in the Existing Traffic Monitor: Firstly, we demonstrate the improvement that can be obtained with dynamic query updating by evaluate the time cost that Sonata takes to an query update, which equals to the time that the switches refuse the normal workflow. We use switch.p4 as the logic of packet forwarding and change the number of table rules that represent the forwarding policies. As shown in Figure 8a, Sonata brings about 7.5s outage when there are no forwarding rules, and the switch throughput reduces to zero. *Newton* does not influence the normal workflow since we do not halt the switch and only update the table rules to deploy new queries. Figure 8b further introduces the interruption of Sonata when changing the number of forwarding rules. The delay grows up to 30s with 60K forwarding rules because Sonata reboots the switch and thus need to reconfigure all forwarding rules into the switch. Therefore, compared with Sonata that involves obvious interruption, *Newton* succeeds to perform dynamic queries that is suitable to apply in a practical network.

Delay of Updating Queries in Newton: Secondly, we evaluate how fast *Newton* can deploy the new query to show the efficiency of *Newton*. We measure the delay of deploying and offloading rules of the 9 queries one by one. The result in Figure 9 proves that no single installation/removal exceeds 20ms. In conclusion, *Newton* can complete the deployment of new queries in negligible time without interrupting the normal workflow.

Monitoring Overheads: Thirdly, we demonstrate that *Newton* is intent-oriented and produce minor overheads close to Sonata. As for overheads, we use the ratio of the number of monitoring messages against the number of packets as the monitoring overheads. We compare *Newton* with 5 baselines: *Flow, FlowRadar, TurboFlow, Scream, and Sonata. We realize our baselines based on the open-source code and configure them as the paper suggested. Overheads of *Flow and TurboFlow is irrelevant to the query. For FlowRadar,



(a) Throughput degradation with no forwarding rules when updating monitoring queries



(b) Interruption delay of Sonata

Fig. 8. Interruption brought by Sonata.

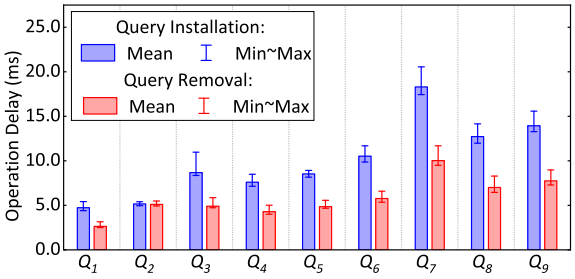
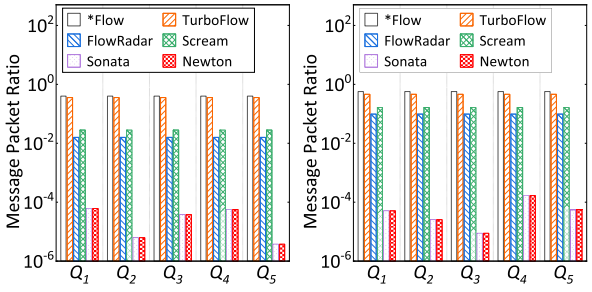
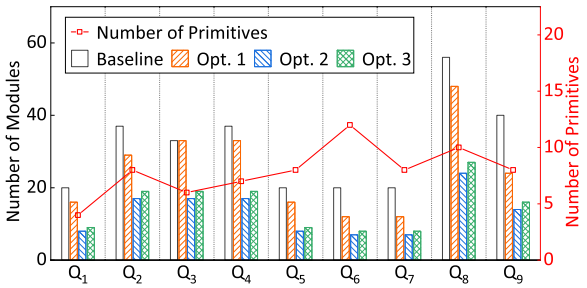


Fig. 9. Installation and removal delay of *Newton*.

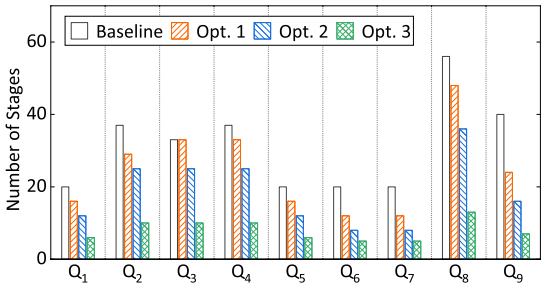


(a) CAIDA (b) MAWI

Fig. 10. Monitoring overheads of Q_1-Q_5.



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



(b) The number of stages vs. Queries

Fig. 11. Evaluation of rule compilation.

the overheads is proportional to the register array size, thus keeping the same since we fix the size as 4096 and do not change the report rate. As shown in Figure 10, on both traffic traces, *Newton* and Sonata lead to minor overheads that are two orders of magnitude smaller than others credit to intent-oriented data exportation.

B. Resource Usage of Newton

We reveal the efficiency of resource utilization of *Newton* from three perspectives: stage level, module level, and primitive level. In the experiments, we configure each module to accommodate 256 rules.

Per-Stage Resource: The baselines refer to the average value of implementing one module in one stage and supporting dynamic traffic query as a chain of $\mathbb{F}$-$\mathbb{H}$-$\mathbb{S}$-$\mathbb{R}$ (the naive module layout discussed in IV-B). Compact module layout manages to accommodate four modules in one stage, thus enhances the utilization of stages in data planes.

Per-Module Resource: Per-module category shows the resource usage of ($\mathbb{F}$, $\mathbb{H}$, $\mathbb{S}$, $\mathbb{R}$). Each module consumes a small amount of resources so we can incorporate tens of modules in one switch while reserving enough resources for other switch functions.

Per-Primitive Resource: Each module can accommodate 256 rules, which means each module supports up to 256 queries. We calculate per-primitive usage through multiplying the number of modules required by the primitive to the average resources dispatched to the query in the module. Each primitive occupies minor resources and thus proves *Newton* enables resource-efficient concurrent queries.

C. Evaluation of Rule Compilation

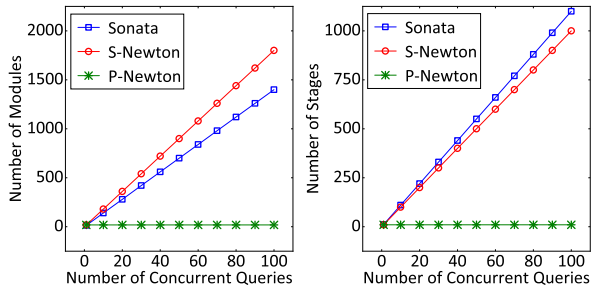
We evaluate the optimization step by step as described in IV-C, *i.e. Opt.1, Opt.2 and Opt.3* against the baseline. Figure 11 illustrates the number of modules, primitives, and stages after each optimization step. The overall reduction ratios are released in Figure 5. Furthermore, we choose Q_4 to test the capability of resource multiplexing of *Newton* (Figure 12).

Module and Stage Usage: Figure 11 reveals the number of modules, primitives, and stages of different queries. Note that due to the parallelism more queries do not mean more stages and modules. For instance, Q_6 has more primitives

TABLE III

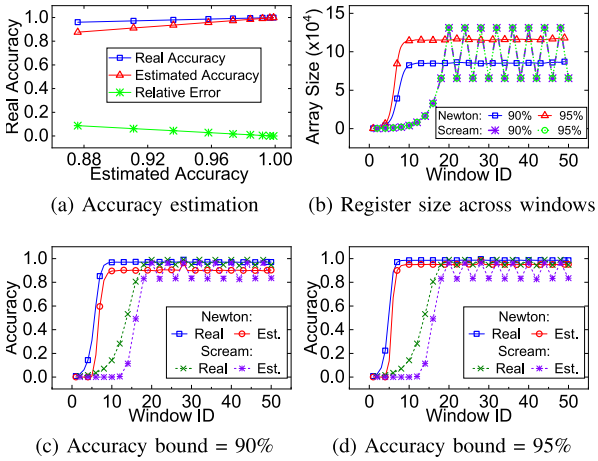
HARDWARE RESOURCES CONSUMED BY *Newton*. THE VALUES ARE NORMALIZED BY THE RESOURCE USAGE OF SWITCH.P4

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%



(a) Module multiplexing (b) Stage multiplexing

Fig. 12. Resource multiplexing over Q_4.



(a) Accuracy estimation (b) Register size across windows

(c) Accuracy bound = 90% (d) Accuracy bound = 95%

Fig. 13. Evaluation of memory allocation over CAIDA.

(12 primitives) than Q_8 (10 primitives) while needs less modules and stages, because Q_6 owns multiple independent sub-queries that can be completed in parallel. As shown in Figure 5, *Newton* reduces over 42.4% modules and over 69.7% stages through the optimization of rule compilation. The baseline needs over 50 stages for some queries, while *Newton* needs no more than 10 stages for all the 9 queries. For instance, for Q_6 *Newton* only requires 5 stages which is even smaller than the number of primitives.

Concurrent Queries: To examine the capability of resource multiplexing, we compare *Newton* with Sonata in terms of module and stage consumption when there are multiple concurrent queries. We assume all the queries own the same logic as Q_4. *S-Newton* refers that the queries monitor the same traffic, and both *Newton* and Sonata cannot implement queries in parallel. Meanwhile, *P-Newton* devotes that the queries monitor different traffic, which means *Newton* can deploy multiple queries through adding corresponding rules. Figure 12 illustrates the results. Sonata and *S-Newton* are linearly proportional to the number of queries, while *P-Newton* multiplexes resources and yields small resource usage. As a result, *Newton* can amortize resource consumption among queries and thus further improve resource efficiency.

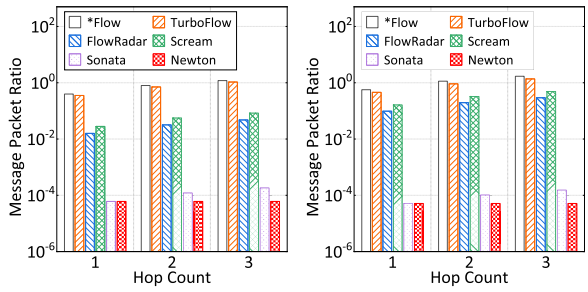
D. Evaluation of Memory Allocation

As for memory allocation, we compare *Newton* with Scream in terms of convergence rates, stability, and memory efficiency. We set 256 as the initial array size when allocating register for the *distinct* primitive. We estimate the accuracy of spotting new elements in the bloom filter. We adopt two accuracy bounds, 90% and 95%, and the allocation epoch of Scream spans two windows whose length is 100ms.

Accuracy Estimation: Firstly, we evaluate the estimation of accuracy in Figure 13a. The relative error grows slowly but stays low (less than 8.5%) as real accuracy decreases. The result proves that *Newton* provides good accuracy estimation, thus *Newton* can leverage the estimated accuracy for the register allocation.

Register Allocation: Figure 13b shows the register allocation under 90% and 95% accuracy bound. The strategy of Scream cannot tell the difference between 90% and 95% due to the coarse granularity of multiplicative increase, which indicates over-provisioning for 90%. Besides, the allocation is not stable and perform as continuously two-phase switching due to the rapidly multiplicative reduce. *Newton* supports fine-grained memory allocation that in both 90% and 95% circumstances *Newton* occupies fewer registers than Scream.

Convergence and Stability: We further discuss the performance of *Newton* and Sonata under 90% and 95% accuracy bound in Figure 13c and 13d. The *Real* stands for the actual accuracy of each window, while the *Est* for the estimated accuracy after each epoch. The result demonstrate that *Newton* has: 1) high convergence efficiency. *Newton* can converge the accuracy for over 100K flows within several windows credit to aggressive increase, and the convergence delay is



(a) CAIDA (b) MAWI

Fig. 14. Network monitoring overheads of Q_1 with different number of hops.

over 50% smaller than Scream. 2) good stability. *Newton* converges to a robust allocation with the presence of traffic changing appreciated to conservative reduces, yielding much better stability and less modifications to switches than Scream.

E. Evaluation of CQE

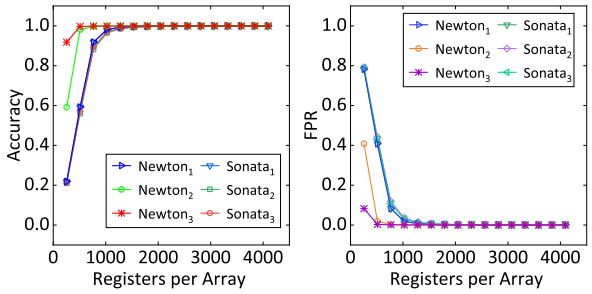
To evaluate the cross-switch query execution (CQE) proposed in V-A, we deploy Q_1 in all switches in the topology. Consequently, the query can utilize stages and registers among all switches.

Monitoring Overheads: We evaluate monitoring overheads through message packet ratio (the same metric as VI-A). As shown in Figure 14, all the baselines are linearly proportional to the hop count since existing works consider queries independently. Thus, switches export data independently. *Newton*, on the contrary, conducts switch combination to realize network-wide deployment, thus switches deploy the query properly and export data once. Therefore, the overheads are irrelevant to the hop of forwarding path, yielding good scalability to the network scale.

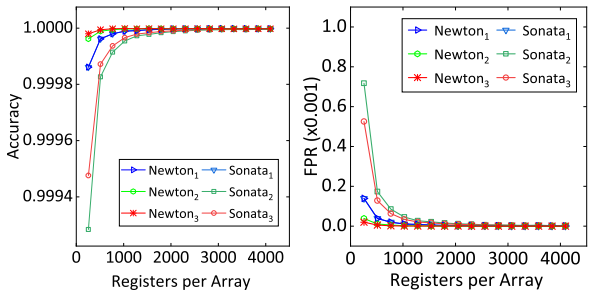
Accuracy: We calculate the accuracy and FPR (false positive rate) of *Newton* and Sonata under the configurations of different number of available registers per array in *reduce* and the length of path. Since both Sonata and *Newton* can leverage the servers to overcome the collisions and shortage of memory, the results only rely on the data plane to show the effect of CQE. The $Newton_1$ refers that the hop count is 1 and represent the performance on the single switch. As illustrated in Figure 15, $Newton_2$ and $Newton_3$ achieve obvious improvement against other configurations, and $Newton_3$ improves the accuracy on CAIDA by about 350% when there are 256 registers. The results prove that *Newton* combines switches efficiently to utilize the resources and obviously improves the accuracy and FPR.

F. Evaluation of Query Placement

We evaluate the total number of table entries in the entire network and the average number of table entries in the single switches to represent the overhead of network-wide query placement of Q_4. We assume that each switch can have 10, 5, 4, 3, 2 stages thus Q_4 that needs 10 stages and 19 modules (also 19 table entries) requires 1 to 5 switches for switch combination. Figure 16 displays the results under different topology, switch capability, and network scales.

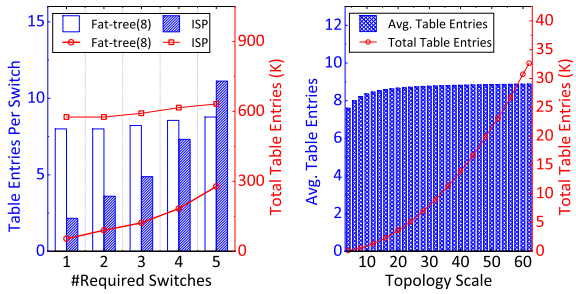


(a) Accuracy on CAIDA (b) FPR on CAIDA



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

Fig. 15. Evaluation of monitoring accuracy and false positive rate (FPR).



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

Fig. 16. Network-wide query placement of Q_4.

Firstly, we evaluate the entries required in an 8-ary fat-tree topology and a backbone topology from the top-tier North America ISP [45] and change the switch required to support Q_4 (Figure 16a). For the fat-tree topology, the traffic is emitted from servers to top-of-rack switches, and for the ISP topology the traffic is emitted from California. As the number of required switches increases, the average and total number of table entries increases due to the resilient placement, and the growth on the ISP topology is more obvious. The average number of table entries in both the fat-tree and ISP topology is moderate and under control.

Secondly, we examine the influence of network scale by changing the scale of the fat-tree topology (Figure 16b). As the topology scale grows, the average number of table entries tends to converge upward to a stable value. In other words, the resilient query placement is of high scalability to the scale of network.

The evaluations demonstrate that resilient query placement comes with acceptable overheads regardless of the topology and the capability of switch. Besides, the strategy can scale to large networks.

VII. Discussion

In this section, we further discuss three dimensions of *Newton* that may be highly concerned: the novelty of *Newton*, how to deploy *Newton*, and the limitations of *Newton*.

Novelty: Newton supports dynamic monitoring query and exports intent-oriented data. Traffic monitors like Marple and Sonata support traffic monitoring through deploying queries into the P4 program, while *Newton* proposes modules to decompose query primitives and leverage the table rules to deploy queries dynamically. *Flow and Turboflow support dynamic monitoring query by implementing intents on software and exporting general data. *Newton* exports intent-oriented data that produces much less overheads. The primitive decomposition in *Newton* is inspired by dataplane virtualization [25], [26], and the philosophy is also applied by other monitoring methods [33]. Works of data plane virtualization can be configured as the traffic monitor, but consume plenty of resources and influence the performance. Besides, the deployment on hardware P4 targets can be difficult. Recent works like BeauCoup [35] can support multiple queries dynamically, but the expressibility is limited (focusing on the *distinct* primitive). Compared with many traffic monitoring methods that focus on special query or primitive, *Newton* is a general-purpose system to support more monitoring intents.

Deployment: Newton can be implemented in the P4 hardware, and can coexist with other dataplane functions, such as routing and load balancing. Operators can adjust the resources of *Newton* in the reconfiguration phase to meet the requirement of other functions. To deploy CQE, switches require the Result Snapshot header to achieve communication of result sets between upstream and downstream switches, which can be realized in a INT-enable device, which have been supported. In conclusion, *Newton* supports partial deployment. A P4-programmable switch can implement *Newton* at device level, while switch combination requires extra support for communication that is practical in nowadays switches.

Limitations: Newton definitely owns its limitations and we hope to discuss the limitations in detail.

Firstly, similar to Sonata, *Newton* only support partial primitives entirely on data planes. Complex primitives, e.g., *join* should be executed by CPU due to the limitation of data planes. If the total queries require memory above the capacity of the devices, *Newton* also needs help from the servers to leave some queries to the CPU, or tolerate an compromising result.

Secondly, CQE and resilient placement duplicate queries in different switches, but cannot support multi-path traffic. To make it explicit, we give an example based on Figure 7b. Q_b need to be parted as (Q_b^1, Q_b^2), and Q_b^2 contains stateful operations, e.g., *reduce*, to find targets exceeding the threshold. If the flow under monitoring can be forwarded to the two switches with Q_b^2 in parallel, possibly the stateful values on each switch do not exceed but the total number exceeds, which give wrong results. Therefore, we emphasize that the switch combination that we propose aims to solve the network dynamic problem (the forwarding path may change, but only one at a certain time), rather than the multi-path problem (more than one forwarding path for the same flow in a certain time).

VIII. Conclusion

We present *Newton* , a traffic monitor that supports dynamic traffic monitoring query and exports intent-oriented data with low overheads, high resource efficiency, and high resilience to network dynamics. The significant insight of *Newton* is decomposing primitives into modules that can be reconfigured through table rules, and *Newton* proposes compact module layout, optimizations on rule compilation, and strategies of memory allocation to make the idea practical and resource-efficient. Besides, *Newton* develops an original strategy of cross-switch query execution with resilient query placement to support combining the resources across switches. Evaluations demonstrate that *Newton* introduces visible improvements on scalability, dynamics and accuracy at a cost of moderate resource and bandwidth consumption.

References

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. NSDI*, Berkeley, CA, USA, 2010, p. 19. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855711.1855730

[2] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Proc. CoNEXT*, New York, NY, USA, 2011, pp. 8:1–8:12, doi: 10.1145/2079296.2079304.

[3] R. Schweller, A. Gupta, E. Parsons, and Y. Chen, "Reversible sketches for efficient and accurate change detection over network data streams," in *Proc. 4th ACM SIGCOMM Conf. Internet Meas.*, New York, NY, USA, 2004, pp. 207–212.

[4] Barefoot Networks. *In-Network DDoS Detection*. [Online]. Available: https://goo.gl/WfQUur

[5] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement.*, Lombard, IL, USA, Apr. 2013, pp. 99–111. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/kazemian

[6] Y. Zhu *et al.*, "Packet-level telemetry in large datacenter networks," in *Proc. 2015 ACM Conf. Special Interest Group Data Commun.*, 2015, pp. 479–491.

[7] Website. *Sflow*. [Online]. Available: https://sflow.org/

[8] B. Claise. *Cisco Systems Netflow Services Export Version 9*. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3954.txt

[9] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, 2013, pp. 99–110, doi: 10.1145/2486001.2486011.

[10] S. Chole *et al.*, "DRMT: Disaggregated programmable switching," in *Proc. Conf. ACM Special Interest Group Data Commun.*, New York, NY, USA, 2017, pp. 1–14, doi: 10.1145/3098822.3098823.

[11] Barefoot Networks. *Barefoot Tofino Switch*. [Online]. Available: https://barefootnetworks.com/technology/

[12] Broadcom. *Broadcom's New Trident 4 and Jericho 2 Switch Devices Offer Programmability at Scale*. [Online]. Available: https://www.broadcom.com/blog/trident4-and-jericho2-offer-programmability-at-scale

[13] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proc. HotSDN*, 2013, pp. 127–132.

[14] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014, doi: 10.1145/2656877.2656890.

[15] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Lombard, IL, USA, Apr. 2013, pp. 29–42.

[16] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "DREAM: Dynamic resource allocation for software-defined measurement," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 419–430, Feb. 2015, doi: 10.1145/2740070.2626291.

[17] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, "Scaling hardware accelerated network monitoring to concurrent and dynamic queries with flow," in *Proc. USENIX Annu. Tech. Conf.*, Boston, MA, USA, Jul. 2018, pp. 823–835. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/sonchack

[18] Amazon. *Cloud Watch*. [Online]. Available: https://aws.amazon.com/cloudwatch/

[19] A. Gupta *et al.*, "Sonata: Query-driven network telemetry," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2018, pp. 357–371.

[20] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers," in *Proc. 13rd USENIX Symp. Netw. Syst. Design Implement.*, 2016, pp. 311–324.

[21] Y. Li, R. Miao, C. Kim, and M. Yu, "LossRadar: Fast detection of lost packets in data center networks," in *Proc. 12nd Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2016, pp. 481–485.

[22] S. Narayana *et al.*, "Language-directed hardware design for network performance monitoring," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 85–98.

[23] A. Spark. *Spark: Lightning-Fast Unified Analytics Engine*. [Online]. Available: http://spark.apache.org

[24] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "TurboFlow: Information rich flow record generation on commodity switches," in *Proc. 13rd EuroSys Conf.*, New York, NY, USA, 2018, pp. 1–16, doi: 10.1145/3190508.3190558.

[25] D. Hancock and J. van der Merwe, "Hyper4: Using P4 to virtualize the programmable data plane," in *Proc. 12nd Int. Conf. Emerg. Netw. Exp. Technol.*, New York, NY, USA, 2016, pp. 35–49, doi: 10.1145/2999572.2999607.

[26] C. Zhang, J. Bi, Y. Zhou, A. Dogar, and J. Wu, "Hyperv: A high performance hypervisor for virtualization of the programmable data plane," in *Proc. ICCCN*, Jul. 2017, pp. 1–9.

[27] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu, "MPVisor: A modular programmable data plane hypervisor," in *Proc. SOSR*, 2017, pp. 179–180.

[28] A. Sivaraman *et al.*, "Packet transactions: High-level programming for line-rate switches," in *Proc. SIGCOMM*, 2016, pp. 15–28.

[29] R. Durairajan and R. Rejaie. *Cedar: A Reconfigurable Data Plane Telemetry System*. [Online]. Available: https://cs.uoregon.edu/directed-research-project/cedar-reconfigurable-data-plane-telemetry-system

[30] Broadcom. *BCM56275: 244 Gb/s Programmable Multilayer Switch*. [Online]. Available: https://docs.broadcom.com/docs/56275-PB

[31] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proc. SOSR*, 2017, pp. 164–176.

[32] T. Yang *et al.*, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. SIGCOMM*, 2018, pp. 561–575.

[33] R. Ben Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Designing heavy-hitter detection algorithms for programmable switches," *IEEE/ACM Trans. Netw.*, vol. 28, no. 3, pp. 1172–1185, Jun. 2020.

[34] Y. Zhou *et al.*, "HyperSight: Towards scalable, high-coverage, and dynamic network monitoring queries," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 6, pp. 1147–1160, Jun. 2020.

[35] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford, "Beaucoup: Answering many network traffic queries, one memory update at a time," in *Proc. SIGCOMM*, 2020, pp. 226–239.

[36] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *Proc. SIGCOMM*, 2016, pp. 101–114.

[37] L. Zaoxing, Z. Samson, R. Ori, and J. Rexford, "Memory-efficient performance monitoring on programmable switches with lean algorithms," in *Proc. APoCS*, 2020, pp. 31–44.

[38] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[39] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970. [Online]. Available: http://doi.acm.org/10.1145/362686.362692

[40] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "SCREAM: Sketch resource allocation for software-defined measurement," in *Proc. 11st ACM Conf. Emerg. Netw. Exp. Technol.*, Dec. 2015, pp. 1–13, doi: 10.1145/2716281.2836099.

[41] Princeton University. *Query-Driven Streaming Network Telemetry*. [Online]. Available: https://github.com/Sonata- Princeton

[42] *Newton*. [Online]. Available: https://gitlab.com/virtualized-network-telemetry/clickq

[43] CAIDA. *The Caida UCSD Anonymized Internet Traces—Chicago 2014-03-20*. [Online]. Available: https://www.caida.org/data/passive/passive_2014_dataset.xml

[44] WIDE Project. *Mawi Working Group Traffic Archive*. [Online]. Available: http://mawi.wide. ad.jp/mawi/

[45] AT&T. *Next-Generation IP MPLS Backbone*. [Online]. Available: https://www.att.com/Common/merger/files/pdf/wired-network/Domestic_0C-768_Network.pdf