Martini: Bridging the Gap between Network Measurement and Control Using Switching ASICs

Paper #69, 15 pages

Abstract

Network management involves two phases: network measurement and traffic control. Many advanced management systems rely on a remote controller to make control decisions. However, this approach incurs a long control loop of a few seconds to minutes. Even if we use the switch-local control plane for decision, it still incurs a loop of 10s of milliseconds, which is unacceptable for many latency-sensitive tasks. In this paper, we propose Martini, a general framework that supports measurement-based timely control. The key idea is to perform measurement, control decision, and control entirely in the switch data plane. This could shorten the control loop of management tasks that require timely control based on only locally measured statistics in the switch. First, Martini introduces a set of primitives to describe management tasks. Next, Martini provides an innovative network-wide task placement mechanism to exploit resources of all network switches to accommodate massive management tasks. Finally, Martini provides a code library and a compiler to support measurement and control on a state-of-the-art switching ASIC. Evaluation results show that Martini can effectively support a wide range of fine-timescale management tasks such as microburst detection and fast load balancing by reducing the control loop from seconds to nanoseconds.

1 Introduction

Network management in modern networks encompasses a range of tasks including anomaly detection [63, 98], attack defense [70, 94], and flow scheduling [2, 13, 31]. Network management normally involves two phases: (1) *measuring* network traffic in real time to collect statistics (*e.g.,* flow sizes, unique flow number), and (2) *controlling* the network in response to detected events (*e.g.,* load balancing elephant flows, explicit congestion notification) [47, 70, 94]. Many efforts have been devoted *separately* to network measurement and control. Researches including OpenSketch [94], Dream [68], UnivMon [63], Trumpet [70], SketchVisor [47], Sonata [39] and more focused on improving the performance, accuracy, and resource efficiency of measurement, while Pyretic [66], FlowTags [30], FlexSwitch [84] and so on focused on efficient declaration and execution of control actions.

Actually, there exists a *gap* between measurement and control, as management tasks need to collect measurement results, *i.e.* statistics of flows or packets, and *decide* which control actions should be executed. Many advanced management systems [60, 63, 70, 77, 94] employ a dedicated controller to connect the measurement and control phases. As shown in Figure 1(a), the measurement module inside a switch collects statistics and reports to the remote controller. The controller then makes decisions and installs control rules back into switches. This approach introduces a *long control loop* between measurement and control. The loop includes the data plane report interval, data transmission and rule installation latency, and the time for the controller to make decisions. Our experiments in §6 shows that this control loop could range from seconds to minutes. Even if we use the switch-local control plane to process events from the data plane, only the message transmission latency of a few milliseconds could be reduced from the control loop.

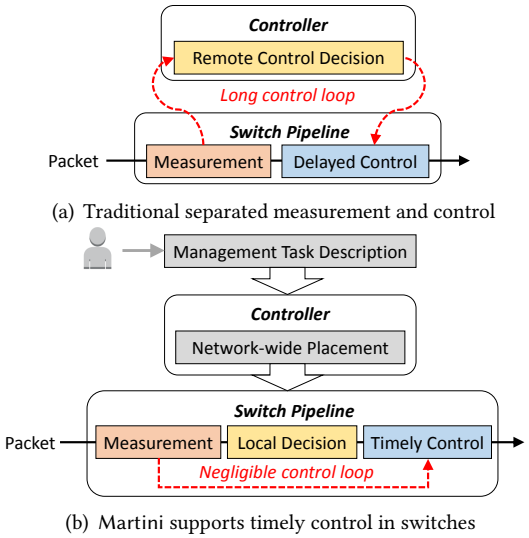


Figure 1. Traditional vs. Martini support for measurement-based network control.

Unfortunately, such a long control loop is unacceptable for many latency-sensitive management tasks, such as congestion control and security tasks. For instance, an advanced congestion control mechanism [36] reveals that *microbursts* cause the majority of congestion and performs fine-grained load balancing to avoid congestion. However, most microbursts last for a few microseconds [12, 36, 97]. If we rely on the controller for control decision, a microburst may have caused congestion before the new rule reaches the switch. This means the control is untimely. Such untimeliness prevents effective support for many management tasks including DNS reflection attack defense [39, 88], superspreader identification and quarantine [11], fast link failure recovery [21], *etc.*

To reduce the control loop, an effective solution is to make control decisions in the *switch data plane*. In this way, entire tasks are offloaded to the switch data plane, which eliminates

the detour through the controller. (1) Some researches propose to enhance the programmability of *OpenFlow switches* to support network tasks [14, 28, 36, 53, 67, 85, 88]. They *customize* OpenFlow switches by adding new types of tables or actions to support *specific tasks*. Offloading new tasks may require redeveloping OpenFlow Switching ASIC, which introduces high cost and long development cycles [18]. (2) Some recent researches [6, 52, 65, 70, 84, 86] exploit *programmable switches* [18] to offload network tasks based on low-level languages such as P4 [17]. (2.1) Researches including [70, 86] propose to offload the *detection of network events* such as heavy hitters to the data plane. However, the detected events still have to be reported to the controller for final control decision, which compromises the effectiveness of above solutions to reduce control loop. (2.2) Researches such as [6, 52, 65] offload specific tasks into switches. However, they are not extensible to other tasks and lack a general framework that provides reusable function modules for network management tasks. FlexSwitch [84] took a first step towards providing a library of reusable modules. However, it only provided building blocks related to resource allocation protocols. In summary, due to the high variety and complexity of management tasks, each of existing solutions is tailored for one specific task and requires system redesign and reprogramming to support new tasks.

Different from above works, our goal is to design a *general framework* that can enable operators to easily describe management tasks based on *high-level primitives*, *translate* task description into low-level programs, and run multiple management tasks *simultaneously* in switches. To achieve this target, we study a series of management tasks and observe that a large portion of tasks require only *local* measurement results in a switch for decision. For example, LocalFlow [82] measures flow sizes, detects burst flows, and performs local load balancing in switches to avoid congestion. We provide more examples in §2.1 and 16 typical tasks in Table 7.

Based on our observation, we propose Martini, a general framework that supports measurement-based *timely* network control using programmable switching ASICs. As shown in Figure 1(b), Martini shortens the control loop by offloading management tasks *entirely to the switch data plane*. The Martini framework includes three components. First, in comparison with languages that can describe either measurement tasks [39, 70, 72] or control actions [9, 30, 66, 84, 87], Martini provides a set of high-level *management task description primitives* for operators to easily describe and assemble measurement, control decision, and control actions. Second, Martini presents an innovative *network-wide task placement* mechanism to exploit resources of all switches in the network to accommodate massive management tasks. Finally, Martini provides a library of measurement and control components and a compiler that can automatically compose them together to generate resource-efficient programs for programmable switching ASICs.

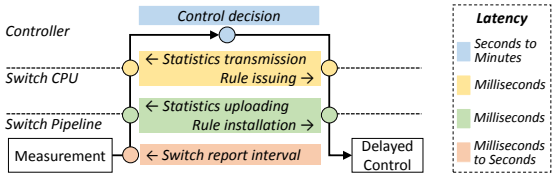


Figure 2. The latency components of the control loop.

In summary, Martini makes the following contributions.

- We identify the motivations and challenges of enabling measurement-based timely traffic control for network management tasks, and propose the Martini framework to shorten the control loop using switching ASICs. (§2)
- We provide a set of *task description primitives* for operators to easily describe the measurement, local control decision, and control actions in management tasks. (§3)
- We design a *network-wide task placement* mechanism that deploys operator-defined management tasks onto all switches in the entire network with high resource efficiency. (§4)
- We introduce the *implementation details* of Martini including a code library of measurement and control components and a compiler that automatically generates codes for all network switches to deploy management tasks. (§5)
- We implement Martini on a state-of-the-art programmable switching ASIC, and test 16 commonly used management tasks. Evaluations show that Martini achieves timely control in all 16 management tasks by reducing the control loop from seconds to nanoseconds. (§6)

2 Motivations and Challenges

Today's network management systems collect statistics in a centralized controller for control decision, which introduces a long control loop. However, modern networks need timely traffic control based on measurement. Above observations motivate us to introduce Martini for measurement-based timely network control. However, we encounter three major challenges in designing Martini including management task description, network-wide task placement, and task implementation on switching ASICs. This section introduces the motivations and challenges in detail.

2.1 Motivations

We elaborate the problems of current management systems and the requirements of modern management tasks, which motivate the design of Martini.

Problem: Long latency between measurement and control in current network management systems. As shown in Figure 2, the current long control loop includes:

- *Switch report interval:* Switches often report statistics *periodically* to the controller. For example, SNMP provides per port counters every few *minutes* [70]. OpenFlow reports flow counters every few *seconds* [2, 28, 70, 76]. Even if a packet that arrives at the beginning of an interval could indicate a network event such as a microburst, the switch

has to report statistics to the controller at the end of this interval, which results in untimely response to events.

- *Statistics uploading and rule installation:* For statistics report, the switch pipeline first moves statistics to the switch CPU, which then communicates with the controller. Returned flow rules will first reach the switch CPU and then be installed into the pipeline. The latency between pipeline and CPU could reach tens of milliseconds and even more under high load [23, 43, 46, 59, 68].
- *Statistics transmission and rule issuing:* Measured statistics and control rules are transmitted between the switch CPU and the controller through OpenFlow [68, 94] or message queues [47]. However, it takes $50\mu s$ to 3 *ms* for an OpenFlow switch to receive a message from the one-hop-away controller [83]. ZeroMQ [44, 47] also needs several milliseconds to finish a round-trip loop. This latency may prevent effective support for flow scheduling or anomaly detection tasks at fine timescales [28, 35].
- *Control decision in the controller:* Many monitoring systems use *sketches* for measurement [47, 63, 69, 94]. A sketch summarizes packets into counters with theoretical guarantees on memory usage and error bounds, making it attractive for measurement in switches with limited memory [39, 68, 69]. However, in order to reduce the communication latency, switches can only report counters without large flow keys to the controller [27, 47, 63, 94]. The controller then exploits techniques such as reversible sketch [80, 94], group testing [63], or sequential hashing [20] to retrieve the flow keys for control decision. Our evaluation in §6.2 shows that this process takes *seconds* to *minutes*, which significantly harms control timeliness.

In summary, traditional management systems rely on the centralized control plane for control decision, which introduces a long control loop. This latency may increase as networks grow in capacity and utilization [70].

Requirement: Timely control based on measurement results in modern networks. Modern networks need to quickly react to measurement results for timely control. We describe a few examples here and list more in Table 7.

- *Microburst detection and fast load balancing:* Some recent works exploit the control plane for congestion control [2, 13, 75]. The control loop for managing one flow may reach 5 seconds due to the controller polling interval and control rule setup latency [28]. Congestion further increases the communication latency between switches and controller. However, the majority of congestion is caused by microbursts with a few μs life cycle [12, 36, 79, 97]. Slow-reacting load balancing may not effectively resolve congestion, since the links appear to be balanced at the coarser timescale of 1*s* [97]. Therefore, we should quickly identify bursty flows that could cause microbursts and ensure timely load balancing within several μs in the switch [82].
- *DNS reflection attack detection and defense:* In a DNS reflection attack, compromised machines send spoofed DNS requests with IP addresses of the target network to DNS resolvers, which respond to the target network and create an attack [38, 78]. Such an attack could grow to Internet-threatening size of 300Gbps [19]. A coarse-grained countermeasure is to identify abnormally large number of responses to a particular IP [3, 39] and report to the controller. However, this method could introduce false positive since a host running tasks such as crawlers could possibly send massive DNS requests and establish a large number of connections. Moreover, if the attack does exist, the attack packets below threshold are still allowed into the network. The slow reaction of the controller and untimely control makes the situation even worse by allowing more attack traffic into the network. A fine-grained countermeasure is to maintain the DNS requests from the network in a switch. If a DNS response without a corresponding request is detected by the switch, it is considered illegal and dropped. The suspicious DNS responses should be handled quickly, since *300Gb* attack traffic can escape and harm the network with 1 second delay before control.
- *Superspreader identification and quarantine:* A superspreader refers to a source IP that simultaneously contacts more than a threshold of unique destination IPs during a time interval [39, 94]. Superspreaders could be responsible for fast worm propagation [91], thus timely identifying and quarantining them is of paramount importance.

Martini: These tasks are beyond the capability of traditional management systems. In this paper, we propose Martini that performs timely control based on local measurement results in switches. In Martini, the latency of the report interval and data transmission is eliminated since the control happens immediately after measurement in the same switch. Furthermore, the latency of retrieving flow keys for control decision is left out, as timely control is precisely executed on the packet that triggers network events. Finally, despite Martini focuses on tasks that control based on *local* measurement results, it can easily be extended to support tasks that require *global* statistics across switches. We will discuss it in §7.

2.2 Design Challenges

We introduce three major challenges in designing Martini.

Management task description: Recent researches have proposed languages for either *measurement* [39, 70, 72] or *control* [9, 64, 66, 84], but have not considered them collectively. However, to support *management* tasks, we are challenged to design new primitives to integrate the contents and connections of measurement, control decision, and control phases. Furthermore, we require high-level primitives to enable code reuse and reduce operator burden. To this end, Martini proposes a set of high-level task description primitives that could easily describe and assemble all phases of many management tasks. We introduce this in §3.

Table 1. Martini measurement primitives and comparison with measurement primitives in Marple [72], Sonata [39].

Primitive	Description	Marple	Sonata
filter (*p*)	Filter packets that satisfy predicate *p*.	✓	✓
update (*i*, *f*, *v*)	Update counter *i* with function *f* & value *v*.	✓	✓
query (*i*)	Query the counter at index *i*.	✓	✓
groupby (*k*, *f*)	Group counters with key *k* and function *f*.	✓	✓
set_window (*t*)	Set the measurement window *t*.	×	✓
set_error_rate (*e*)	Set allowed maximum error rate *e*.	×	×
set_flow_number (*n*)	Set estimated flow number *n*.	×	×
set_flow_rate (*r*)	Set estimated total flow rate *r*.	×	×
get_change ()	Get the change of data across two intervals.	×	×
get_distribution ()	Get the distribution of measurement results.	×	×

Network-wide task placement: Martini aims to accommodate all phases of managements tasks in switches. However, switching ASICs such as Programmable Independent Switch Architecture (PISA) switches [18] have limited resources including pipeline stages, SRAM, and stateful actions [39, 63, 68, 69], which introduces the challenge of accommodating massive management tasks within limited resources. These limitations are likely to remain challenging in the future as there is always a battle for chip die between supporting higher performance and richer resources. To address this challenge, we perform measurement using *sketches* that are highly resource efficient and provide an easily understandable and configurable trade-off between resource usage and accuracy. Our key idea is to place management tasks in the scope of the entire network to exploit the resources of *all network switches*. We propose two techniques including *data partition* and *computation partition* to split a task into several subtasks for placement. Finally, we design a network-wide task placement algorithm that places subtasks on network switches to optimize global resource usage. (§4).

Task implementation on switching ASICs: Implementing management tasks in the switch pipeline is challenging in two aspects. First, we are challenged to support various sketches for measurement and algorithms for control in switching ASICs. In response, Martini provides a *library* of measurement and control components so that operators no longer need to reinvent the wheels. Second, one management task may span across several switches, making it challenging to configure each switch and configure the information exchange among switches. In response, Martini designs a *compiler* to automatically create codes and configurations for all switches by composing components in the code library based on task placement results. (§5).

3 Management Task Description

Martini aims to support management tasks in switches. Therefore, languages that focus solely on measurement [39, 70, 72] or control [9, 64, 66, 84] are not sufficient. We define a set of primitives for operators to easily describe measurement, control decision, and control action. The primitives are designed to be *modular*, *reusable*, and *extensible* so that each phase can be declared separately based on advanced features supported in Martini and composed together easily. Next we present the primitives and some example tasks.

Table 2. Control actions that could be integrated in Martini.

Primitive	Description
drop ()	Drop the packet.
forward (*p*)	Forward the packet out through port *p*.
report_to_controller ()	Report information to the controller.
ecmp ([*p*]) [45, 84]	Per-flow load balancing across a list of ports [*p*].
wcmp ([{*p*, *w*}]) [90, 99]	Per-flow balancing across a list of weighted ports. Each port *p* is assigned a forwarding weight *w*.
spray ([*p*]) [29]	Per-packet load balancing across a list of ports [*p*].
flowlet_ecmp (*i*, [*p*]) [6, 90]	Identify flowlets with inter-arrival time *i*. Per-flowlet load balancing across a list of ports [*p*].
rate_limit (*v*)	Limit the sending rate of a flow to *v*.
ecn () [7]	Tag the ECN bits of a packet once a queue is congested.
qcn (*th*, *p*) [5]	Tag pkts with probability *p* if queue congestion exceeds *th*.
hull (*t*) [8]	Implement phantom queues with threshold *t*.
drill ([*q*]) [36]	Per-packet balancing based on lengths of queues [*q*].
localflow ([q]) [82]	Balance flows into a list of equal cost queues [*q*]
set_queue (*qid*)	Insert a packet into the queue with ID *qid*.
red (*p*) [34]	Drop packets with probability *p* on congestion.
wred ([{*h1*, *h2*, *qid*}])	Set dropping policies for each queue with *qid*. If the queue size < *h1*, do not drop packets. If the queue size > *h2*, drop subsequent packets. Otherwise, drop packets according to the queue size.

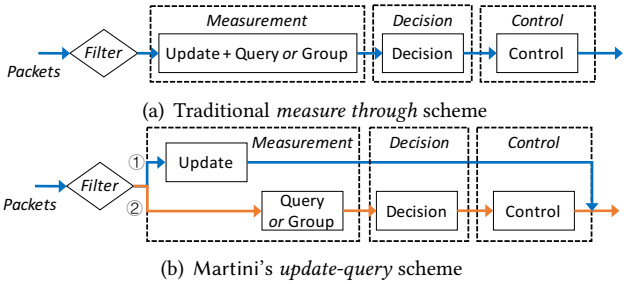


Figure 3. Measurement schemes supported in Martini.

3.1 Martini Primitives

We introduce Martini primitives to describe measurement, control decision, and control phases in a management task.

Measurement: The measurement phase generates statistics that serve as an input for control decision. Different from other primitives for measurement [39, 72], Martini carefully designs measurement primitives with two unique features.

(1) Decoupling statistics updating and querying. We observe that traditional measurement primitives follow a *measure through* scheme when describing measurement tasks. As shown in Figure 3(a), for each packet that satisfies the *predict* in `filter`, the traditional scheme will *update* the measurement record, and *directly use this updated record* for further processing or control decision [39, 70].

However, coupling statistics updating and querying may not be able to support some tasks. For the fine-grained DNS reflection attack defense task (§2.1), we should maintain a *record* of *DNS requests*. For each *DNS response*, we *query* the record and output the existence of the corresponding request. Updating and querying are triggered by different packets, which cannot be supported by the traditional scheme.

To address this challenge, *besides supporting the measure through scheme*, Martini refers to database operations and also supports an *update-query* scheme as shown in Figure 3(b). We list our measurement primitives in Table 1. operators could `filter` one set of packets to `update` the record

and a different set to `query` or `group` the record to generate output statistics. We use a counter array to maintain measurement record. The `update` primitive takes the index i of the array as input, and uses function f to either *add* or set the counter with value v. On the other hand, the `query` primitive returns the single counter value at index i, while the `group` primitive groups the counter array with the key k and function f that may be either *sum* or *count* (`group` is realized using sketches). Finally, we design the `set_window` primitive to configure the measurement window t.

With respect to expressivity, Martini measurement primitives support the *measure through* scheme and can describe all measurement tasks in Sonata [39]. Even though Sonata supports a unique *join* operation, this operation cannot be implemented in programmable switches [39]. Moreover, Martini's *update-query* scheme allows us to easily describe the measurement phase of many management tasks such as DNS reflection attack defense, FTP monitoring, NTP amplification attack defense, *etc.* We present several examples in §3.2.

(2) Resource-aware measurement description. Measurement is often memory intensive [39, 86]. Accurately estimating the resource usage of measurement tasks is important for placement. However, existing measurement primitives focus solely on declaring the functions of measurement tasks. Therefore, operators are heavily burdened to manually estimate resource usage for placement. To relieve operator burden, Martini allows operators to input estimated maximum `error_rate`, `flow_number`, and `flow_rate`. Above parameters are optional, and the default values can be calculated based on historical traffic matrices [39, 81]. The memory resource consumption of measurement tasks can then be automatically inferred based on above values (§4.1).

Decision: The control decision phase detects network events based on the statistics *stats* generated by the measurement phase as well as switch performance information such as queue size. Martini detects network events using conditional expressions, *e.g.,* $f(stats, switch) > threshold$ or $f(stats, switch)$ *is* (not) $true$. Function f performs calculations including *min*, *max*, *sum*, *and*, *or*, *not*, *etc.* Martini allows to detect multiple events at the same time.

Control: This phase executes control actions on the packets that trigger network events. To enable modularity and reusability, we abstract control actions into a unified paradigm *action_name* (*action_parameters*[]). Any control actions implemented in programmable data plane such as P4 switches, NetFPGA, or SmartNIC can expose their names and parameters to be used in Martini task description. We list typical control actions in data centers, wide area networks (WANs), and enterprise networks in Table 2. Operators can choose to `drop` or `forward` a packet, `report` events to the controller, enforce load balancing or congestion control, *etc.* Multiple control actions can be assigned to one event, while different events could result in the same control actions.

In summary, Martini provides modular, reusable, and extensible primitives for operators to collectively describe measurement, decision, control, and their connection. We will discuss potential extensions to Martini primitives in §7.

3.2 Example Management Tasks

Operators could compose primitives to describe management tasks based on a *one-big-switch abstraction* [10, 63]. Operators do not need to care about implementation and placement of the tasks as if they are deployed entirely in one single switch. This section specifies three example management tasks mentioned in §2. More examples are in Table 7.

```
1 microburst_balancing_task
2   .set_window (10μs)
3   .set_error_rate (5%)
4   .set_flow_number (10 ^ 6)
5   .set_flow_rate (10Gbps)
6   .filter (dstIP = 192.168.1.1/24)
7     .update (i = [5 tuple], f = add, v = 1)
8     .query (i = [5 tuple])
9     .detect (stats > 100).localflow([2, 3])
```

Figure 4. Microburst detection and load balancing.

Microburst detection and fast load balancing. As shown in Figure 4, we first assign a short measurement window of 10 μs (line 2) for this task. Next, we input the error rate, flow number, and flow rate (line 3-5) of the task for resource estimation. We filter interesting flows (line 6) and record in the granularity of 5 tuple. For each flow, we count (line 7) and query the number of packets within a window (line 8) for event detection. If the packet number exceeds a threshold [12], we will output the flow of the packet from the port with a shorter queue length for load balancing (line 9).

```
1 DNS_reflection_task
2   .set_window (1s)
3   .filter (udp.dstPort == 53)
4     .update (i = [srcIP, dstIP, transID], f = set, v = 1)
5   .filter (udp.srcPort == 53)
6     .query (i = [dstIP, srcIP, transID])
7     .detect (stats ≠ 1).drop().report_to_controller()
```

Figure 5. DNS reflection attack detection and defense.

DNS reflection attack detection and defense. As illustrated in Figure 5, we filter DNS requests (line 3), use source IP, destination IP, and DNS transaction ID as index, and record the existence of a request by setting its corresponding counter to 1 (line 4). For each DNS response packet, we query the existence of the corresponding request (line 5, 6). If the request does not exist, this response will be dropped. This event will also be reported to the controller (line 7).

```
1 superspreader_task
2   .set_window (10ms)
3   .update (i = [srcIP, dstIP], f = set, v = 1)
4   .groupby (k = [srcIP], f = sum)
5   .detect (stats > 100)
6     .forward (2).report_to_controller()
```

Figure 6. Superspreader identification and quarantine.

Superspreader identification and quarantine. This task (Figure 6) counts unique connections of each source IP address. We set a short measurement window for timeliness

Table 3. Mapping measurement description to sketches and estimating the resource consumption of each sketch. *u_* is short for *update_*, and *q_* stands for *query_*. We omit the sketches' usage of stateful actions for brevity.

u_key	u_func	u_value	q_type	Example	Sketch	Parameter	SRAM (bits)	Stage
wild card	add	1	query	Total packet count	*Single counter*		32	1
		otherwise	query	Total packet size	*Single register*		32	1
	set	1	query	Unused port detection	*Single bit*		1	1
Specific	add	1	query	Flow packet count	*Count-min sketch* [26]	Threshold: $0 < \phi < 1$ Per-flow relative error: ϵ	$\frac{64}{\epsilon\phi}$	1
		otherwise	query	Microburst detection / PIAS				
	set	1	groupby (*)	Unique connection number	*PCSA sketch* [33]	Relative error: ϵ	$(\frac{0.78}{\epsilon})^2 \times (32 - 2\lceil \log_2 \frac{0.78}{\epsilon} \rceil)$	6
			query	DNS reflection attack defense	*Bloom filter* [15, 37]	Number of flows: n Hash function number: k False positive rate: ϵ	$\frac{kn}{\ln[1/(1-\sqrt[k]{\epsilon})]}$	2
			groupby	Superspreader / DDoS victim	*Bloom filter + Count-min*	Input of both sketches	Sum of two sketches	3
change ()				Flow size change detection	*k-ary sketch* [27, 80]	Threshold: $0 < \phi < 1$ Per-flow relative error: ϵ	$\frac{512}{\epsilon^2\phi^2}$	2
distribution ()				Flow size distribution	*multi-resolution sketch* [58]		34M	4

(line 2). We maintain connections in the granularity of source and destination IPs (line 3), and group the record with the key of source IP. If the value exceeds a threshold (line 5), the packets of this IP will be forwarded to a honeypot for quarantine and the event will be reported to the controller to maintain the record of potential superspreaders (line 6).

4 Network-wide Task Placement

Martini controller takes the descriptions of management tasks as input and places tasks on switches in the entire network. We first estimate the resource usage of each task, and then partition tasks into fine-grained *subtasks* that could be distributed to all switches. Finally, we perform network-wide task placement to optimize global resource usage.

4.1 Task Resource Estimation

One management task may operate on many flows that follow different paths. We split one task into several task slices, each of which only manages flows following the same path. We later use *task* to represent *task slice* for brevity. Based on the task description, we estimate the resource usage of each phase in a task as input for placement.

Measurement: Recent researches have proposed many techniques to support measurement inside switches, such as sketches [47, 63, 69, 94], hash tables [4, 86], sampling [24, 81, 92], *etc.* We select sketches with *high resource efficiency* and *bounded error* for measurement. As listed in Table 3, we first *map* the description of the measurement phase into its corresponding sketch. Then we theoretically calculate the SRAM usage of the sketch based on the input or default *error_rate*, *flow_number*, and *flow_rate*. We also collect the stage usage of each sketch based on our implementation (§5).

Note that operators could possibly input inaccurate flow number or flow rate and lead to inaccurate resource estimation. In this case, operators could deploy a simple measurement task to gather accurate flow statistics during runtime [84]. If flow statistics change, operators could adjust resource allocation and deploy a new measurement task.

Decision: As mentioned above, we have defined the control decision phase as a conditional expression. This phase occupies one pipeline stage and negligible SRAM resources to detect network events based on measurement results.

Table 4. Stage consumption of control actions.

Primitive	Stage	Primitive	Stage	Primitive	Stage
drop	1	forward	1	report_to_controller	1
ecmp	2	wcmp	2	spray	2
flowlet_ecmp	5	rate_limit	1	ecn	1
qcn	4	hull	3	drill	6
set_queue	2	red	2	wred	3

Control: Based on our implementation (§5), we collect and list the pipeline stage consumption of Martini control actions in Table 4. Each control action requires negligible SRAM to hold a few flow table entries as control rules.

4.2 Computation and Data Partition

A programmable switch has limited resource including pipeline stages (1-32), SRAM (tens to hundreds of Mbs), and stateful actions [18, 39]. However, as shown in Tables 3 and 4, measurement could take lots of SRAM resources, while both measurement and control phases may consume many pipeline stages. Therefore, it is infeasible for some management tasks to be entirely accommodated inside one switch.

To address this challenge, we present our key observation that *a flow may go through multiple switches* before reaching its destination. Therefore, a task can be placed on any switch in the path to correctly manage a flow. Based on this observation, we propose to split a task into fine-grained *subtasks* and distribute them onto the switches alongside the flow forwarding path. We split a task in two ways including *computation partition* and *data partition*.

Computation partition: A management task defined in Martini can be naturally split into three segments including measurement, decision, and control. We could correctly support a management task by distributing the three phases onto different switches while ensuring their *ordering*. However, such partition introduces extra information transmission among switches and may affect the goodput. We notice that the measurement phase generates *statistics* (hundreds of bits) for decision, while the decision phase detects *network events* (of a few bits) and informs the control phase. As the decision phase merely occupies one pipeline stage and few SRAM, we *couple measurement and decision* and simply partition a task

Table 5. Notations for network-wide task placement.

(Output) Variables and indexes	
$x_{(j,t)}^{(i,p)}$	0-1 variable indicating the start stage of subtask p
(i, p)	subtask p of task i
(j, t)	stage t on switch j
(Input) Subtasks	
$\mathcal{T}$	Set of tasks
sub_i	Number of subtasks of task i
$CS^{(i,p)}$	Number of stages consumed by subtask (i, p)
$Mem_t^{(i,p)}$	Memory consumed by the t-th stage of subtask (i, p)
$State_t^{(i,p)}$	Number of stateful actions for the t-th stage of subtask (i, p)
C	Set of ordering dependencies
$(i, p_1 \trianglelefteq p_2)$	Subtask (i, p_1) should be before (i, p_2) on the path of task i
(Input) Forwarding Information	
$\mathcal{S}$	Set of switches
$path_i$	Path of task i
$\alpha_{i,j}$	Position of switch j on the path of task i
$\delta_{i,j}$	Indicate whether switch j is on the path of task i
$\rho_i^{(j,t)}$	Stage t on switch j is the $\rho_i^{(j,t)}$-th stage along $path_i$
(Input) Hardware Resource Constraints	
SN	Number of stages inside a switch
$StageMem$	Total SRAM memory inside a switch
$StageState$	Total number of stateful actions inside a switch

Table 6. Formulation for network-wide task placement.

Objective:

$$\min \sum_{j\in\mathcal{S}} \sum_{t=1}^{SN-1} sgn\left(occupy^{(j,t)}\right)$$

where

$$occupy^{(j,t)} = \sum_{i\in\mathcal{T}} \left(\delta_{i,j} \cdot \sum_{p=1}^{sub_i} \left(\sum_{\tau=1}^{\min\{CS^{(i,p)},t\}} x_{(j,t-\tau+1)}^{(i,p)} \right) \right)$$

Constraints:

(C1) $$\forall (j,t): \sum_{i\in\mathcal{T}} \left(\delta_{i,j} \cdot \sum_{p=1}^{sub_i} x_{(j,t)}^{(i,p)} \otimes Mem_t^{(i,p)} \right) \leqslant StageMem$$

(C2) $$\forall (j,t): \sum_{i\in\mathcal{T}} \left(\delta_{i,j} \cdot \sum_{p=1}^{sub_i} x_{(j,t)}^{(i,p)} \otimes State_t^{(i,p)} \right) \leqslant StageState$$

(C3) $$\forall (p_1 \trianglelefteq p_2,\, i) \in C: \sum_{j,t} \left(\rho_i^{(j,t)} \cdot x_{(j,t)}^{(i,p_1)} \right) + CS^{(i,p_1)} \leqslant \sum_{j,t} \left(\rho_i^{(j,t)} \cdot x_{(j,t)}^{(i,p_2)} \right)$$
where $\rho_i^{(j,t)} = (\alpha_{i,j} - 1) \cdot SN + t$

(C4) $$\forall (i,p): \sum_{j\in\mathcal{S}} \sum_{t=1}^{SN-1} x_{(j,t)}^{(i,p)} = 1$$

into *measurement* and *control*, so that goodput will not be much affected. Moreover, multiple control phases in one task can also be partitioned as long as we ensure their ordering.

Data partition: Each pipeline *stage* in programmable switches has limited SRAM (tens of Mbs) and stateful actions [39]. However, measurement is SRAM intensive. For instance, according to Table 3, a Bloom Filter that monitors 10^6 flows using 2 hash functions with 1% false positive rate needs 19Mb SRAM, which may exceed the capacity of a single stage. Meanwhile, we observe that the SRAM usage of sketches such as Bloom Filter is positively correlated to the number of flows. Therefore, we *equally divide the flows* (as in number of flows) into several partitions, and use multiple instances of the same sketch to monitor the flow partitions. To minimize stage usage, we minimize the number of partitions while ensuring one stage has enough resources to measure one partition. We name this technique as data partition.

After computation and data partition, a task is split into several subtasks. Next we place the subtasks on switches with respect to resource and ordering constraints.

4.3 Network-wide Placement Algorithm

The algorithm takes descriptions of management tasks as input. We first split a task into multiple task slices, each of which manages flows on one unique path. We then partition task slices into subtasks and model the placement as a 0-1 Nonlinear Programming problem in Table 6. We list related notations in Table 5. To *accelerate* problem resolution, we further *linearize* the objective into an Integer Linear Programming problem (ILP) without sacrificing accuracy or optimality. Details can be found in Appendix A. We next introduce the objective and constraints.

Objective. Our intuition is that management tasks should occupy minimal switch resources to make space for general switch functions such as forwarding. However, a switch has *multiple types of mutually dependent resources*. Each stage has fixed amount of SRAM and stateful actions [18, 39]. The problem is which type of resource to optimize. We notice that the total SRAM and stateful actions usage of all tasks can be pre-estimated, while subtasks can share a pipeline stage if the SRAM and stateful actions in the stage are sufficient. Therefore, we choose to minimize the *stage occupancy* of all tasks. As shown in Table 6, $occupy^{(j,t)} > 0$ indicates that pipeline stage t on switch j is occupied. We use sgn function to normalize $occupy$ to $\{0, 1\}$, where $sgn(x) = 1\ if\ x > 0$.

Resource Constraints (C1, C2). Similar to [49], we analyze SRAM and stateful actions constraints in the granularity of a *stage*. As one subtask may span across multiple stages and consume different amounts of SRAM and stateful actions in each stage, we use $x_{(j,t)}^{(i,p)} = 1$ to indicate that the *first stage* of subtask p of task i is the jth stage in switch t. Thus the SRAM for subtask (i, p) on stage (j, t) is:

$$\sum_{\tau=1}^{\min\{CS^{(i,p)},t\}} x_{(j,t-\tau+1)}^{(i,p)} \cdot Mem_\tau^{(i,p)} \tag{1}$$

The formula is the *convolution* of $x_{(j,t)}^{(i,p)}$ and $Mem_t^{(i,p)}$ [74], denoted as $x_{(j,t)}^{(i,p)} \otimes Mem_t^{(i,p)}$ for simplicity. Thus the total memory resource on stage (j, t) should satisfy (C1). Similarly, we can also get the constraints of stateful actions (C2).

Ordering Constraints (C3). We should ensure the ordering between measurement and control subtasks and between control subtasks. Suppose subtask p_1 should be placed *before* p_2 in $path_i$, denoted as $(i, p_1 \trianglelefteq p_2)$. The start stage (j_1, t_1) and (j_2, t_2) of subtasks (i, p_1) and (i, p_2) should satisfy:

$$\rho_i^{(j_1,t_1)} + CS^{(i,p_1)} \leqslant \rho_i^{(j_2,t_2)} \tag{2}$$

However, we do not know the start stages of p_1 and p_2 since they are exactly the variables to solve. This makes it impossible to directly use the above expression as the constraint. To address this challenge, we first number the

stages along the $path_i$. The t^{th} stage of switch j on $path_i$ is numbered as $\rho_i^{(j,t)} = (\alpha_{i,j} - 1) \cdot SN + t$. Meanwhile, the solution of subtasks (i, p_σ) $(\sigma = 1, 2)$ satisfies:

$$x_{(j,t)}^{(i,p_\sigma)} = \delta_{j,t}\left(\rho_i^{(j,t)} - \rho_i^{(j_\sigma,t_\sigma)}\right),\ \sigma = 1, 2 \tag{3}$$

where $\delta_{j,t}(x) = 1$ iff $x = 0$, and 0 otherwise. Next, we innovatively exploit the *sifting property* [93] of *impulse function* $\delta(\cdot)$ and turn implicit constraints into explicit expressions:

$$\rho_i^{(j_\sigma,t_\sigma)} = \sum_{j,t}\left(\rho_i^{(j,t)} \cdot \delta_{j,t}\left(\rho_i^{(j,t)} - \rho_i^{(j_\sigma,t_\sigma)}\right)\right) \tag{4}$$

Variable Constraints (C4). Finally, each subtask can only be placed once. Therefore, C4 should be satisfied.

Summary: The above 0-1 ILP problem can be solved within limited time [40] in Martini. We evaluate the effectiveness and efficiency of the algorithm in §6.3.

Runtime incremental task deployment: During runtime, operators may deploy new tasks or enable deployed tasks to monitor a new set of flows with new forwarding paths, which we also consider as new tasks. To avoid disrupting deployed tasks, we *trade optimality for stability* through incremental deployment. We first estimate resource usage of new tasks and partition them into subtasks. Then we perform incremental network-wide placement of the new subtasks by using the same 0-1 ILP formulation while modifying the resource constraints to represent *remaining* resources in the network. We evaluate its efficiency in §6.3.

5 Implementation Details

We implement the Martini framework based on Barefoot Tofino [73], a state-of-the-art PISA target. In this section, we first elaborate our implementation of the *component library*, which serves as the code template for measurement and control subtasks. Then we introduce the Martini compiler that automatically deploys massive management tasks onto switches based on placement results.

5.1 Component Library

We use P4 [17] to create the component library that can run on Tofino. We implement all sketches introduced in Table 3 and control actions in Table 4. We encounter two major challenges in the implementation of the code library.

Support for the measurement window: Measurement functions often collect statistics with a window [63, 94]. Thus we need to maintain a *timer* to periodically *clear* counter arrays in switches. However, P4 does not provide hardware primitives for timer or counter array clearance, making it uneasy to support windows in PISA switches.

Martini addresses this challenge with the following approach. To simulate the timer, we exploit the *timestamp* that is attached to every packet once it enters the switch pipeline [55, 72]. We store the initial time `cur_time` in the switch. For each packet, we use its timestamp to subtract the `cur_time`. Once the difference exceeds the measurement window, the `cur_time` is added by the interval to indicate a new round. Next, we enable each packet to clear its corresponding flow counter. However, if we just use one register to maintain time, only the counter of the first packet in a new round will be cleared. Therefore, we maintain a time register for every counter in counter arrays. To reduce the resource overhead, we design a *round based timing* mechanism shown in Figure 7. We use an 8-bit *round* to represent the 48-bit timestamp. We update the round in the *round indicator* and attach it to the packet. If the round in the packet is larger than `cur_r` of a counter, the counter is cleared. For a 32-bit counter, the resource overhead of this approach is 25%.

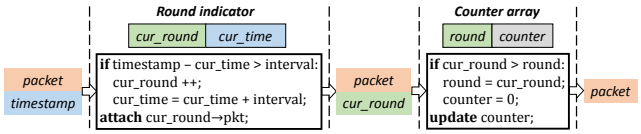


Figure 7. Round based timing mechanism.

However, representing a 48-bit timestamp with an 8-bit round may introduce errors. The 8-bit round value repeats the cycle of increasing from 0 to 255 and then returns to 0. If two consecutive packets in a flow come within an interval of approximately 256 rounds, the `cur_round` for the two packets will be the same due to wrap-around thus the corresponding counter will be incorrectly updated instead of reset, which causes errors. However, evaluation in §6.4 shows that the error rate tops at 0.2% for real world traces.

Finally, different measurement tasks on one switch may require measurement windows with different lengths. We allocate a separate round indicator for each window length.

Control based on queue lengths: Control actions such as `drill` or `LocalFlow` decide which queue a packet or a flow should enter based on the lengths of candidate queues. However, in PISA [18, 39], queue length information can only be obtained in the egress pipeline, while only the ingress pipeline can adjust the target queue of the packet. This is because in high speed networking chips, back-propagating the fast changing sizes of all queues to the ingress pipeline at a nanosecond timescale is generally extremely expensive. To address this challenge, in the egress pipeline, Martini *samples* packets at a rate of 5% for each queue and *clones* the sampled packets as probes. For each probe, Martini *tags* the length of the queue into the probe and *recirculates* it to ingress. Ingress then collects queue lengths from probes and *drops* them. In this way, we could acquire queue lengths in ingress. As probes are recirculated from egress to ingress using recirculation bandwidth in each pipeline and are dropped in ingress, the throughput will not be compromised. We demonstrate this in §6.2.

5.2 Martini Compiler

According to the network-wide placement result of subtasks, Martini compiler generates codes and table entries for all network switches by assembling the component library. For each switch, the compiler composes the codes of the

Table 7. Management tasks implemented in Martini. We show the *lines of code* to describe management tasks without resource related primitives in column *Martini*, and the lines of generated P4 code in *P4*.

Task type	#	Task that handles ...	Description: The task measures network traffic → identifies ...	*Martini*	*P4*
Attack Defense	1	TCP SYN flood [95]	IPs that receive more half-open TCP connections than threshold → drops later SYN packets	6	232
	2	Port scan [50]	IPs that send traffic to more than a threshold of destination ports → drops their packets	6	290
	3	DDoS victim [94]	IPs that receive traffic from more than a threshold of unique sources → performs RED on those traffic	6	322
	4	DNS reflection attack [57]	DNS responses without corresponding requests → drops the illegal requests	7	260
	5	NTP amplification attack [78]	IPs that receive NTP packets from more than a threshold of unique sources → drops these packets	7	291
	6	Stateful firewall [67]	Unsolicited inbound TCP connections without any outbound flows → drops the connections	8	245
Anomaly Detection	7	Superspreader [94]	IPs that contact more than a threshold of unique destinations → reports to the controller	6	303
	8	FTP monitoring [67]	FTP data channel setup requests when their control channels are not established → drops the requests	8	237
	9	Heavy changer [27]	Flows whose sizes have changed significantly across two intervals → reports to the controller	7	386
Flow Scheduling	10	Heavy hitter [63]	Flows whose size exceed threshold → performs per-flow port balancing	6	259
	11	Microburst [36]	Flows whose packet numbers within a window exceed a threshold → performs queue-based balancing	9	319
	12	PIAS [11]	Flow bytes sent exceed a threshold → inserts flow into a lower priority queue and ECN on congestion	8	207
	13	Video congestion control [16]	An I frame in an MPEG stream is dropped → drops later differentially-encoded B frames	7	296
	14	Link failure recovery [21]	Failure-carrying packets coming backward → uses backup routes for subsequent flows on this path	8	237
Network Monitoring	15	Flow size distribution [94]	The distribution of flow sizes → reports this information to the controller	7	240
	16	TCP incast [96]	IPs that receive TCP connections from more than a threshold of unique sources → informs controller	6	290

subtasks according to their start stage and resource usage. Control actions may be implemented in the ingress or egress pipeline, which is intelligently identified by the compiler to place their codes at the right place. We pay special attention to the following challenges during compilation.

Flow classification for data partition: Data partition requires us to split flows on a path to several measurement subtasks. This additional flow classification is decided dynamically during task placement and cannot be predefined in the components. To realize this goal, the compiler *reuses* the `filter` stage in each measurement subtask to identify the class of flows it should monitor. In this way, we support classification without consuming an extra stage.

Event transmission for computation partition: Computation partition requires transmitting network events between measurement and control. If the two phases are placed in the same switch, we create a *metadata* to carry the event. Otherwise, we modify the packet headers for event delivery between switches. However, adding new header fields may hurt the goodput. We refer to prior wisdom [30, 41] and exploit unused bits in current headers including the 20-bit Flow Label field in IPv6, 6-bit DS field in IPv4, and the 12-bit VLAN Identifier field if unused. Martini compiler enables the measurement component to tag the metadata or headers based on whether the two phases are in the same switch.

6 Evaluation

We implement the Martini framework on a testbed with several Barefoot Tofino switches and servers. Each server is equipped with two Intel(R) Xeon(R) E5-2690 v2 CPUs (3.00GHz, 10 physical cores), 256G RAM and two 10G NICs. For test traffic, we use real world data traces from CAIDA [22], and replay the traces at 100Gbps using the Spirent Test Center [25]. For test topology, we simulate real world topologies including Fat-tree [1], AT&T [56], and Stanford campus backbone [54]. Our evaluation goals are to:

- demonstrate the expressivity of the Martini description primitives to describe many management tasks. (§6.1)
- demonstrate that comparing to the traditional pattern, Martini could significantly reduce the control loop while maintaining high throughput, and therefore could effectively support tasks that require timely control. (§6.2)
- demonstrate that the network-wide placement could achieve optimal resource usage for real world topologies and tasks within reasonable calculation time. (§6.3)
- demonstrate that the error caused by the round based timing mechanism is tiny for real world traces. (§6.4)
- demonstrate the capability of the compiler to generate codes for all network switches within little calculation time to deploy tasks on real world topologies. (§6.5)

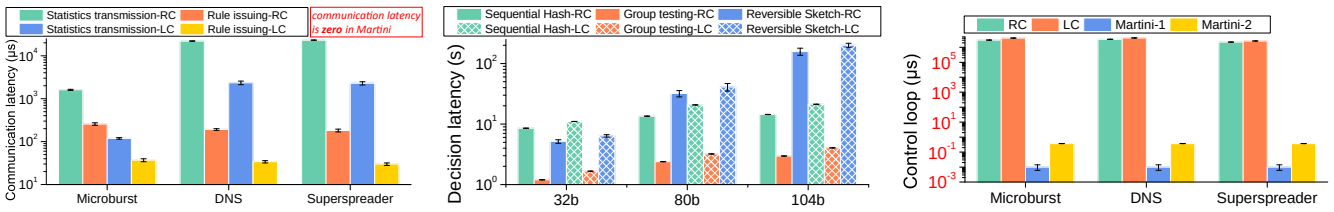
6.1 Expressivity

To demonstrate the expressivity of the Martini task description primitives, we specify sixteen common management tasks shown in Table 7. Among them, the DNS (#4) and FTP tasks (#8) follow the *update-query* scheme introduced in §3.1, while other tasks follow the *measure through* scheme. Tasks 2, 3, 5, 7, and 16 use the `group` primitive, while others use the `query` primitive for measurement. Task 3 performs control in egress. Tasks 11 and 12 control in both ingress and egress. Other tasks control in ingress. We also show that Martini makes it easier to describe management tasks by composing primitives. It takes less than 10 lines of code to describe a task in Martini, while requiring around 30× lines of code to implement the same task in P4.

6.2 Control Loop Reduction

To demonstrate that Martini could effectively reduce control loop, we implement both traditional and Martini patterns on our testbed and deploy all 16 tasks in Table 7. For the traditional pattern, we perform measurement in a Tofino switch and control decision in a server that is directly connected to the switch. We measure the latency of components of its control loop including *switch report interval*, *statistics transmission* to the controller, *control decision*, and *rule issuing* to the switch[1]. We also implement a strawman approach

[1]The latency between switch CPU & hardware pipeline is a few *ms*.



(a) Communication latency (b) Control decision latency (c) Control loop

Figure 8. Control loop and latency of its components of traditional remote control pattern (RC), local control pattern (LC), Martini one-switch pattern (Martini-1), and Martini two-switch pattern (Martini-2).

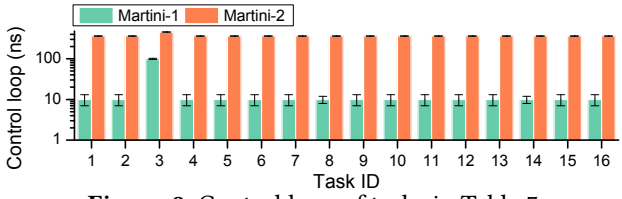


Figure 9. Control loop of tasks in Table 7.

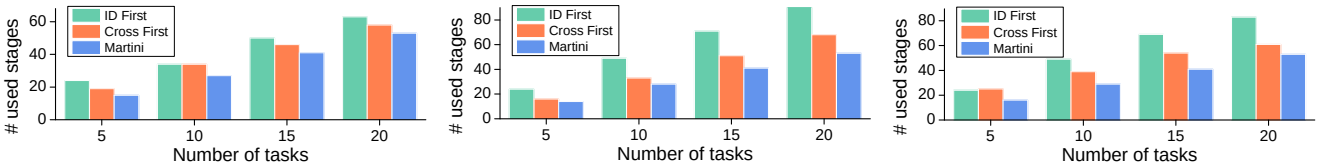
that performs control decision in the *switch local CPU*, which could possibly shorten the control loop by reducing communication latency. Meanwhile, Martini implements both measurement and control in the switch pipeline. We measure the entire pipeline latency with or without control decision and calculate the difference as the control loop. Moreover, due to computation partition, measurement and control may be placed on two switches. In this case, we deploy measurement and control on two directly connected Tofino switches. We make the first switch timestamp a packet and send it to the second switch, and enable the second switch to send the packet back after control. The first switch timestamps the packet again and we report half of the timestamp difference as the control loop. *For each experiment, we report the average latency and the standard deviation across 100 runs.*

Communication latency: We use ZeroMQ [44] for communication between switches and the controller. We set the total flow number as 1.2M according to CAIDA traces, and set the error rate of the sketch in each task as 5%. According to Table 3, for the microburst task, a switch needs to report 1.28Mb statistics to the controller. For the DNS task, the statistics to report are 19.29Mb. The superspreader task needs to upload 20.57Mb. As shown in Figure 8(a), the communication latency is positively correlated to the data size. Uploading statistics takes much longer time than rule issuing. Statistics transmission in the remote control pattern takes 3 to 50 *ms*. Despite local control in switch CPU could significantly reduce the communication latency by 90%, several milliseconds latency still prevents supporting tasks that require timely control within a few microseconds such as the microburst task. In comparison, Martini places measurement and control in switches and *completely avoids this latency.*

Control decision latency: Based on received statistics, the controller makes control decisions by first decoding the counters to derive flow keys. We evaluate three techniques for this process including sequential hashing [20], group testing [27], and reversible sketch [47, 94]. We vary the flow key length and measure the algorithms' running time. For the microburst task, we monitor flows at 5 tuple granularity of 104 bits. For the DNS task, the flow key is 80 bits in total. For the superspreader task, the key is the 32-bit srcIP. We feed CAIDA traces into the sketches in the data plane and decode the sketches with the three algorithms. The results in Figure 8(b) show that the calculation time rises as the key length increases. We also find that group testing takes the shortest time of a few milliseconds, while reversible sketch takes up to 3.5 minutes. Furthermore, running algorithms in the switch CPU takes 30% to 50% more time than the server as switches use relatively low-end CPUs. Such significant latency seriously compromises control timeliness. Note that Martini performs control directly on the packets that trigger events. Therefore, Martini *avoids key deriving latency* and shortens the decision latency to a few *ns*, *i.e.,* the latency of a single pipeline stage. We present this value below.

Total control loop: Finally, we present the total control loop of the three example tasks in traditional and Martini patterns. For the traditional pattern, we set the report interval of the microburst task as 100*ms* [2], and that of DNS and superspreader tasks as 1*s* [39]. We use group testing for fast key deriving. We sum up all components of the control loop and present the results in Figure 8(c). We observe that the control loop of the traditional patterns is 2 to 4 seconds. For the Martini one-switch case, the control loop is the time for control decision in a single pipeline stage, which takes around 10 *ns*. For the Martini two-switches case, the control loop grows to around 370 *ns* due to the event queuing and transmission latency. Furthermore, we measure the control loop of all tasks in Table 7 in Martini. Results in Figure 9 demonstrate that above observations hold across all tasks in Martini. Note that the latency of the DDoS task (#3) in one-switch case reaches 126 *ns*. This is because its control action (RED) occurs in the egress pipeline, while other task control in ingress. Such tiny latency could support tasks requiring extremely short control loop such as the microburst task.

High throughput maintenance: We evaluate the capability of Martini to maintain high throughput even for control actions such as `drill` that clone and recirculate packets to deliver information from egress to ingress. We replay CAIDA traces at 100Gbps and measure the throughput of the tasks



(a) Fat-tree topology (b) AT&T topology (c) Stanford campus backbone

Figure 10. Total number of used stages across all switches for naive algorithms and Martini task placement algorithm.

Figure 11. Running time of Martini task placement algorithm.

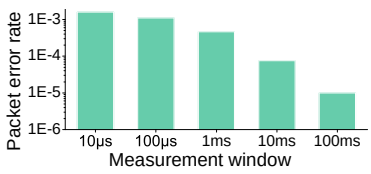


Figure 12. Error caused by round based timing.

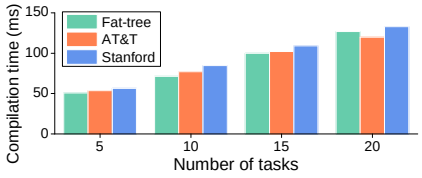


Figure 13. Compilation time for switch code generation.

Table 8. Topologies tested in our experiments

Topology	Description	Switch	Edge
Fat-tree [1]	Fat tree network with $k = 4$	20	32
AT&T [56]	AT&T North America backbone network	25	52
Stanford [54]	Stanford campus backbone network	26	56

in Martini one-switch and two-switch setups. Evaluation results show that all 16 tasks could maintain the *line rate* of 100Gbps including the microburst task (#11) that requires packet cloning and recirculation. This demonstrates Martini's ability to maintain high throughput.

Effective support for management tasks that require timely control: Above results in reducing the control loop means that Martini could effectively support management tasks that require timely control. As revealed in [82], for the microburst task, enabling local control could improve total bandwidth by 19.5% and reduce average flow completion time by 12.6% for a heterogeneous VL2 workload on a 512-host, 4:1 oversubscribed FatTree. For the DNS task, when the attack traffic rate is $300Gbps$, reducing the control loop from $3.4s$ to $10ns$ could theoretically defend against $1020Gb$ attack traffic. For the superspreader task, operators could reduce the superspreader detection time from $3.4s$ to $10ns$ and therefore prevent worm propagation more timely.

6.3 Network-wide Task Placement

We compare the Martini placement algorithm with two strawman solutions including an *ID First* algorithm and a *Cross First* algorithm. The *ID First* algorithm places subtasks following the same path sequentially in the front most available switch in the path. This ensures that a flow is only managed once by an subtask [63]. In the *Cross First* algorithm, we prefer to place subtasks on the switches that are in the forwarding paths of the most number of tasks. In this way, subtasks on these switches could share stages, which potentially reduces the total number of used stages.

We implement the algorithms in a server and simulate the topologies of three real-world networks in Table 8 as input. We randomly pick 5, 10, 15 and 20 times from Table 7 as test task sets, and randomly assign one path to each task to make it a task slice. We measure the total number of used stages across all switches according to the results of the three algorithms. We use LINGO 17.0 [62] to quickly solve the 0-1 ILP problem in Martini. As shown in Figure 10, Martini outperforms the two naive algorithms in all topologies by occupying 9.4% to 56.3% fewer stages. Furthermore, as presented in Figure 11, Martini can quickly generate the optimal placement result within 2.5 minutes even for the largest scale configuration with over 8,000 variables and constraints in the 0-1 ILP. Note that this algorithm *runs offline only once for initial placement*. In our future work, we will adopt advanced solvers in Matlab and other acceleration mechanisms [60] to further reduce the calculation time.

Table 9. Optimality gap of incremental task placement

# tasks	Fat-tree (# stages)		AT&T (#stages)		Stanford Campus (#stages)	
	Optimal	*Incremental*	*Optimal*	*Incremental*	*Optimal*	*Incremental*
5	15	15	14	14	16	16
10	27	31 (+14.8%)	28	28 (+0%)	29	30 (+3.4%)
15	41	45 (+9.8%)	41	46 (+12.2%)	41	48 (+17.1%)
20	53	58 (+9.4%)	53	62 (+17.0%)	53	64 (+20.8%)

For incremental deployment of a new task, Martini performs the same ILP formulation with updated resource constraints. The calculation time is below $1s$ according to Figure 11. However, incremental task placement may be suboptimal. To evaluate its optimality gap from optimal placement, we deploy 5 initial tasks and incrementally deploy new tasks one by one until 20 tasks are deployed. As shown in Table 9, incremental placement consumes 9.4% to 20.8% more stages than the optimal solution depending on the topology, while achieving task deployment stability and fast calculation.

6.4 Error Caused by Round Based Timing

We evaluate the percentage of faulty counter updates due to the wraparound issue in round based timing (§5.1). Based on CAIDA traces, we vary the measurement window and count packets that cannot trigger correct counter clearance by identifying packet pairs whose time intervals are between 255× to 257× measurement windows. As shown in Figure 12,

the percentage of packets causing error increases as the measurement window decreases, and tops at 0.2% when the window is 10μs. We consider this as acceptable since SRAM usage for timing is significantly reduced by 83.3%.

6.5 Compiler Performance

Finally, we demonstrate the scalability of the Martini compiler to quickly generate codes for all network switches in real world topologies. We implement the Martini compiler in an isolated server and use a single core to run the compiler program. We compile the placement results of 5 to 20 tasks on the test topologies in §6.3 and measure the compilation time. As shown in Figure 13, Martini compiler could create codes for switches in real-world topologies within 200 *ms*. This demonstrates its scalability to quickly generate codes.

7 Discussions

Runtime modification of tasks or forwarding rules. Martini has the potential to handle runtime dynamics. First, Martini maintains the measurement window, decision rules, and control rules as table entries, which can be dynamically configured during runtime. Second, as mentioned in §4.3, operators may intend to deploy new tasks or enable deployed tasks to monitor a new set of flows. Besides, the forwarding paths of monitored flows could change. Martini considers above dynamics as new tasks and performs *efficient* (§6.3) incremental network-wide placement to handle them.

Supporting complex or global management tasks: Some management tasks need to maintain *multiple statistics* for control decision. For example, to detect Slowloris Attacks [39], we need to monitor the number of unique connections and bytes sent in each connection. To support this, we could slightly extend the primitives to allow multiple measurement phases in one task. Moreover, operators may intend to place a management task at a specific *position* in the network (*e.g.,* leaf switches in data centers). We could also provide interfaces to specify the task location. We consider above primitive extensions as future work. Finally, some tasks require *global* information from multiple switches for control decision, such as detecting network-wide heavy hitters [42]. Martini can support this kind of tasks by assigning the `report_to_controller` action after event detection. The global task in the controller can then gather information for control. Thus, Martini is already helpful for the description, placement, and implementation of global tasks.

Stability, scalability, and vulnerability: Finally, we discuss several system concerns. (1) In Martini, control action is triggered a few nanoseconds after the decision based on measurement results, which may lead to control *instability*. A potential solution is to use a hardware-supported low-pass filter to smoothen measurement results and stabilize control [48]. (2) Martini task placement may suffer from long calculation time as topology becomes larger, incurring a *scalability* issue. Nevertheless, Martini runs the placement algorithm only once and offline for initialization. Runtime incremental task deployment is proved to be fast. (3) Offloading too much logic to switches may make switches *vulnerable* to exhaustion attacks. However, Martini allocates fixed resources to sketches according to estimation. Flooding attacks could merely compromise measurement accuracy.

8 Related Work

Network measurement or management frameworks: Recent studies [47, 60, 61, 63, 72, 86, 94, 95] have proposed frameworks for high performance and resource efficient measurement in switches. In comparison, Martini is a general framework that supports measurement-based timely control in management tasks. Trumpet [32] proposed to detect network events in end hosts. Martini chooses a switch-based solution to avoid wasting CPU for networking [70], ensure predictable performance, and bypass potential security issues [89]. SNAP [10] offered a centralized framework for stateful task description and deployment based on NetASM. In comparison, Martini focuses on a specific set of tasks, *i.e.* management tasks that require measurement-based timely control and addresses the challenges of task description, network-wide placement, and implementation on advanced programmable switching ASICs.

Task description languages: Many recent works have proposed languages to describe either measurement or control tasks. For measurement tasks, Trumpet [70] designed a match-action like language to define network events. Marple [72] focused on querying performance information. Sonata [39] proposed primitives that imitated stream processing. NetQRE [95] focused on quantitative monitoring. Meanwhile, Pyretic [66], NetKat [9], FlexSwitch [84] and so on have been proposed to describe control tasks. In contrast, Martini presents high-level *modular*, *reusable*, and *extensible* primitives to easily describe and assemble all phases in a management task.

Management task placement: The need for placing tasks with respect to switch resource constraints has been identified by many researches [10, 39]. However, above works mainly solved the placement problem on one single switch, while Martini performs network-wide placement. Some researches [63, 81] proposed to place measurement tasks on multiple switches in the network. Especially, cSamp [81] also exploited data partition for measurement task placement, while Martini innovatively proposes computation partition and models ordering constraints to handle more resource-intensive management tasks. Some researches [51, 71] have proposed mechanisms to place management rules in the entire network. However, they only considered the memory constraints in SDN switches, while Martini takes into account all types of resource constraints in switching ASICs.

9 Conclusion and Future Work

This paper presents Martini, a general framework that enables measurement-based timely network control using programmable switching ASICs. Our expressive task description primitives can describe many network management tasks. Our network-wide task placement algorithm is able to exploit the resources of all network switches to accommodate massive tasks. Our component library and compiler ensure efficient code generation for switches. Evaluations show that Martini could reduce the control loop to nanoseconds. In our future work, we will enrich the primitives to support more complex tasks and propose techniques to detect primitive conflicts. Furthermore, we will consider optimizing resource usage by sharing the same measurement and control tasks. Finally, we will study how to better support global tasks.

References

[1] Al-Fares, M., Loukissas, A., and Vahdat, A. A scalable, commodity data center network architecture. In *SIGCOMM* (2008), vol. 38, pp. 63–74.

[2] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic flow scheduling for data center networks. In *NSDI* (2010), vol. 10, pp. 19–19.

[3] Alert, U.-C. Udp-based amplification attacks, 2014.

[4] Alipourfard, O., Moshref, M., and Yu, M. Re-evaluating measurement algorithms in software. In *HotNets* (2015), p. 20.

[5] Alizadeh, M., Atikoglu, B., Kabbani, A., Lakshmikantha, A., Pan, R., Prabhakar, B., and Seaman, M. Data center transport mechanisms: Congestion control theory and ieee standardization. In *Annual Allerton Conference on Communication, Control, and Computing* (2008), pp. 1270–1277.

[6] Alizadeh, M., Edsall, T., Dharmapurikar, S., Vaidyanathan, R., Chu, K., Fingerhut, A., Matus, F., Pan, R., Yadav, N., Varghese, G., et al. Conga: Distributed congestion-aware load balancing for datacenters. In *SIGCOMM* (2014), vol. 44, pp. 503–514.

[7] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data center tcp (dctcp). In *SIGCOMM* (2010), vol. 40, pp. 63–74.

[8] Alizadeh, M., Kabbani, A., Edsall, T., Prabhakar, B., Vahdat, A., and Yasuda, M. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *NSDI* (2012), pp. 19–19.

[9] Anderson, C. J., Foster, N., Guha, A., Jeannin, J.-B., Kozen, D., Schlesinger, C., and Walker, D. Netkat: Semantic foundations for networks. In *SIGPLAN Notices* (2014), vol. 49, pp. 113–126.

[10] Arashloo, M. T., Koral, Y., Greenberg, M., Rexford, J., and Walker, D. Snap: Stateful network-wide abstractions for packet processing. In *SIGCOMM* (2016), pp. 29–43.

[11] Bai, W., Chen, K., Wang, H., Chen, L., Han, D., and Tian, C. Information-agnostic flow scheduling for commodity data centers. In *NSDI* (2015), pp. 455–468.

[12] Benson, T., Anand, A., Akella, A., and Zhang, M. Understanding data center traffic characteristics. *SIGCOMM CCR 40*, 1 (2010), 92–99.

[13] Benson, T., Anand, A., Akella, A., and Zhang, M. Microte: Fine grained traffic engineering for data centers. In *CoNEXT* (2011).

[14] Bianchi, G., Bonola, M., Capone, A., and Cascone, C. Openstate: programming platform-independent stateful openflow applications inside the switch. *SIGCOMM CCR 44*, 2 (2014), 44–51.

[15] Bloom, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (1970), 422–426.

[16] Bonomi, F., Mitzenmacher, M., Panigrah, R., Singh, S., and Varghese, G. Beyond bloom filters: from approximate membership checks to approximate state machines. In *SIGCOMM* (2006), vol. 36, pp. 315–326.

[17] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al. P4: Programming protocol-independent packet processors. *SIGCOMM CCR 44*, 3 (2014), 87–95.

[18] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *SIGCOMM* (2013).

[19] Bright, P. Spamhaus ddos grows to internet-threatening size. *ArsTechnica, March* (2013).

[20] Bu, T., Cao, J., Chen, A., and Lee, P. P. Sequential hashing: A flexible approach for unveiling significant patterns in high speed networks. *Computer Networks 54*, 18 (2010), 3309–3326.

[21] Caesar, M., Casado, M., Koponen, T., Rexford, J., and Shenker, S. Dynamic route recomputation considered harmful. *SIGCOMM CCR 40*, 2 (2010), 66–71.

[22] CAIDA. The caida anonymized internet traces 2016 dataset, 2016.

[23] Chen, H., and Benson, T. The case for making tight control plane latency guarantees in sdn switches. In *SOSR* (2017), pp. 150–156.

[24] Claise, B., Sadasivan, G., Valluri, V., and Djernaes, M. Rfc 3954: Cisco systems netflow services export version 9 (2004). *Retreived online: http://www. ietf. org/rfc/rfc3954. txt* (2007).

[25] Communications, S. Spirent testcenter., 2017.

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

[27] Cormode, G., and Muthukrishnan, S. What's new: Finding significant differences in network data streams. *IEEE/ACM Transactions on Networking (TON) 13*, 6 (2005), 1219–1232.

[28] Curtis, A. R., Mogul, J. C., Tourrilhes, J., Yalagandula, P., Sharma, P., and Banerjee, S. Devoflow: Scaling flow management for high-performance networks. In *SIGCOMM* (2011), vol. 41, pp. 254–265.

[29] Dixit, A., Prakash, P., Hu, Y. C., and Kompella, R. R. On the impact of packet spraying in data center networks. In *INFOCOM* (2013), pp. 2130–2138.

[30] Fayazbakhsh, S. K., Chiang, L., Sekar, V., Yu, M., and Mogul, J. C. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *NSDI* (2014).

[31] Feldmann, A., Greenberg, A., Lund, C., Reingold, N., Rexford, J., and True, F. Deriving traffic demands for operational ip networks: Methodology and experience. *IEEE/ACM Transactions on Networking (ToN) 9*, 3 (2001), 265–280.

[32] Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A., Chung, E., et al. Azure accelerated networking: Smartnics in the public cloud. In *NSDI* (2018).

[33] Flajolet, P., and Martin, G. N. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences 31*, 2 (1985), 182–209.

[34] Floyd, S., and Jacobson, V. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking (ToN) 1*, 4 (1993), 397–413.

[35] Garcia-Teodoro, P., Diaz-Verdejo, J., Maciá-Fernández, G., and Vázquez, E. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security 28*, 1 (2009), 18–28.

[36] Ghorbani, S., Yang, Z., Godfrey, P., Ganjali, Y., and Firoozshahian, A. Drill: Micro load balancing for low-latency data center networks. In *SIGCOMM* (2017), pp. 225–238.

[37] Goel, A., and Gupta, P. Small subset queries and bloom filters using ternary associative memories, with applications. *SIGMETRICS Performance Evaluation Review 38*, 1 (2010), 143–154.

[38] Gupta, A., Birkner, R., Canini, M., Feamster, N., Mac-Stoker, C., and Willinger, W. Network monitoring as a streaming analytics problem. In *HotNets* (2016), pp. 106–112.

[39] Gupta, A., Harrison, R., Pawar, A., Canini, M., Feamster, N., Rexford, J., and Willinger, W. Sonata: Query-driven streaming network telemetry. In *Proceedings of the ACM SIGCOMM 2018 conference (SIGCOMM'18)* (2018), ACM.

[40] Hansen, P. Methods of nonlinear 0-1 programming. *Annals of Discrete Mathematics 5* (1979), 53–70.

[41] Hari, A., Lakshman, T., and Wilfong, G. Path switching: Reduced-state flow handling in sdn using path information. In *CoNEXT* (2015), p. 36.

[42] Harrison, R., Cai, Q., Gupta, A., and Rexford, J. Network-wide heavy hitter detection with commodity switches. In *SOSR* (2018).

[43] He, K., Khalid, J., Gember-Jacobson, A., Das, S., Prakash, C., Akella, A., Li, L. E., and Thottan, M. Measuring control plane latency in sdn-enabled switches. In *SOSR* (2015), p. 25.

[44] Hintjens, P. Zeromq: The guide. *URL http://zeromq. org* (2010).

[45] Hopps, C. E. Analysis of an equal-cost multi-path algorithm.

[46] Huang, D. Y., Yocum, K., and Snoeren, A. C. High-fidelity switch models for software-defined network emulation. In *HotSDN* (2013), pp. 43–48.

[47] Huang, Q., Jin, X., Lee, P. P., Li, R., Tang, L., Chen, Y.-C., and Zhang, G. Sketchvisor: Robust network measurement for software packet processing. In *SIGCOMM* (2017), pp. 113–126.

[48] Jepsen, T., Moshref, M., Carzaniga, A., Foster, N., and Soulé, R. Life in the fast lane: A line-rate linear road. In *SOSR* (2018).

[49] Jose, L., Yan, L., Varghese, G., and McKeown, N. Compiling packet programs to reconfigurable switches. In *NSDI* (2015), pp. 103–115.

[50] Jung, J., Paxson, V., Berger, A. W., and Balakrishnan, H. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy* (2004), pp. 211–225.

[51] Kang, N., Liu, Z., Rexford, J., and Walker, D. Optimizing the one big switch abstraction in software-defined networks. In *CoNEXT* (2013), pp. 13–24.

[52] Katsikas, G. P., Barbette, T., Kostic, D., Steinert, R., and Maguire Jr, G. Q. Metron: Nfv service chains at the true speed of the underlying hardware. In *NSDI* (2018).

[53] Katta, N., Hira, M., Kim, C., Sivaraman, A., and Rexford, J. Hula: Scalable load balancing using programmable data planes. In *SOSR* (2016), p. 10.

[54] Kazemian, P., Varghese, G., and McKeown, N. Header space analysis: Static checking for networks. In *NSDI* (2012), vol. 12, pp. 113–126.

[55] Kim, C., Sivaraman, A., Katta, N., Bas, A., Dixit, A., and Wobker, L. J. In-band network telemetry via programmable dataplanes. In *SOSR* (2015).

[56] Knight, S., Nguyen, H. X., Falkner, N., Bowden, R., and Roughan, M. The internet topology zoo. *IEEE Journal on Selected Areas in Communications 29*, 9 (2011), 1765–1775.

[57] Kührer, M., Hupperich, T., Rossow, C., and Holz, T. Exit from hell? reducing the impact of amplification ddos attacks. In *USENIX Security Symposium* (2014), pp. 111–125.

[58] Kumar, A., Sung, M., Xu, J. J., and Wang, J. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *SIGMETRICS Performance Evaluation Review* (2004), vol. 32, pp. 177–188.

[59] Lazaris, A., Tahara, D., Huang, X., Li, E., Voellmy, A., Yang, Y. R., and Yu, M. Tango: Simplifying sdn control with automatic switch property inference, abstraction, and optimization. In *CoNEXT* (2014), pp. 199–212.

[60] Li, Y., Miao, R., Kim, C., and Yu, M. Flowradar: A better netflow for data centers. In *NSDI* (2016), pp. 311–324.

[61] Li, Y., Miao, R., Kim, C., and Yu, M. Lossradar: Fast detection of lost packets in data center networks. In *CoNEXT* (2016), pp. 481–495.

[62] Lindo. Lingo and optimization modelling, 2016.

[63] Liu, Z., Manousis, A., Vorsanger, G., Sekar, V., and Braverman, V. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *SIGCOMM* (2016), pp. 101–114.

[64] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. Openflow: enabling innovation in campus networks. *SIGCOMM CCR 38*, 2 (2008), 69–74.

[65] Miao, R., Zeng, H., Kim, C., Lee, J., and Yu, M. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *SIGCOMM* (2017), pp. 15–28.

[66] Monsanto, C., Reich, J., Foster, N., Rexford, J., Walker, D., et al. Composing software defined networks. In *NSDI* (2013), vol. 13, pp. 1–13.

[67] Moshref, M., Bhargava, A., Gupta, A., Yu, M., and Govindan, R. Flow-level state transition as a new switch primitive for sdn. In *HotSDN* (2014).

[68] Moshref, M., Yu, M., Govindan, R., and Vahdat, A. Dream: dynamic resource allocation for software-defined measurement. *SIGCOMM 44*, 4 (2015), 419–430.

[69] Moshref, M., Yu, M., Govindan, R., and Vahdat, A. Scream: Sketch resource allocation for software-defined measurement. In *CoNEXT* (2015), p. 14.

[70] Moshref, M., Yu, M., Govindan, R., and Vahdat, A. Trumpet: Timely and precise triggers in data centers. In *SIGCOMM* (2016), pp. 129–143.

[71] Moshref, M., Yu, M., Sharma, A. B., and Govindan, R. Scalable rule management for data centers. In *NSDI* (2013), vol. 13, pp. 157–170.

[72] Narayana, S., Sivaraman, A., Nathan, V., Goyal, P., Arun, V., Alizadeh, M., Jeyakumar, V., and Kim, C. Language-directed hardware design for network performance monitoring. In *SIGCOMM* (2017), pp. 85–98.

[73] Networks, B. Barefoot: The world's fastest and most programmable networks, 2017.

[74] Oppenheim, A., Willsky, A., and Nawab, S. *Signals and Systems*. Prentice-Hall signal processing series. Prentice Hall, 1997.

[75] Perry, J., Ousterhout, A., Balakrishnan, H., Shah, D., and Fugal, H. Fastpass: A centralized zero-queue datacenter network. *SIGCOMM 44*, 4 (2015), 307–318.

[76] Pfaff, B., Pettit, J., Koponen, T., Jackson, E. J., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., et al. The design and implementation of open vswitch. In *NSDI* (2015), pp. 117–130.

[77] Rasley, J., Stephens, B., Dixon, C., Rozner, E., Felter, W., Agarwal, K., Carter, J., and Fonseca, R. Planck: Millisecond-scale monitoring and control for commodity networks. In *SIGCOMM* (2014), vol. 44, pp. 407–418.

[78] Rossow, C. Amplification hell: Revisiting network protocols for ddos abuse. In *NDSS* (2014).

[79] Roy, A., Zeng, H., Bagga, J., Porter, G., and Snoeren, A. C. Inside the social network's (datacenter) network. In *SIGCOMM* (2015), vol. 45, pp. 123–137.

[80] Schweller, R., Gupta, A., Parsons, E., and Chen, Y. Reversible sketches for efficient and accurate change detection over network data streams. In *IMC* (2004), pp. 207–212.

[81] Sekar, V., Reiter, M. K., Willinger, W., Zhang, H., Kompella, R. R., and Andersen, D. G. csamp: A system for network-wide flow monitoring. In *NSDI* (2008), vol. 8, pp. 233–246.

[82] Sen, S., Shue, D., Ihm, S., and Freedman, M. J. Scalable, optimal flow routing in datacenters via local link balancing. In *CoNEXT* (2013).

[83] Shalimov, A., Zuikov, D., Zimarina, D., Pashkov, V., and Smeliansky, R. Advanced study of sdn/openflow controllers. In *central & eastern european software engineering conference in russia* (2013), p. 1.

[84] Sharma, N. K., Kaufmann, A., Anderson, T. E., Krishnamurthy, A., Nelson, J., and Peter, S. Evaluating the power of flexible packet processing for network resource allocation. In *NSDI* (2017), pp. 67–82.

[85] Shin, S., Yegneswaran, V., Porras, P., and Gu, G. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *SIGSAC CCS* (2013), pp. 413–424.

[86] Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., and Rexford, J. Heavy-hitter detection entirely in the data plane.

In *SOSR* (2017), pp. 164–176.

[87] Soulé, R., Basu, S., Marandi, P. J., Pedone, F., Kleinberg, R., Sirer, E. G., and Foster, N. Merlin: A language for provisioning network resources. In *CoNEXT* (2014), pp. 213–226.

[88] Sun, C., Bi, J., Chen, H., Hu, H., Zheng, Z., Zhu, S., and Wu, C. Sdpa: Toward a stateful data plane in software-defined networking. *IEEE/ACM Transactions on Networking* (2017).

[89] Thimmaraju, K., Shastry, B., Fiebig, T., Hetzelt, F., Seifert, J.-P., Feldmann, A., and Schmid, S. Taking control of sdn-based cloud systems via the data plane. In *SOSR* (2018).

[90] Vanini, E., Pan, R., Alizadeh, M., Taheri, P., and Edsall, T. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *NSDI* (2017), pp. 407–420.

[91] Venkataraman, S., Song, D. X., Gibbons, P. B., and Blum, A. New streaming algorithms for fast detection of superspreaders. In *NDSS* (2005), pp. 149–166.

[92] Wang, M., Li, B., and Li, Z. sflow: Towards resource-efficient and agile service federation in service overlay networks. In *ICDCS* (2004), pp. 628–635.

[93] Weisstein, E. W. Sifting property. http://mathworld.wolfram.com/siftingproperty.html. In *MathWorld–A Wolfram Web Resource.*

[94] Yu, M., Jose, L., and Miao, R. Software defined traffic measurement with opensketch. In *NSDI* (2013), vol. 13, pp. 29–42.

[95] Yuan, Y., Lin, D., Mishra, A., Marwaha, S., Alur, R., and Loo, B. T. Quantitative network monitoring with netqre. In *SIGCOMM* (2017), pp. 99–112.

[96] Zhang, J., Ren, F., and Lin, C. Modeling and understanding tcp incast in data center networks. In *INFOCOM* (2011), pp. 1377–1385.

[97] Zhang, Q., Liu, V., and Zeng, H. High-resolution measurement of data center microbursts.

[98] Zhang, Y. An adaptive flow counting method for anomaly detection in sdn. In *CoNEXT* (2013), pp. 25–30.

[99] Zhou, J., Tewari, M., Zhu, M., Kabbani, A., Poutievski, L., Singh, A., and Vahdat, A. Wcmp: Weighted cost multipathing for improved fairness in data centers. In *European Conference on Computer Systems* (2014), p. 5.

A Linearization of 0-1 NLP

To linearize the original problem into a 0-1 ILP problem, we introduce a series of auxiliary variables, $y^{(j,t)} \in \{0,1\}$, *s.t.*:

$$\forall i,p,\ y^{(j,t)} \geqslant \delta_{i,j} \cdot \sum_{\tau=1}^{\min\{CS^{(i,p)},t\}} x^{(i,p)}_{(j,t-\tau+1)} \tag{5}$$

We then have the following proposition:

Proposition 1. The objective function in Figure 6 is equivalent to:

$$\min \sum_{j\in\mathcal{S}} \sum_{t=1}^{SN-1} y^{(j,t)},\ i.e.\ y^{(j,t)} = sgn\left(occupy^{(j,t)}\right) \tag{6}$$

Proof. We denote the right side of Equation 5 as $\mathcal{R}^{(j,t)}_{(i,p)}$. From the constraint (C4) in Table 6, we have:

$$\mathcal{R}^{(j,t)}_{(i,p)} \leqslant \sum_{t=1}^{SN-1} x^{(i,p)}_{(j,t)} \leqslant \sum_{j\in\mathcal{S}} \sum_{t=1}^{SN-1} x^{(i,p)}_{(j,t)} = 1 \tag{7}$$

Thus $\forall j,t,\ \mathcal{R}^{(j,t)}_{(i,p)} \in \{0,1\}$. We demonstrate the proposition above by discussing the value of for a certain $\{j_0,t_0\}$.

Case 1: If $\exists i_0,p_0,\ s.t.\ \mathcal{R}^{(j_0,t_0)}_{(i_0,p_0)} = 1$, from Equation 5, we have

$$1 \geqslant y^{(j_0,t_0)} \geqslant \mathcal{R}^{(j_0,t_0)}_{(i_0,p_0)} = 1 \tag{8}$$

At this time, $occupy^{(j_0,t_0)} \geqslant \mathcal{R}^{(j_0,t_0)}_{(i_0,p_0)} = 1$. Therefore

$$sgn\left(occupy^{(j_0,t_0)}\right) = 1 = y^{(j,t)} \tag{9}$$

Case 2: If $\forall i,p,\ \mathcal{R}^{(j_0,t_0)}_{(i,p)} = 0$. In this case, $y^{(j_0,t_0)}$ is feasible for both 0 and 1. However, as the optimization objective is minimize the sum of $y^{(j_0,t_0)}$, $y^{(j_0,t_0)}$ will be equal to 0 in the optimal solution. Meanwhile, we have:

$$occupy^{(j_0,t_0)} = \sum_{i\in\mathcal{T}} \left(\delta_{i,j_0} \cdot \sum_{p=1}^{sub_i} \left(\sum_{\tau=1}^{\min\{CS^{(i,p)},t_0\}} x^{(i,p)}_{(j_0,t_0-\tau+1)} \right) \right) = 0 \tag{10}$$

Thus we demonstrate **Proposition 1** and linearize the programming model in Table 6. □