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

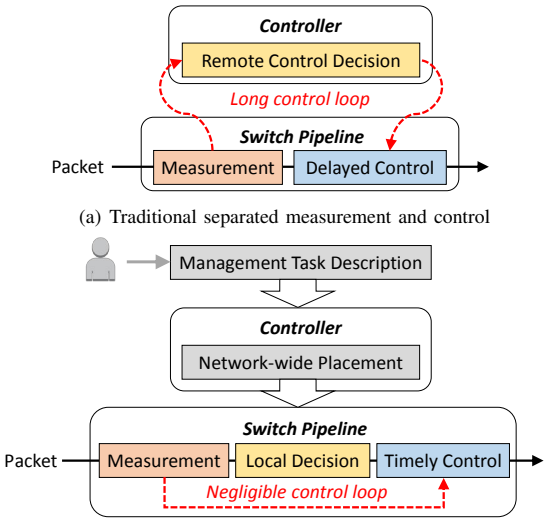
Shuhe Wang[1], Chen Sun[1,2], Zili Meng[1], Minhu Wang[1], Jiamin Cao[1], Mingwei Xu[1], Jun Bi[1],
Qun Huang[3], Masoud Moshref[4], Tong Yang[3], Hongxin Hu[5], Gong Zhang[6]
[1]Institute for Network Sciences and Cyberspace, BNRist, Tsinghua University
[2]Alibaba Group, [3]Peking University, [4]Barefoot Networks, [5]Clemson University, [6]Huawei Technologies
{{wangshuh18, wangmh19, cjm18}@mails., {xumw, junbi}@}tsinghua.edu.cn,
zilim@ieee.org, qichen.sc@alibaba-inc.com, huangqun@pku.edu.cn,
mmoshref@google.com, yangtongemail@gmail.com, hongxih@clemson.edu, nicholas.zhang@huawei.com

***Abstract*—Advanced network management systems, including network measurement and traffic control, 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 switch to switch-local controller, the latency is still tens of milliseconds and 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 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.**

I. Introduction

Network management in modern networks encompasses a range of tasks including anomaly detection [59], [91], attack defense [66], [87], and flow scheduling [2], [12], [30]. 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) [45], [66], [87]. Many efforts have been devoted *separately* to network measurement and control. Literatures including OpenSketch [87], Dream [64], UnivMon [59], Trumpet [66], SketchVisor [45], Sonata [37] and more focused on improving the performance, accuracy, and resource efficiency of measurement, while Pyretic [62], FlowTags [29], FlexSwitch [80] and so on focused on declaration and execution of control actions.

Actually, there exists a *gap* between measurement and control, as the choices of control actions are based on measurement. Advanced management systems [56], [59], [66], [73], [87] employ a dedicated controller to connect the two phases. As shown in Figure 1(a), the measurement module collects statistics and reports to the remote controller, which 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 to make decisions in controller. Our experiments in §VI show that this loop ranges from seconds to minutes. Also, using the switch-local control plane to process the data plane events could only reduce part of the transmission latency.



(a) Traditional separated measurement and control

(b) Martini supports timely control in switches

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

Unfortunately, such a long control loop is unacceptable for many latency-sensitive management tasks. For instance, an advanced congestion control mechanism [33] 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 [11], [33], [90]. If we rely on the controller for control decision, a microburst may have already caused congestion before the new rule reaches the switch. Such untimeliness also prevents support for many other management tasks including DNS reflection attack defense [37], [83], superspreader identification and quarantine [10], fast link failure recovery [20], *etc.*

To reduce the control loop, a basic idea, contrary to SDN, is making 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

978-1-7281-6992-7/20/$31.00 ©2020 IEEE

studies propose to enhance the programmability of *OpenFlow switches* to support network tasks [13], [27], [33], [50], [63], [81], [83]. But they requires redeveloping OpenFlow Switching ASIC to support new tasks, which introduces high cost and long development cycles [17]. (2) Some recent studies exploit *programmable switches* [17] to offload network tasks. (2.1) [82], [66] 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) [5], [49], [61] 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 [80] took a first step towards providing a library of reusable modules. However, it only provides building blocks related to resource allocation protocols. In summary, due to the high diversity and complexity of management tasks, each of existing solutions is tailored for one specific task or requires system redesign 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, 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 [78] measures flow sizes, detects burst flows, and performs local load balancing in switches to avoid congestion. We provide more examples in §II-A and Table VII. 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*. Since measurement and control already reside in the switch data plane, offloading control decision merely occupies very few additional resources (§IV-A). The Martini framework includes three components. First, Martini provides a set of high-level *management task description primitives* for operators to easily describe and assemble the above all three phases of management. Second, Martini presents an innovative *network-wide task placement* mechanism to exploit resources of all network switches to accommodate massive management tasks. Finally, Martini provides a library of measurement and control components and a compiler that automatically composes them to generate programs on programmable switching ASICs.

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. (§II)
- We provide a set of *task description primitives* for operators to easily describe the measurement, local control decision, and control actions in management tasks. (§III)
- We design a *network-wide task placement* mechanism that deploys management tasks onto all switches in the entire network with high resource efficiency. (§IV)
- 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. (§V)
- 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. (§VI)

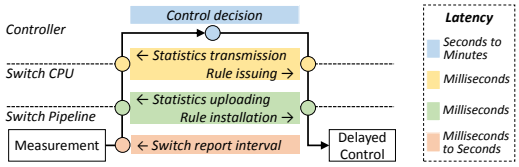


Fig. 2: The latency components of the control loop.

II. Motivations and Challenges

We first motivate from the observation that today's network management systems, requiring timely control, suffer from a long control loop. Then, we introduce three major design challenges including management task description, network-wide task placement, and task implementation in switches.

A. Motivations

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:* Statistics are *periodically* reported to the controller. For example, SNMP provides per port counters every few *minutes* [66]. OpenFlow reports flow counters every few *seconds* [2], [27], [72]. A network event cannot be reported until the end of its measurement interval.
- *Statistics uploading and rule installation:* The switch pipeline first sends statistics to report to the switch CPU, which then communicates with the controller. Returned flow rules follow the inverse path. The latency between pipeline and CPU could reach tens of milliseconds and even more under high load [22], [41], [44], [55], [64].
- *Statistics transmission and rule issuing:* Statistics and control rules are transmitted between the switch CPU and the controller through communication APIs like OpenFlow [64], [87]. However, it takes $50\mu s$ to $3ms$ for an OpenFlow switch to receive a message from the one-hop-away controller [79].
- *Control decision in the controller:* Many monitoring systems use *sketches* for measurement [45], [59], [65], [87] as they can provide theoretical guarantees on error bound with limited memory in switches. However, in order to reduce the communication latency and save bandwidth, switches can only report counters without large flow keys to the controller [26], [45], [59], [87]. The controller then exploits techniques such as reversible sketch [76], [87], group testing [59], or sequential hashing [19] to retrieve the flow keys for control decision. Our evaluation in §VI-B shows that this process could take *seconds* to *minutes*.

In summary, traditional centralized control plane for control decision introduces a long control loop. This latency may increase as networks grow in capacity and utilization [66].

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 VII.

- *Microburst detection and fast load balancing:* Some recent works exploit the control plane for congestion control [2], [12], [71]. However, congestion instead increases the communication latency between switches and controller. Slow-reacting load balancing may then only be able to effectively resolve congestion at the coarser timescale of $1s$ [90], while the majority of congestion is caused by microbursts with a few μs life cycle [11], [33], [75], [90]. Therefore, we should quickly identify bursty flows that could cause microbursts and ensure timely load balancing within several μs [78].
- *DNS reflection attack detection and defense:* 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 [74], [35]. Such an attack could grow to a threatening size of 300Gbps [18]. If the target network is executing a crucial and timely task, tradition mitigation that take seconds to take effect may be too slow to stop failing the task.
- *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 [37], [87]. Untimely controlled superspreaders could cause serious fast worm propagation [85], thus quickly identifying and quarantining them is highly important.

Above cases focus on handling violent situations in short time-scales. Despite they may also work at coarser time-scale (e.g. central control decision), a much quicker reaction can work together with traditional methods and optimize the overall management timeliness. Moreover, controlling at finer time-scale is a new option to capture and deal with many situations at their early stages, which can also save lots of global processing resources such as CPU and bandwidth.

Martini: To achieve such fine time-scale management, Martini proposes a scheme that performs timely control based on local measurement results in switches. Therefore, 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.

Generality: Martini can abstract the common tasks into one unified scheme and put them in a library for users to reuse them without reinventing the wheel. Any new task that follows the scheme can be added to the library. Also, since the network resources are limited, a general framework is far more economical than multiple specific frameworks.

Benefiting global management tasks: Although some other tasks require global information from multiple switches for control decision and cannot follow such a scheme, such as detecting network-wide heavy hitters [40], Martini can benefit them by reporting to the controller after event detection. The global task in the controller can then gather information for control. These tasks do not require local control decisions and are therefore out of the scope of Martini. Nevertheless, Martini is still helpful for the description, placement, and implementation of measurement and control parts of them.

TABLE I: Martini measurement primitives and comparison with Marple [68], Sonata [37].

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.	×	×

B. Design Challenges

Management task description: Recent studies have proposed languages for either *measurement* [37], [66], [68] or *control* [8], [60], [62], [80], but have not considered them collectively. However, we are challenged to give descriptions for the contents and connections of all phases in one management task. Furthermore, we require high-level primitives to enable code reuse to reduce operational burdens. To this end, We proposes a set of high-level task description primitives to give all phases of management tasks a unified description. (§III).

Network-wide task placement: Martini aims to accommodate all phases of managements tasks in switches. However, switching ASICs have limited resources [17], which constrains accommodating massive management tasks. To address this challenge, we use highly resource-efficient *sketches* for measurement and provide a balanced and configurable trade-off between resource usage and accuracy. Then we 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, and design a network-wide task placement algorithm that places subtasks on network switches to optimize global resource usage. (§IV).

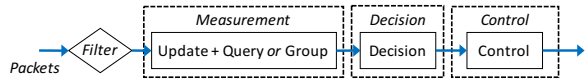
Task implementation on switching ASICs: Implementation is challenging in two aspects. First, we are challenged to support various sketches for measurement and algorithms for control in switching ASICs. In response, Martini provides a rich component library for operators. Second, one management task may span across several switches, making it challenging to configure each switch and the information exchange among switches. In response, Martini automatically compiles configurations for all switches by composing components in the code library based on task placement results. (§V).

III. Management Task Description

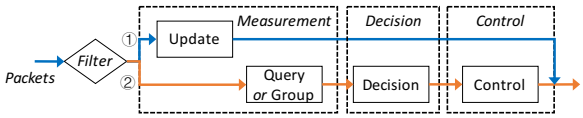
We define a set of primitives for operators to easily describe management phases. The primitives are designed to be *modular*, *reusable*, and *extensible* so that each phase can be declared separately and composed together easily. Then we present an example management task using our primitives.

TABLE II: Partial control actions supported by 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*]) [43], [80]	Per-flow load balancing across a list of ports [*p*].
wcmp ([{*p*, *w*}]) [92], [84]	Per-flow balancing across a list of weighted ports. Each port *p* is assigned a forwarding weight *w*.
spray ([*p*]) [28]	Per-packet load balancing across a list of ports [*p*].
flowlet_ecmp (*i*, [*p*]) [5], [84]	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 () [6]	Tag the ECN bits of a packet once a queue is congested.
qcn (*th*, *p*) [4]	Tag pkts with probability *p* if queue congestion exceeds *th*.
hull (*t*) [7]	Implement phantom queues with threshold *t*.
drill ([*q*]) [33]	Per-packet balancing based on lengths of queues [*q*].
localflow ([q]) [78]	Balance flows into a list of equal cost queues [*q*]
set_queue (*qid*)	Insert a packet into the queue with ID *qid*.
red (*p*) [32]	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.



(a) Traditional *measure through* scheme



(b) Additional *update-query* scheme supported by Martini

Fig. 3: Measurement schemes in Martini.

A. Martini Primitives

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

Measurement: There are two unique features in measurement: (1) *Decoupling statistics updating and querying.* We observe that traditional measurement primitives follow a *measure through* scheme. 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 [66], [37]. However, coupling statistics updating and querying may not be able to support all tasks. For the fine-grained DNS reflection attack defense task (§II-A), we should maintain records when receiving DNS requests, and query the record and output the existence of the corresponding request when receive DNS responses. Thus updating and querying are triggered by different packets, which cannot be supported by 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 I. Operators could `filter` one set of packets to `update` the record and a different set to `query` or `group` the record to generate output. Finally, we design the `set_window` primitive to enable operators to configure the measurement window *t*.

(2) *Resource-aware measurement description.* Measurement is often memory-intensive [82], [37]. Accurately estimating the resource usage of measurement tasks is important for placement. However, existing measurement primitives focus solely on functional declaration. Martini allows operators to input estimated maximum `error_rate`, `flow_number`, and `flow_rate`. The memory consumption of tasks can be automatically inferred based on above values (§IV-A). To gain accurate resource parameters, operators could deploy a simple measurement task to gather accurate flow statistics during runtime [80]. If flow statistics change, operators could simply adjust resource allocation and deploy a new measurement task.

```
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)
8       .drop().report_to_controller()
```

Fig. 4: DNS reflection attack detection and defense.

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. Detections are expressed 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.*.

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 customized control actions implemented in the programmable data plane 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 II.

B. Example Management Tasks

Operators compose primitives for management tasks based on the *one-big-switch abstraction* [9], [59] as if deployed entirely in one single switch. We describe the DNS reflection attack defense task here, and more examples are in Table VII.

Primitive integration. We start by introducing how to integrate primitives of all phases into management tasks. Regarding measurement and decision primitives, a control decision could be made based on one or multiple measurement results. Meanwhile, multiple control actions can be assigned to one event, while different events could result in the same control actions. We use an intuitive *python*-like padding style to denote the conjunction of measurement phase and decision phase, as well as the conjunction of events and corresponding actions.

DNS reflection attack detection and defense. As illustrated in Figure 4, we filter DNS requests (line 3), use source IP, destination IP, and DNS transaction ID as index, and record their existence by setting corresponding counters to 1 (line 4). For DNS responses, we query the existence of corresponding requests (line 5, 6). If they does not exist, the responses will be dropped and reported to the controller (line 7, 8).

IV. Network-wide Task Placement

Controller inputs the task descriptions and places tasks on network-wide switches. Existing placement solutions either

TABLE III: 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* [25]	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* [31]	Relative error: ϵ	$(\frac{0.78}{\epsilon})^2 \times (32 - 2\lfloor \log_2 \frac{0.78}{\epsilon} \rfloor)$	6
			query	DNS reflection attack defense	*Bloom filter* [14], [34]	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* [26], [76]	Threshold: $0 < \phi < 1$ Per-flow relative error: ϵ	$\frac{512}{\epsilon^2\phi^2}$	2
distribution ()				Flow size distribution	*multi-resolution sketch* [54]		34M	4

TABLE IV: 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

focus on placement on a single switch [36] or do not consider the intrinsic constraints of measurement based tasks [77] and the switches capacities [9]. We discuss more details in §VII. In comparison, 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.

A. 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. Martini is based on pipelined switches [17], [37], with critical resources including pipeline stages (1-32), SRAM (tens to hundreds of Mb), and stateful actions. However, for non-pipelined switches, the concept of "stage" can still be generalized to node in the network and Martini will still be applicable.

Measurement: Recent studies have proposed many techniques to support measurement inside switches, such as sketches [45], [59], [65], [87], hash tables [3], [82], sampling [23], [77], [86], *etc.* We select sketches due to their *high resource efficiency* and *bounded error* are suitable for limited switch resources. As listed in Table III, we first *map* the description of the measurement phase into its corresponding sketch. Then we calculate the SRAM usage of the sketch based on the input or default resource parameters. We also collect the stage usage of each sketch based on our implementation (§V).

Decision: Since the control decision phase is a single conditional expression, it only occupies one pipeline stage and negligible SRAM resources for network events detection.

Control: Based on our implementation (§V), we found that the pipeline stage is the main critical resource of control actions in Table II, and list the consumption in Table IV. Each control action requires negligible SRAM to hold control rules.

B. Computation and Data Partition

As shown in Tables III and IV, measurement could take lots of SRAM resources, while both measurement and control phases may consume many pipeline stages. Therefore, it is infeasible for some 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 split according to its execution phases as in (§III). We could distribute the three phases onto different switches while ensuring their *ordering*. However, such partition introduces extra information transmission among switches and may affect the goodputs. We notice that the measurement phase generates *statistics* (hundreds of bits) for decision, while the decision phase detects *network events* and informs the control phase with a single flag(of a few bits). As the decision phase merely occupies one pipeline stage and few SRAM, we *couple measurement and decision* and simply partition a task into *measurement* and *control* for minimum overhead. Moreover, multiple control phases in one task can also be partitioned.

Data partition: According to Table III, measurement is usually SRAM intensive. For instance, 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* into several partitions, and use multiple instances of the same sketch to monitor the flow partitions. We minimize the number of partitions while ensuring one stage has enough resources to measure one partition to minimize stage usage.

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.

C. Network-wide Placement Algorithm

Given the subtasks obtained above, we model their placement as a 0-1 Nonlinear Programming problem in Table VI.

TABLE V: 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)
$\mathcal{C}$	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 VI: Formulation for network-wide 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) $$\forall (p_1 \trianglelefteq p_2, i) \in \mathcal{C}: \sum_{j,t} \left(\rho_i^{(j,t)} \cdot x_{(j,t)}^{(i,p_1)}\right) + CS^{(i,p_1)} \leqslant \sum_{j,t} \left(\rho_i^{(j,t)} \cdot x_{(j,t)}^{(i,p_2)}\right)$$
where $\rho_i^{(j,t)} = (\alpha_{i,j}-1)\cdot SN + t$

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

We list related notations in Table V. We further linearize the objective into an Integer Linear Programming problem (ILP) , as there exist many advanced ILP tools to accelerate problem solving. Detailed modeling process is shown below.

Objective. Our intuition is that management tasks should occupy minimal switch resources. However, a switch has *multiple types of mutually dependent resources*. Each stage has fixed amount of SRAM and stateful actions [17], [37]. The problem is which type of resources to optimize. We notice that the total SRAM and stateful actions usage 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. (j,t) is occupied by subtask p if the first stage of p is on switch j and less than $CS^{(i,p)}$ stages ahead of (j,t), *i.e.*

$$\exists \tau \in \{1,2,...,\min\{CS^{(i,p)},t\}\},\ s.t.\ x_{(j,t-\tau+1)}^{(i,p)} = 1 \quad (1)$$

we traverse all subtasks and sum the formula above up as $occupy^{(j,t)}$. Therefore $occupy^{(j,t)} > 0$ indicates that (j,t) is occupied. We use sgn function to normalize *occupy* to $\{0,1\}$, where $sgn(x) = 1\ if\ x > 0$, and 0 otherwise. The final expression of the objective is shown in Table VI.

Resource Constraints (C1, C2). Similar to [46], we analyze SRAM and stateful actions constraints in the granularity of a *stage*. As one subtask may span across multiple stages, the SRAM for subtask p in task i on stage (j,t) is:

$$\sum_{\tau=1}^{\min\{CS^{(i,p)},t\}} x_{(j,t-\tau+1)}^{(i,p)} \cdot Mem_\tau^{(i,p)} \quad (2)$$

The formula is the *convolution* of $x_{(j,t)}^{(i,p)}$ and $Mem_t^{(i,p)}$ [70], 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 constraint of stateful actions (C2).

Ordering Constraints (C3). We should ensure the ordering between subsequent control subtasks. In $path_i$, one subtask p_1 placed *before* another subtask p_2, is denoted as $(i, p_1 \trianglelefteq p_2)$. It requires that the start stage of p_1, (j_1,t_1), should be more than $CS^{(i,p_1)}$ stages ahead of the start stage of p_2, (j_2,t_2). We denote $\rho_i^{(j,t)}$ as the absolute order of stage (j,t), calculated as $\rho_i^{(j,t)} = (\alpha_{i,j}-1)\cdot SN + t$, with $\alpha_{i,j}$ as the position of switch j on $path_i$. The absolute order of (j_1,t_1) naturally equals to $\rho_i^{(j_1,t_1)} \cdot x_{(j_1,t_1)}^{(i,p_1)}$. Since $x_{(j,t)}^{(i,p_1)}$ turns into 1 only in (j_1,t_1), and stays 0 throughout all the other stages, it establishes that

$$\rho_i^{(j_1,t_1)} \cdot x_{(j_1,t_1)}^{(i,p_1)} = \sum_{j,t} \left(\rho_i^{(j,t)} \cdot x_{(j,t)}^{(i,p_1)}\right) \quad (3)$$

This holds true for p_2 as well. So we replace the relation of p_1 and p_2 with the above expression and get constraints (C3).

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

Linearization of 0-1 NLP. To further 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.*:

$$\forall i,p,\ y^{(j,t)} \geqslant \delta_{i,j} \cdot \sum_{\tau=1}^{\min\{CS^{(i,p)},t\}} x_{(j,t-\tau+1)}^{(i,p)} \quad (4)$$

We then have the following proposition:

Proposition 1. The objective in Table VI is equivalent to:

$$\min \sum_{j\in\mathcal{S}} \sum_{t=1}^{SN-1} y^{(j,t)},\ i.e.\ y^{(j,t)} = sgn\left(occupy^{(j,t)}\right) \quad (5)$$

Summary: The above 0-1 ILP problem can be solved within limited time [38] in Martini. We evaluate the effectiveness and efficiency of the algorithm in §VI-C.

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 §VI-C.

Scalability and Limitations: The following aspects discuss the limitations in our placement algorithm, and how to address them to improve scalability:

Task placement constraints. First of all, some management tasks may not permit arbitrary partition and placement. For example, for control actions such as load balancing and scheduling, where to place them may make a big difference. Martini allows the user to add additional constraints on task locations, for example, the user can regulate that certain tasks must be placed close to the gateway, or at leaf switches.

Task Resource estimation. Our model requires prior knowledge about the tasks such as their routings and basic parameters such as flow numbers and flow rate. In real world settings, they might be highly variant and hard to predict. But operators could estimate traffic volume according to historical data [37]. Moreover, even if the traffic is larger than expected, only the measurement accuracy is affected without compromising Martinis availability, as long as traffic volume is within the capacity of switches.

Algorithm scalability. For large scale networks, the number of tasks will grow exponentially and ILP may be hard to scale. To solve it, we can only deploy partial tasks to the network each time, and incrementally deploy remaining tasks, which could leverage our fast incremental deployment algorithm. Moreover, Users can also specify additional constraints such as task locations to reduce the overall solution space complexity.

V. Implementation Details

We implement the Martini framework based on Barefoot Tofino [69], 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.

A. Component Library

We create the component library in P4 [16], with all sketches introduced in Table III and control in Table IV. Two major challenges emerge through the implementation.

Support for the measurement window: Measurement functions often collect statistics with a time window [59], [87]. 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 *round based timing* mechanism shown in Figure 5. We set each round as a fixed-time interval, and counters are cleared across different rounds. In the switch, we store `cur_time` as the the start time of the current round, and `cur_round` as the current round. Once the difference between the ingress *timestamp* of a packet and `cur_time` exceeds the measurement window, `cur_round` and `cur_time` are updated. Each counter is associated with a separate "last-time-updated" `round`, and if it is below `cur_round`, the counter gets cleared and `round` is renewed to `cur_round`. We use an 8-bit field for *round*, thus for a 32-bit counter, the resource overhead is 25%. However, an 8-bit round covers at most 256 consecutive intervals. If the difference of the rounds of two consecutive packets is multiple

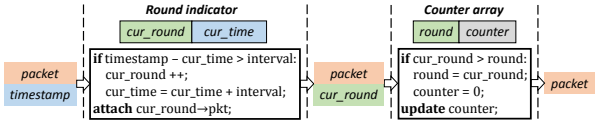


Fig. 5: Round based timing mechanism.

of 256, the `cur_round` for the two packets will be the same due to wrap-around. Then the corresponding counter will be incorrectly updated instead of reset. Nevertheless, evaluation in §VI-D shows that the error rate tops at 0.2% for real world traces, so this design is practical to use.

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 [17], [37], queue length information can only be obtained in egress, while queuing decisions are made only in ingress. To address this challenge, Martini *samples* packets in egress at a rate of 5% for each queue and *clones* the sampled packet as a probe. Martini *tags* the length of the queue on the probe and *recirculates* it to ingress. The probe provide queue lengths to ingress and *drops* after use. In this way, we could acquire queue lengths in ingress. As probes are recirculated using recirculation bandwidth in each pipeline and are dropped in ingress, the throughput will not be compromised. We demonstrate this in §VI-B.

B. 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. The compiler composes the codes of the subtasks according to their start stage and resource usage. We pay special attention to the following challenges during compilation.

Extra flow classification for data partition: Data partition requires to split flows on a path to several measurement subtasks, each of which corresponds to one part of the flows. To establish the correspondence, an extra flow classification is decided dynamically during task placement and cannot be predefined in the component library. To realize this goal, the compiler *reuses* the `filter` stage in each measurement subtask to identify the class of flows it should monitor, 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 [29], [39] 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.

VI. Evaluation

We evaluate the Martini framework on a testbed with two Barefoot Tofino switches(33 x 100 GbE), each of which is di-

TABLE VII: 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 [88]	IPs that receive more half-open TCP connections than threshold → drops later SYN packets	6	232
	2	Port scan [47]	IPs that send traffic to more than a threshold of destination ports → drops their packets	6	290
	3	DDoS victim [87]	IPs that receive traffic from more than a threshold of unique sources → performs RED on those traffic	6	322
	4	DNS reflection attack [53]	DNS responses without corresponding requests → drops the illegal requests	7	260
	5	NTP amplification attack [74]	IPs that receive NTP packets from more than a threshold of unique sources → drops these packets	7	291
	6	Stateful firewall [63]	Unsolicited inbound TCP connections without any outbound flows → drops the connections	8	245
Anomaly Detection	7	Superspreader [87]	IPs that contact more than a threshold of unique destinations → reports to the controller	6	303
	8	FTP monitoring [63]	FTP data channel setup requests when their control channels are not established → drops the requests	8	237
	9	Heavy changer [26]	Flows whose sizes have changed significantly across two intervals → reports to the controller	7	386
Flow Scheduling	10	Heavy hitter [59]	Flows whose size exceed threshold → performs per-flow port balancing	6	259
	11	Microburst [33]	Flows whose packet numbers within a window exceed a threshold → performs queue-based balancing	9	319
	12	PIAS [10]	Flow bytes sent exceed a threshold → inserts flow into a lower priority queue and ECN on congestion	8	207
	13	Video congestion control [15]	An I frame in an MPEG stream is dropped → drops later differentially-encoded B frames	7	296
	14	Link failure recovery [20]	Failure-carrying packets coming backward → uses backup routes for subsequent flows on this path	8	237
Network Monitoring	15	Flow size distribution [87]	The distribution of flow sizes → reports this information to the controller	7	240
	16	TCP incast [89]	IPs that receive TCP connections from more than a threshold of unique sources → informs controller	6	290

rectly connected with several 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 [21], and replay the traces at 100Gbps using the Spirent Test Center [24]. For test topology, we simulate real world topologies in Table VIII. Our evaluation goals are to:

- demonstrate the expressivity of the Martini description primitives to describe many management tasks. (§VI-A).
- 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. (§VI-B).
- demonstrate that the network-wide placement could achieve optimal resource usage for real world topologies and tasks within reasonable calculation time. (§VI-C).
- demonstrate that the error caused by the round based timing mechanism is tiny for real world traces. (§VI-D).
- demonstrate the capability of the compiler to generate codes for all network switches within little calculation time to deploy tasks on real world topologies. (§VI-E).

A. Expressivity

To demonstrate the expressivity of the Martini task description primitives, we specify sixteen common management tasks shown in Table VII. Among them, the DNS (#4) and FTP tasks (#8) follow the *update-query* scheme introduced in §III-A, while other tasks follow the *measure through* scheme. Tasks 2, 3, 5, 7, and 16 use the `groupby` 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.

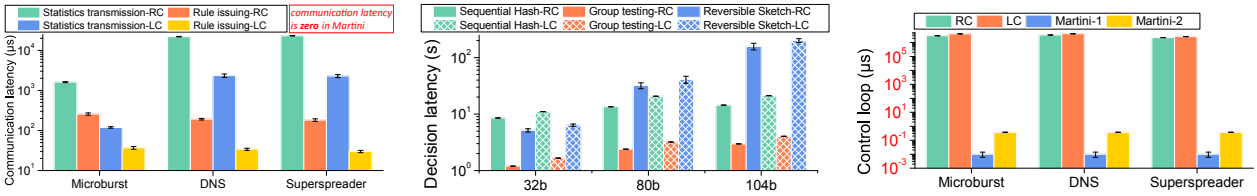
B. Control Loop Reduction

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 VII. 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 each components in the control loop including *switch report interval*, *statistics transmission* to the controller, *control decision*, and *rule issuing* to the switch. Note that the latency between switch CPU & hardware pipeline is a few *ms*.

We also implement a strawman approach that performs control decision in the *switch local CPU* to 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, we assume that computation partition breaks measurement and control into two switches. In this case, we deploy measurement and control in two directly connected Tofino switches. Packets get timestamped in first switch, and loop back from the second to the first. The first switch timestamps packets again and the control loop is calculated as half of the timestamp difference. *For each experiment, we report the average latency and the standard deviation across 100 runs.*

Communication latency: We use ZeroMQ [42] 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 III, 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 6(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. Local control in switch CPU could significantly reduce the communication latency by 90%, but the latency is still millisecond-level. 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 [19], group testing [26], and reversible sketch [45], [87]. 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



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

Fig. 6: 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).

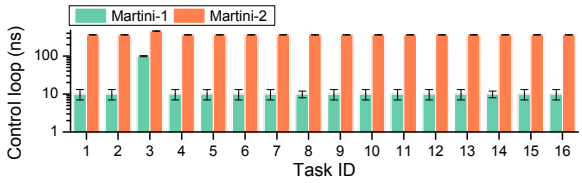


Fig. 7: Control loop of tasks in Table VII.

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 6(b) show that the calculation time rises as the key length increases. We also find that group testing takes the shortest time for 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.

Total control loop: Finally, we present the total control loop of the three example tasks in traditional and Martini patterns. For the former, we set the report interval of the microburst task as $100ms$ [2], and other tasks as $1s$ [37]. We use group testing for fast key deriving. We sum up all components of the control loop and present the results in Figure 6(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 VII in Martini. Results in Figure 7 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 egress, while other task control in ingress. But the latency is still tiny compared with traditional patterns.

High throughput maintenance: We evaluate the capability of Martini to maintain high throughput. We replay CAIDA traces at 100Gbps and measure the throughput of the tasks in above settings. Evaluation results show that all 16 tasks could maintain *line rate* of 100Gbps including the microburst task (#11) which requires packet cloning and recirculation. This demonstrates Martini's ability to maintain high throughput.

TABLE VIII: Topologies tested in our experiments

Topology	Description	Switch	Edge
Fat-tree [1]	Fat tree network with $k = 4$	20	32
AT&T [52]	AT&T North America backbone network	25	52
Stanford [51]	Stanford campus backbone network	26	56

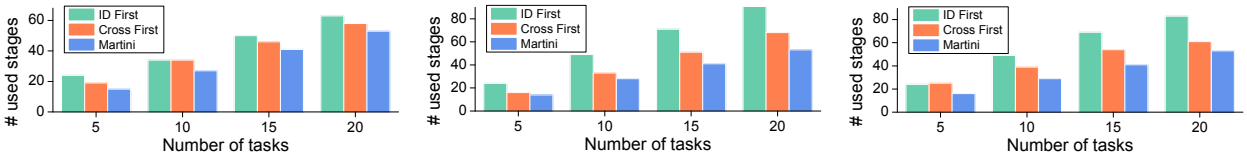
Effective support for management tasks that require timely control: Above results in reducing the control loop means that Martini could effectively support timely management tasks. As revealed in [78], for the microburst task, enabling local control could improve total bandwidth by 19.5% and reduce average 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 300*Gbps*, reducing the control loop from 3.4*s* to 10*ns* could theoretically defend against 1020*Gb* attack traffic. For the superspreader task, operators could reduce the superspreader detection time from 3.4*s* to 10*ns* and therefore prevent worm propagation.

C. 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 [59]. 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 VIII as input. We randomly pick tasks 5, 10, 15 and 20 times from Table VII as test sets, and randomly assign one path to each task to make 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 [58] to quickly solve the 0-1 ILP problem in Martini. As shown in Figure 8, Martini outperforms the two naive algorithms in all topologies by occupying 9.4% to 56.3% fewer stages. Furthermore, as presented in Figure 9, 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*.

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 9. However, incremental task placement may be suboptimal. To



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

Fig. 8: Total number of used stages across all switches for naive algorithms and Martini placement algorithm.

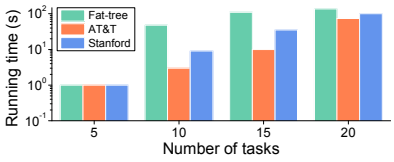


Fig. 9: Running time of Martini task placement algorithm.

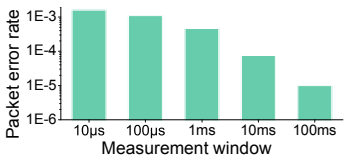


Fig. 10: Error caused by round based timing.

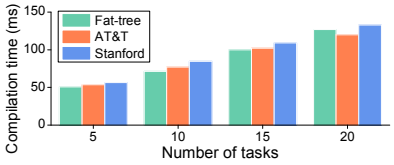


Fig. 11: Compilation time for switch code generation.

TABLE IX: Optimality gap of incremental 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%)

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 IX, 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.

D. Error Caused by Round Based Timing

We evaluate the percentage of faulty counter updates in round based timing (§V-A). Based on CAIDA traces, we vary the measurement window and count packets that cannot trigger correct counter clearance by identifying packet pairs whose rounds difference are multiples of 256 measurement windows. As shown in Figure 10, the percentage of packets causing error increases as the measurement window decreases, and tops at 0.2% when the window is $10\mu s$. This is acceptable since SRAM usage for timing is significantly reduced by 83.3%.

E. Compiler Performance

Finally, we run the the Martini compiler program with a single isolated server core. We compile the placement from 5 to 20 tasks on the test topologies in §VI-C and measure the compilation time. As shown in Figure 11, Martini finishes compilation within 200 ms. This demonstrates its scalability to quickly generate codes for real world network topologies.

VII. Related Work

Network measurement or management frameworks: Recent studies [45], [56], [57], [59], [87] have proposed frameworks for high performance and resource efficient *measurement* in switches. SNAP [9] offered a framework for stateful task description and deployment. In comparison, Martini is a general framework that supports measurement-based timely control in *management* tasks, and is based on advanced programmable switches with complex resource constraints.

Task description languages: Many recent works have proposed languages to describe either measurement or control tasks. For measurement tasks, Trumpet [66] designed a match-action like language to define network events. Marple [68] focused on querying performance information. Sonata [37] proposed primitives that imitated stream processing. NetQRE [88] focused on quantitative monitoring. Meanwhile, Pyretic [62], NetKat [8], FlexSwitch [80] and so on could describe control tasks. In contrast, Martini presents *modular*, *reusable*, and *extensible* primitives to describe and assemble all phases in a management task. Furthermore, Martini covers resource estimation and support for advanced measurement features, which have not been addressed by previous works.

Management task placement: Sonata [37] identifies the need for placing tasks *w.r.t.* switch resource constraints, but mainly solves the problem on a single switch. Martini instead performs network-wide placement. Some work [9], [59], [77] proposed to place measurement tasks on multiple switches in the network. Especially, SNAP [9] designed mechanisms for network-wide placement of stateful applications. However, it only considers link constraints and neglects resource constraints directly on switches. In comparison, Martini proposes algorithm to place resource-intensive management tasks in consideration of switch constraints. Some studies [48], [67] 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 multiple types of resource constraints in switching ASICs.

VIII. Conclusion and Future Work

This paper presents Martini, a general framework that enables measurement-based timely network control using programmable switching ASICs. Evaluations show that Martini could reduce the control loop to nanoseconds. In the future, we will enrich the primitives for more complex tasks and generate automatic verification for management task description.

Acknowledgments. We thank our shepherd, Gianni Antichi, and the anonymous reviewers for their valuable comments. This research is supported by National Key R&D Program of China (2017YFB0801701) and the National Science Foundation of China (61625203, 61832013, 61872426). Mingwei Xu is the corresponding author.

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] ALIPOURFARD, O., MOSHREF, M., AND YU, M. Re-evaluating measurement algorithms in software. In *HotNets* (2015), p. 20.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. Understanding data center traffic characteristics. *SIGCOMM CCR 40*, 1 (2010), 92–99.

[12] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. Microte: Fine grained traffic engineering for data centers. In *CoNEXT* (2011).

[13] 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.

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

[15] 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.

[16] 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.

[17] 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).

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

[19] 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.

[20] CAESAR, M., CASADO, M., KOPONEN, T., REXFORD, J., AND SHENKER, S. Dynamic route recomputation considered harmful. *SIGCOMM CCR 40*, 2 (2010), 66–71.

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

[22] CHEN, H., AND BENSON, T. The case for making tight control plane latency guarantees in sdn switches. In *SOSR* (2017), pp. 150–156.

[23] 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).

[24] COMMUNICATIONS, S. Spirent testcenter., 2017.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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).

[30] 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.

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

[32] FLOYD, S., AND JACOBSON, V. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking (ToN) 1*, 4 (1993), 397–413.

[33] 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.

[34] 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.

[35] GUPTA, A., BIRKNER, R., CANINI, M., FEAMSTER, N., MACSTOKER, C., AND WILLINGER, W. Network monitoring as a streaming analytics problem. In *HotNets* (2016), pp. 106–112.

[36] GUPTA, A., HARRISON, R., PAWAR, A., CANINI, M., FEAMSTER, N., REXFORD, J., AND WILLINGER, W. Sonata: Query-driven streaming network telemetry.

[37] 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.

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

[39] HARI, A., LAKSHMAN, T., AND WILFONG, G. Path switching: Reduced-state flow handling in sdn using path information. In *CoNEXT* (2015), p. 36.

[40] HARRISON, R., CAI, Q., GUPTA, A., AND REXFORD, J. Network-wide heavy hitter detection with commodity switches. In *SOSR* (2018).

[41] 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.

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

[43] HOPPS, C. E. Analysis of an equal-cost multi-path algorithm.

[44] HUANG, D. Y., YOCUM, K., AND SNOEREN, A. C. High-fidelity switch models for software-defined network emulation. In *HotSDN* (2013), pp. 43–48.

[45] 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.

[46] JOSE, L., YAN, L., VARGHESE, G., AND MCKEOWN, N. Compiling packet programs to reconfigurable switches. In *NSDI* (2015), pp. 103–115.

[47] 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.

[48] 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.

[49] 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).

[50] KATTA, N., HIRA, M., KIM, C., SIVARAMAN, A., AND REXFORD, J. Hula: Scalable load balancing using programmable data planes. In *SOSR* (2016), p. 10.

[51] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *NSDI* (2012), vol. 12, pp. 113–126.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] LI, Y., MIAO, R., KIM, C., AND YU, M. Flowradar: A better netflow for data centers. In *NSDI* (2016), pp. 311–324.

[57] 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.

[58] LINDO. Lingo and optimization modelling, 2016.

[59] 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.

[60] 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.

[61] 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.

[62] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., WALKER, D., ET AL. Composing software defined networks. In *NSDI* (2013), vol. 13, pp. 1–13.

[63] 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).

[64] MOSHREF, M., YU, M., GOVINDAN, R., AND VAHDAT, A. Dream: dynamic resource allocation for software-defined measurement. *SIGCOMM 44*, 4 (2015), 419–430.

[65] MOSHREF, M., YU, M., GOVINDAN, R., AND VAHDAT, A. Scream: Sketch resource allocation for software-defined measurement. In *CoNEXT* (2015), p. 14.

[66] MOSHREF, M., YU, M., GOVINDAN, R., AND VAHDAT, A. Trumpet: Timely and precise triggers in data centers. In *SIGCOMM* (2016), pp. 129–143.

[67] MOSHREF, M., YU, M., SHARMA, A. B., AND GOVINDAN, R. Scalable rule management for data centers. In *NSDI* (2013), vol. 13, pp. 157–170.

[68] 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.

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

[70] OPPENHEIM, A., WILLSKY, A., AND NAWAB, S. *Signals and Systems*. Prentice-Hall signal processing series. Prentice Hall, 1997.

[71] PERRY, J., OUSTERHOUT, A., BALAKRISHNAN, H., SHAH, D., AND FUGAL, H. Fastpass: A centralized zero-queue datacenter network. *SIGCOMM 44*, 4 (2015), 307–318.

[72] 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.

[73] 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.

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

[75] 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.

[76] 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.

[77] 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.

[78] SEN, S., SHUE, D., IHM, S., AND FREEDMAN, M. J. Scalable, optimal flow routing in datacenters via local link balancing. In *CoNEXT* (2013).

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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).

[84] 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.

[85] 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.

[86] 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.

[87] YU, M., JOSE, L., AND MIAO, R. Software defined traffic measurement with opensketch. In *NSDI* (2013), vol. 13, pp. 29–42.

[88] 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.

[89] ZHANG, J., REN, F., AND LIN, C. Modeling and understanding tcp incast in data center networks. In *INFOCOM* (2011), pp. 1377–1385.

[90] ZHANG, Q., LIU, V., AND ZENG, H. High-resolution measurement of data center microbursts.

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

[92] 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.