TIMELY, ACCURATE AND SCALABLE NETWORK MANAGEMENT FOR DATA CENTERS

by

Masoud Moshref Javadi

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER ENGINEERING)

December 2016

Copyright 2016 Masoud Moshref Javadi

Acknowledgements

I would like to express my deepest appreciation to professors Ramesh Govindan and Minlan Yu for being excellent mentors and providing me with insightful guidance toward academic excellence. All of my work in this dissertation has been guided by them and, as a result, has been improved tremendously. I would also like to thank my dissertation committee, including professors Murali Annavaram, Konstantinos Psounis, and Viktor Prasanna, for their insightful feedback and guidance on improving the quality of this document. Throughout this dissertation, I have been honored to have the opportunity to collaborate with prominent researchers in industry: DREAM, SCREAM and Trumpet (Chapters 2 and 3) are joint work with Amin Vahdat; vCRIB (Chapter 4) is a joint work with Abhishek B. Sharma. Although not included in this dissertation, I have been blessed to work with Harsha V. Madhyastha, Leana Golubchik, Omid Alipourfard, Adhip Gupta and Apoorv Bhargava in other projects.

Finally, I would also like to thank my family for their sincere support; specifically my beloved wife Zahra, my parents Amineh and Asghar and lastly my lovely son, Ali. Without their support, this dissertation would not exist.

Table of Contents

List of Tables

List of Figures

Abstract

Managing data center networks is critical to keeping cloud services always available, fast and efficient. Today, network operators have limited tools that provide a delayed and inaccurate view and control for the network. This causes huge service delay and hours of service disruption before resolving a failure, which can cost millions of dollars. The key problem is that these tools are designed in a bottom-up fashion and are driven by device limitations instead of high-level goals. This keeps human operators always involved in device-level details such as the allocated resources for each management task at each device, which prevents detecting all events of interest and a fast reaction to network events. In this dissertation, we follow a top-down approach, where operators program a centralized controller based on high-level abstractions. Knowing what the operator wants, the controller uses algorithms to program switches and servers. These algorithms can quickly fine tune the switches and servers to keep high accuracy, react to events quickly, and leverage device optimizations and network knowledge to scale. In this dissertation, we developed four systems following the top-down approach that configure network monitoring and control resources at switches and servers.

First, we focused on providing scalable and accurate network measurement. There can be a variety of concurrent, dynamically instantiated, measurement tasks in a data center. These tasks configure flow counters in commodity switches to monitor traffic. However, flow counters use TCAM memory which is fundamentally limited, and the accuracy of the measurement tasks is a function of the resources devoted to them on each switch and traffic properties that change over time. We developed DREAM as an adaptive measurement framework that allows the operators to define measurement tasks with a required level of accuracy. DREAM dynamically adjusts the resources devoted to each measurement task while ensuring the

user-specified accuracy. DREAM can support 30% more concurrent tasks with up to 80% more accurate measurements than static allocation.

Next, we focused on the measurement algorithms that use sketches. Sketches support more measurement tasks than flow counters and use hash-tables that run on a cheaper memory at switches (SRAM). However, their counters may have random errors depending on the hash collisions in the hash-tables. We developed SCREAM as a measurement framework to adaptively allocate SRAM memory to measurement tasks considering the probability of such errors while ensuring the user-specified level of accuracy. SCREAM can support 2x more tasks with higher accuracy than the state-of-the-art static allocation.

Third, we explored timely, accurate and scalable monitoring using the CPU resources and programmability of end-hosts (servers). We proposed Trumpet, an event monitoring system that translates a network-wide event to per end-host triggers, monitors every packet and aggregates triggers into an event in millisecond timescales. Using careful design, Trumpet can evaluate triggers by inspecting every packet at full line rate even on future generations of NICs, scale to thousands of triggers per end-host while bounding packet processing delay to a few microseconds, and report events to a controller within 10 milliseconds, even in the presence of attacks.

Finally, we developed vCRIB to achieve scalable and accurate network control for data centers. Cloud operators increasingly need more and more fine-grained rules to better control individual network flows for various traffic management policies. vCRIB provides the abstraction of a centralized rule repository using which operators will not be worried about where to place rules and the resources at the target switch/server. vCRIB automatically places rules on switches and servers with enough resources on/off the shortest path of flows while respecting the semantics of rules. Moreover, vCRIB minimizes the traffic overhead induced by rule placement in the face of traffic changes and VM migration.

Chapter 1

Introduction

In recent years, cloud services become popular. Many Fortune 500 companies (*e.g.,* Netflix, Best Buy and Expedia), startup companies (*e.g.,* Lyft, Airbnb, Spotify and Snapchat) and government organizations (*e.g.,* FDA) use the public cloud. Selling cloud services is also a profitable business for public cloud providers. For example, Amazon made $1.86B profit out of $7.8B revenue in 2015 from its cloud service [67].

Cloud services are enabled by data centers with a huge scale. Data centers have about 100k servers and a huge network to connect the servers together. For data centers to support cloud, data center networks must satisfy tight requirements in four different areas, and to achieve each, they need network management tasks:

Availability: Cloud must be available 99.95% of the time per month [60]. This means that the data center cannot be down for more than 20 minutes per month. However, network elements have many failures per day. For example, Microsoft reported 40 link and 5 device failures per day in Azure data centers [57]. Even when everything works perfectly fine, there are thousands of updates per hour [104] that can go wrong [139, 59]. To keep high availability, we need to run network management tasks such as failure monitoring to respond before users notice the problem.

Delay: Delay decreases sale and website traffic [71]: Amazon reported that every 100ms of latency cost 1% in sales, and Google found that half a second extra delay in search pages dropped traffic by 20%.

Network congestion can double the service response time [7]. Therefore, we need management tasks such as rate limiting to prevent aggressive flows from causing congestion.

Utilization: Data centers are very expensive, and cloud providers want to maximize the utilization of their investment. To maintain competitiveness, cloud providers seek to reduce network cost by increasing its utilization [80]. For this, they run network management tasks such as traffic engineering that distributes network flows on different links.

Scalability: Finally, data center networks have a huge scale: They connect hundreds of thousands of servers inside a data center using thousands of network switches that must work in a coordinated way [104]. The bandwidth demand on these networks is in the millions of Gbps and is doubling every year [136]. Moreover, thousands of users concurrently use the network for applications with diverse requirements.

Network management tasks including failure monitoring, rate limiting and traffic engineering, have three key requirements: First, in order to guarantee high availability and low delay, the monitoring and reaction must be *timely*. Second, finding and controlling flows that are mostly small [127] requires high *accuracy*. An inaccurate network control can cause congestion and packet losses that damage the performance of prevalent short connections. Finally, these tasks must handle millions of flows, so their *scalability* is very important.

Network management uses switches and end-hosts, and the challenge is that management has to compete for limited network resources. At switches, there is limited memory and support for complex programs. At end-hosts, CPU is precious because it runs cloud users workload, so the CPU budget for management is limited.

This dissertation goal is *to design timely, accurate and scalable network management* to achieve higher availability, lower delay and higher utilization in future data centers.

1.1 Current Solutions in Practice

Software-defined Networking (SDN) distinguishes two layers in networks: the control plane running at a logically centralized controller that enforces high-level policies by making routing decisions and the data

plane at switches that applies the decisions in forwarding traffic. The centralized controller receives measurements from switches and sends routing decisions to them using a standard protocol (*e.g.,* OpenFlow). SDN allows users to define their own virtual networks on top of the shared physical network and to measure and control them through applications running on the controller. Operators must decide where and how to run those applications to scale to thousands of users on top of thousands of switches with limited resources without compromising accuracy or timeliness.

The motivation for this dissertation is that the current management tools for operators are too limited. Network operators are involved in every aspect of data centers from design and deployment to operation, and from fault detection to application performance optimization to security. However, they have a limited view of network events and limited tools to control the response to the events. Inaccurate measurement tools prevent operators from knowing where bandwidth bottlenecks are and why network packets are delayed. Slow measurement tools increase the delay of detecting a failure or an attack to minutes. The control system must also scale to thousands of switches and hundreds of application requirements.

The key problem is that these tools are designed in a bottom-up fashion and are driven by device limitations instead of high-level goals. This keeps operators always involved in device-level details such as the allocated resources for each management task at each device, which prevents a fast reaction to network events and using resources efficiently for accurate and scalable network management.

For example in monitoring, switches and end-hosts collect packet samples and send them to a controller. At the controller, the operator looks into the data, and if she notices an issue, she may increase sample rates to drill down and find its root cause. This is not timely, accurate or scalable: The right sampling rate for high accuracy depends on monitoring parameters, available resources and traffic properties on each switch. Currently, operators must set it based on their intuition. This process is not scalable to many measurement tasks, many switches and many operators for fine time-scale tuning, thus operators may either over-provision resources and reduce scalability, or under-provision resources and lose accuracy. Finally, the operators are involved in drilling down and responding to measurements which is not fast. By

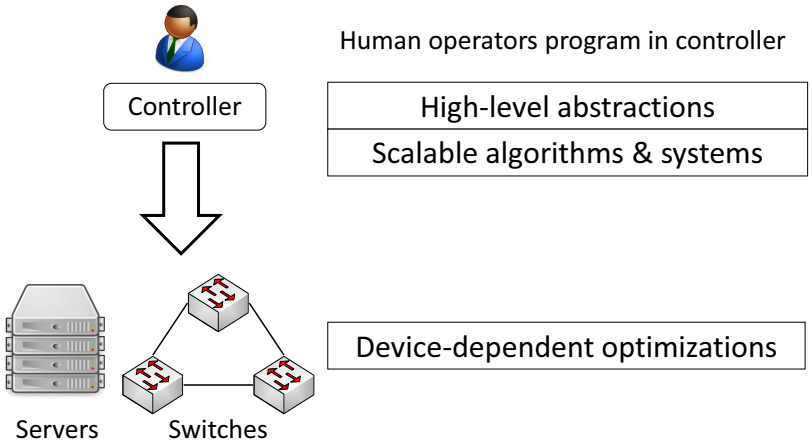


Figure 1.1: *A top-down approach for network management*

the time they look into the problem, the root cause may have vanished. We reviewed related research that tried to address some of these issues in Chapter 5.

1.2 Contributions: Timely, Accurate and Scalable Network Management

A top-down approach: In this dissertation, we follow a top-down approach (Figure 1.1), where operators program a centralized controller based on high-level abstractions. Knowing what the operator wants, the controller uses algorithms to program switches and servers. These algorithms can quickly fine tune the switches and servers to maintain high accuracy, react to events fast, and leverage device optimizations and network knowledge to scale.

Network management involves both measurement and control since controlling a network is not possible without observing the events in the network. In this dissertation, we built both accurate and timely measurement systems and an accurate and scalable network control system (Figure 1.2). The measurement systems on switches allows accurate network monitoring for many measurement tasks by using switch resources efficiently for each task on each switch. The measurement system on end-hosts leverages the programmability of CPUs to provide timely, accurate and scalable measurement. Finally, we proposed a control system that provides scalable network control using fine-grained rules by placing the rules on both switches and end-hosts dynamically while maintaining the semantics of rules.

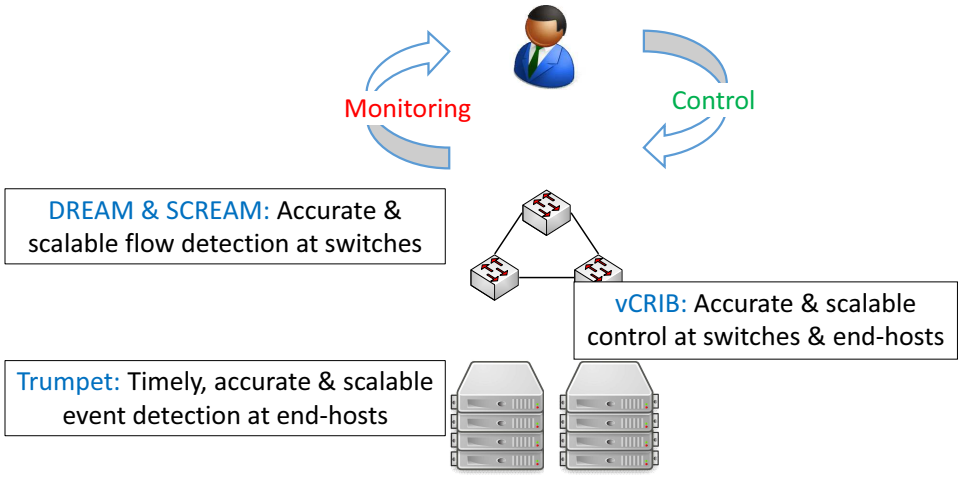


Figure 1.2: *Timely, accurate and scalable network management systems*

From another point of view: On hardware switches, we defined general abstractions for operators that free them from the complexities imposed by hardware resource limitations. On end-hosts, we developed systems that are tuned based on the effect of new features at end-host CPU on packet processing performance.

We introduce four systems developed for this dissertation briefly in the rest of this Section.

1.2.1 Accurate, yet Resource Efficient, Measurement on Switches

Measurement tasks require significant bandwidth, memory and processing resources, and the resources dedicated to these tasks affect the accuracy of the eventual measurement. However, the resources are limited, and data centers must support a variety of concurrent measurement tasks. Thus, it is important to design a measurement system that can support many tasks and keep all of them accurate on a network with limited resources.

Measurement tasks can be implemented using different primitives with different resource accuracy trade-offs. In this dissertation, we explored the trade-off space of resource usage versus accuracy for three primitives: (a) Flow counters monitor traffic in hardware switches using expensive and power hungry TCAM (Ternary Content-Addressable Memory) and are available in commodity switches. (b) Hash-based counters implement sketches that can express many more measurement task types with higher accuracy

and use cheap SRAM (Static Random-Access Memory), but are not available yet. (c) Arbitrary program fragments are more expressive, but they are only possible at end-hosts and have complex resource-accuracy trade-offs. Focusing on flow counters and hash-based counters, we noticed that although the accuracy of a measurement task is a function of its allocated memory on each switch, this function changes with traffic properties, which forces operators to provision for the worst case.

We developed DREAM for flow counters and SCREAM for hash-based counters (Chapter 2) to provide operators with the abstraction of guaranteed measurement accuracy that hides resource limits from operators. Our insight is to dynamically adjust resources devoted to each measurement task and multiplex TCAM and SRAM entries temporally and spatially among them to support more accurate tasks on limited resources. The key idea is to iteratively allocate resources to tasks by continuously estimating their accuracy. We proposed new algorithms to solve three challenges: (a) network-wide measurement tasks that can correctly merge measurement results from multiple switches with a variable amount of resources; (b) online accuracy estimation algorithms for each type of task that probabilistically analyze their output without knowing the ground-truth and (c) a scalable resource allocation algorithm that converges fast and is stable.

We built a prototype of DREAM on OpenFlow switches with three network-wide measurement task types (heavy hitter, hierarchical heavy hitter and change detection), and we showed that DREAM can support 30% more concurrent tasks with up to 80% more accurate measurements than fixed allocation. For SCREAM, we have implemented heavy hitter, hierarchical heavy hitter and super source detection task types. Simulations on real-world traces show that SCREAM can support 2x more tasks with higher accuracy than the state-of-the-art static allocation and the same number of tasks with comparable accuracy as an oracle that is aware of future task resource requirements.

1.2.2 Timely, Accurate and Scalable Measurement Using End-hosts

DREAM and SCREAM are efficient in detecting flows with certain properties, but they are limited by switch capabilities in detecting per packet and fine time-scale events. Thus, we looked at a different point

in the design space: monitoring at end-hosts, where we can do per packet monitoring and in much smaller time-scales. We developed an expressive scalable measurement system on servers, Trumpet (Chapter 3), that monitors every packet in 10G links with small CPU overhead and reports events in less than 10ms even in the presence of an attack. Trumpet is an event monitoring system in which users define network-wide events, and a centralized controller installs triggers at end-hosts, where triggers run arbitrary codes to test for local conditions that may signal the network-wide events. The controller aggregates these signals and determines if the network-wide event indeed occurred.

1.2.3 Scalable and Accurate Control on Switches & End-hosts

In SDN, applying many high-level policies such as access control requires many fine-grained rules at switches, but switches have limited rule capacity. This complicates the operator's job as she needs to worry about the constraints on switches when defining rules. We leveraged the opportunity that there can be different places, on or off the shortest path of flows, to apply rules if we accept some bandwidth overhead and proposed vCRIB (Chapter 4) to provide operators with the abstraction of a scalable rule storage. vCRIB automatically places rules on hardware switches and end-hosts with enough resources and minimizes the bandwidth overhead. We solved three challenges in its design:

1. Separating overlapping rules may change their semantics, so vCRIB "partitions" overlapping rules to decouple them.
2. vCRIB must pack partitions on switches considering switch resources. We solved this as a new bin-packing problem by a novel approximation algorithm with a proven bound. We modeled the resource usage of rule processing at end-hosts and generalized the solution to both switches and end-hosts.
3. Traffic patterns change over time. vCRIB minimizes traffic overhead induced by rule placement using an online greedy algorithm that adaptively changes the location of partitions in the face of traffic changes and VM migration.

We demonstrate that vCRIB can find feasible rule placements with less than 10% traffic overhead when traffic-optimal rule placement is infeasible.

Chapter 2

Dynamic Resource Allocation for Software-defined Measurement

Software-defined networks can enable a variety of concurrent, dynamically instantiated, measurement tasks, that provide fine-grain visibility into network traffic. Recently, there have been many proposals to configure flow counters and sketches in hardware switches to monitor traffic. However, the flow counters and sketches in hardware switches use constrained resources such as TCAM and SRAM memory, and the accuracy of measurement tasks is a function of the resources devoted to them on each switch. This Chapter describes two adaptive measurement framework, DREAM for flow counters and SCREAM for sketches, that dynamically adjust the resources devoted to each measurement task, while ensuring a user-specified level of accuracy. Since the trade-off between resource usage and accuracy can depend upon the type of tasks, their parameters, and traffic characteristics, DREAM and SCREAM do not assume an a priori characterization of this trade-off, but instead dynamically search for a resource allocation that is sufficient to achieve a desired level of accuracy. Our systems estimate the instantaneous accuracy of tasks so as to dynamically adapt the allocated resources for each task. A prototype implementation and simulations with different network-wide measurement tasks (heavy hitter, hierarchical heavy hitter, change and super source/destination detection) and diverse traffic show that DREAM and SCREAM can support more concurrent tasks with higher accuracy than several other alternatives.

2.1 Introduction

Today's data center and enterprise networks require expensive capital investments, yet provide surprisingly little visibility into traffic. Traffic measurement can play an important role in these networks, by permitting traffic accounting, traffic engineering, load balancing, and performance diagnosis [5, 16, 28, 42, 8], all of which rely on accurate and timely measurement of time-varying traffic at all switches in the network. Beyond that, tenant services in a multi-tenant cloud may need accurate statistics of their traffic, which requires collecting this information at all relevant switches.

Software-defined Measurement [150, 82, 112] has the potential to enable concurrent, dynamically instantiated, *measurement tasks*. In this approach, an SDN controller orchestrates these measurement tasks at multiple spatial and temporal scales, based on a global view of the network. Examples of measurement tasks include identifying flows exceeding a given threshold and flows whose volume changes significantly.

Flow-counters and sketches are two recent primitives for network measurement. Flow-counters in measurement algorithms can be used to detect heavy hitters and significant changes [112, 155, 89]. These algorithms can leverage existing TCAM hardware on switches and so have the advantage of immediate deployability. A switch may need to count traffic from thousands of source IP addresses to find heavy users of a specific service, for each of which it would require a counter. To reduce counter usage, many solutions rely on counting traffic to/from prefixes (instead of specific IP addresses), and then iteratively zooming in and out to find the right set of flows to monitor [112, 155, 89].

Similarly, sketches (hash-based counters) have been extensively used for stream processing including network measurement. Sketches are summaries of streaming data for approximately answering a specific set of queries. They can be easily implemented with SRAM memory which is cheaper and more power-efficient than TCAMs. Sketches can use sub-linear memory space to answer many measurement tasks such as finding heavy hitters [37], super-spreaders [35], large changes [94], flow-size distribution [95], flow quantiles [37], and flow-size entropy [100]. Finally, they can capture the right set of flow properties in the data plane without any iterative reconfiguration from the controller.

Any design for SDM faces two related challenges. First, SDM permits multiple instances of measurement tasks, of different types and defined on different aggregates, to execute concurrently in a network. For example, an operator may run, on a (virtualized) network, tasks such as finding large flows for multi-path routing [5] and finding sources that initiate many connections for anomaly detection [145]. Furthermore, in a cloud setting, each tenant can issue distinct measurement tasks for its own virtual network; cloud providers already offer simple per-tenant measurement services [10] and SDN-based network functionality virtualization for tenants [142]. Thus, in the near future, SDM will be the norm in clouds with a huge number of tenants [11].

The second challenge is that measurement tasks may require significant resources. To achieve a required accuracy, each task may need many TCAM entries and up to a million hash-based counters, and the number of counters is bounded by resources such as the TCAM memory for flow counters, the SRAM memory needed for saving sketch counters, the control datapath inside switches required to report counters from ASIC to CPU, and the control network bandwidth that is shared among many switches.

Therefore, an SDM design must ensure efficient usage of these resources. However, for flow-based counters, the relationship between resource and accuracy for measurement tasks cannot be characterized *a priori* because it depends upon the traffic characteristics. For many forms of sketches, it is possible to estimate the resources required to achieve a desired accuracy (*i.e.,* there is a *resource-accuracy* trade-off), but these resource estimates are also dependent on traffic. Prior work [150] has assumed *worst-case* traffic in allocating resources to sketches, and this can result in pessimistic overall resource usage, reducing the number of tasks that can be concurrently supported. In contrast, our key idea is to use a *dynamic* resource allocator that gives just enough resources to each task for the *traffic* it observes. It dynamically adapts the resource allocation based on *an instantaneous accuracy estimation* as traffic changes over time and across switches.

Contributions. In this chapter, we discuss the design of a system for TCAM-based Software-defined Measurement (SDM), called DREAM and a system for sketch-based SDM, called SCREAM. Users of DREAM and SCREAM can dynamically instantiate multiple concurrent measurement tasks (such as heavy

hitter or hierarchical heavy hitter detection, change detection or super source/destination detection) at an SDN controller, and additionally specify a *flow filter* (*e.g.,* defined over 5-tuples) over which this measurement task is executed. Since multiple tasks may need to measure at each switch, our systems need to allocate switch resources to each task.

To do this, our systems first leverage two important observations. First, although tasks become more accurate with more resources, there is a point of diminishing returns: beyond a certain accuracy, significantly more resources are necessary to increase the accuracy of the task. Moreover, beyond this point, the *quality* of the retrieved results, say heavy hitters is marginal (as we quantify later). This suggests that it would be acceptable to maintain the accuracy of measurement tasks above a high (but below 100%) user-specified *accuracy bound*. Second, tasks need TCAM and SRAM resources only on switches at which there is traffic that matches the specified flow filter, and the number of resources required depends upon the traffic volume and the distribution. This suggests that allocating just enough resources to tasks at switches and over time might provide *spatial and temporal statistical multiplexing benefits*.

DREAM and SCREAM use both of these observations to permit more concurrent tasks than is possible with a static allocation of TCAM and SRAM resources. We propose a novel resource adaptation strategy for determining the right set of resources assigned to each task at each switch. Getting *instantaneous accuracy estimation* feedbacks from each measurement task, DREAM and SCREAM increase the allocated resources to a task at a switch when its global estimated accuracy and its accuracy at the switch are below the accuracy bound. In this manner, they decouple resource allocation, which is performed locally, from accuracy estimation, which is performed globally. They continuously adapt the resources allocated to tasks since a task's accuracy and resource requirements can change with traffic. Finally, if they are unable to get enough resources for a task to satisfy its accuracy bound, they *drop* the task.

DREAM, Dynamic REsource Allocation for Measurement, is a *flow-based* SDM system that enables the *dynamic* resource allocation for flow-based measurement tasks of three types (heavy hitter, hierarchical heavy hitter and change detection) while achieving the required accuracy for each task. DREAM incorporates a new method to estimate accuracy for measurement tasks on multiple switches without ground-truth

or an a priori knowledge of traffic model. We demonstrate through extensive experiments on a DREAM prototype (in which multiple concurrent tasks three different types are executed) that it performs significantly better than other alternatives, especially at the tail of important performance metrics, and that these performance advantages carry over to larger scales evaluated through simulation. DREAM's satisfaction metric (the fraction of task's lifetime that its accuracy is above the bound) is $2\times$ better at the tail for moderate loads than an approach which allocates equal share of resources to tasks: in DREAM, almost 95% of tasks have a satisfaction higher than 80%, but for equal allocation, 5% have a satisfaction less than 40%. At high loads, DREAM's average satisfaction is nearly $3\times$ that of equal allocation. Some of these relative performance advantages also apply to an approach which allocates a fixed amount of resource to each task, but drops tasks that cannot be satisfied. However, this fixed allocation rejects an unacceptably high fraction of tasks: even at low load, it rejects 30% of tasks, while DREAM rejects none. Finally, these performance differences persist across a broad range of parameter settings.

For sketches, we propose a sketch-based SDM system, called SCREAM (SketCh REsource Allocation for Measurement), which enables *dynamic* resource allocation of limited resources for many concurrent measurement tasks while achieving the required accuracy for each task. SCREAM solves two challenges: *(1) Sketch-based task implementation across multiple switches:* Each task type implementation must gather sketch counters from *multiple* switches and prepare measurement results to the user. As switches see different traffic, each sketch may need *different* sizes for an efficient and accurate measurement. SCREAM uses novel techniques to merge sketches with *different* sizes from multiple switches. This extension of sketch-based measurement [150] to multiple switches is a critical step towards making sketches useful in practice. *(2) Accuracy estimator:* SCREAM incorporates a new method to estimate accuracy for measurement tasks on multiple switches without ground-truth or an a priori knowledge of traffic model with low estimation errors, rather than rely on the worst-case bounds of sketches. SCREAM feeds these *instantaneous accuracy estimates* (which can also give operators some insight into how trustworthy the measurements are) into the dynamic resource allocation algorithm to support more accurate tasks by leveraging *temporal and spatial statistical multiplexing*.

We have implemented three measurement task types (heavy hitter, hierarchical heavy hitter and super source/destination) in SCREAM and improved their design for dynamic resource allocation on multiple switches. Our simulations demonstrate that SCREAM performs significantly better than other allocation alternatives. Compared to OpenSketch, which allocates resources on a single switch based on the worst-case bounds, SCREAM can support 2$\times$ more tasks with higher accuracy. This result is valid across all task types and even when applying OpenSketch on multiple switches. This is because SCREAM can leverage traffic variations over time to multiplex resources across task instances and switches while OpenSketch reserves the same fixed resources for all tasks of the same type. SCREAM can support the same number of tasks with comparable accuracy as an oracle which is aware of future task resource requirements.

2.2 Background

In this section, we go through a short background on the usage of flow-counters and sketches in network measurement.

2.2.1 Flow-based Measurement

To understand how TCAM memory can be effectively used for measurement, consider the heavy hitter detection algorithm proposed in [89]. The key idea behind this (and other TCAM-based algorithms) is, in the absence of enough TCAM entries to monitor every flow in a switch, to selectively monitor *prefixes* and drill down on prefixes likely to contain heavy hitters. Figure 2.1 shows a prefix trie of two bits as part of source IP prefix trie of a task that finds heavy hitter source IPs (IP addresses sending more than, say, 10Mbps in a measurement epoch). The number inside each node is the volume of traffic from the corresponding prefix based on the "current" set of monitored prefixes. The task reports source IP addresses (leaves) with volume greater than threshold.

If the task cannot monitor every source IP in the network because of limited TCAM counters, it only monitors a subset of leaves trading off some accuracy. It also measures a few internal nodes (IP prefixes) to guide which leaves to monitor next to maximize accuracy. For example in Figure 2.1, suppose the task

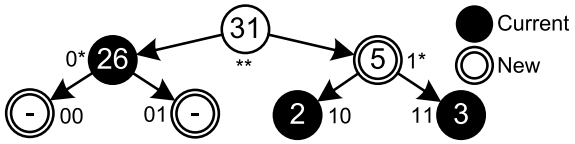


Figure 2.1: *TCAM-based task example*

is only allowed to use 3 TCAM counters, it first decides to monitor 11, 10 and 0*. As prefix 0* sends large traffic, the task decides to drill down under prefix 0* in the next epoch to find heavy hitters hoping that they will remain active then. However, to respect the resource constraint (3 TCAM counters), it must free a counter in the other sub-tree by monitoring prefix 1* instead of 10 and 11.

2.2.2 Sketch-based Measurement

Sketches are memory-efficient summaries of streaming data for approximately answering a specific set of queries. Sketches often provide a provable trade-off between resources and accuracy, where the definition of accuracy depends on the queries. We focus on hash-based sketches because they can be implemented on hardware switches using commodity components (hashing, TCAM, and SRAM modules) as discussed in OpenSketch [150]. Note that the same accuracy estimation and similar resource allocation techniques that are introduced in this Chapter can be applied to software switches where cache for counters and CPU budgets per packet are limited. We leave software switches to future work, but note that measurement in software switches or hypervisors does not extend to wide-area networks across datacenters, networks where operators do not have access to end hosts, and networks which devote most server resources to revenue-generating applications.

A commonly used sketch, the Count-Min sketch [37] can approximate volume of traffic from each item (*e.g.,* source IP) and is used for many measurement tasks such as heavy hitter detection (*e.g.,* source IP addresses sending traffic more than a threshold). Count-Min sketch keeps a two dimensional array, A, of integer counters with w columns and d rows. For each packet from an input item $x \in (0 \dots D)$ with size I_x, the switch computes d pairwise independent hash functions and updates counters, $A[i, h_i(x)] += I_x, i \in (1 \dots d)$. At the end of measurement epoch, the controller fetches all counters. When the controller queries the sketch for the size of an item, Count-Min sketch hashes the item again and reports the minimum of the

corresponding counters. As the controller cannot query every item (*e.g.,* every IP address), we need to limit the set of items to query. We can keep a sketch for each level of prefix tree (at most 32 sketches) and avoid querying lower levels of the tree by using the result of queries on upper levels (Section 2.6.2). Multiple items may collide on a counter and cause an over-approximation error, but Count-Min sketch provides a provable bound on the error. Using d hash functions each mapping to w entries bounds the worst-case error to: $e_{cm} \leq e\frac{T}{w}$ with probability $1 - e^{-d}$, where T is the sum of packet sizes. Approaches to improve Count-Min sketch accuracy, for example by running the least-squares method [98] or multiple rounds of approximation over all detected prefixes [105], add more computation overhead to the controller, and their resource-accuracy trade-off is not known in advance.

Many sketches have been proposed for counting distinct items [51, 49]. We use HyperLogLog [51] as its space usage is near-optimal, and it is easier to implement than the optimal algorithm [69]. First, we hash each item and count the number of leading zeros in the result, say 0_x. Intuitively, hash values with more leading zeros indicate more distinct items. By only keeping the count of maximum leading zeros seen over a stream, $M = max_i(0_{x_i})$, we can estimate the number of distinct items as 2^{M+1}. For this, a 5-bit counter is enough for a 32-bit hash function. We can replicate this sketch m times, and reduce the relative error of approximation with a factor of $\sqrt{m}$ but with no additional hashing overhead by using the first p bits of hash output to select from $m = 2^p$ replicas and the other bits to update the replica counter. For example, a distinct counter with $m = 64$ replicas will require 320 bits, have a standard deviation of the relative error of $\frac{1.04}{8}$, and will use the first 6 bits of hash outputs to select a replica.

2.3 Motivation

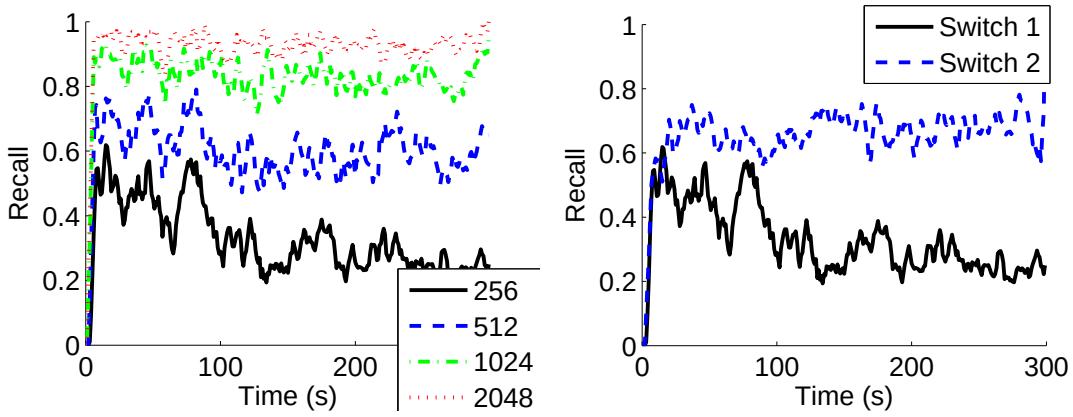
In this section, we motivate the fundamental challenges for real-time visibility into traffic in enterprise and data center networks. Software-defined Measurement (SDM) provides this capability by permitting a large amount of dynamically instantiated network-wide *measurement tasks*. These tasks often leverage flow-based counters in TCAM in OpenFlow switches or hash-based counters for sketches in SRAM. Unfortunately, the size of TCAM and SRAM memory is often limited. To make SDM more practical, we

propose to dynamically allocate measurement resources to tasks, by leveraging the diminishing returns in the accuracy of each task, and temporal/spatial resource multiplexing across tasks.

2.3.1 Task Diversity and Resource Limitations

SDM needs to support a large number of concurrent tasks, and dynamic instantiation of measurement tasks. These tasks can be of different types and defined on different traffic aggregates. For example, an operator may run different types of tasks in a (virtual) network, such as finding large flows for multi-path routing [5] and finding sources that make many connections for anomaly detection [145]. Operators may also instantiate tasks dynamically on different aggregates to drill down into anomalous traffic aggregates. Furthermore, in a cloud setting, each tenant can issue distinct measurement tasks for its virtual network; Amazon CloudWatch already offers simple per-tenant measurement services [10], and Google Andromeda allows SDN-based network functionality virtualization for tenants [142]. Besides, modern clouds service a large number of tenants (3 million domains used AWS in 2013 [11]), so SDM with many measurement tasks will be common in future clouds.

However, switches have limited memory and bandwidth resources for storing these counters. Each measurement task may need hundreds of TCAM entries for sufficient accuracy [112, 89, 155], but typical hardware switches have only a limited number of TCAMs. There are only 1k-2k TCAM entries in switches [42, 75], and this number is not expected to increase dramatically for commodity switches because of their cost and power usage. Moreover, other management tasks such as routing and access control need TCAMs, and this can leave fewer entries for measurement. Today's switches have 128 MB SRAM capacity (HP 5120-48G EI [73]) which can support 4-128 tasks where each task needs 1-32 MB SRAM counters [150]. In practice, SRAM is used for other functions, and only a small part of it is available for measurement. Moreover, there is limited bandwidth for the controller to fetch counters. First, inside a switch the control data-path that transfers counters from the ASIC to the switch CPU has low bandwidth (*e.g.,* 80 Mbps) [42]. Second, there is limited bandwidth to send counters from many switches to the controller. For example, 12 switches each dumping 80 Mb per second can easily fill a 1 Gbps link. Thus, we need sketches with



(a) *Resource change* **(b)** *Spatial diversity*

Figure 2.2: *Accuracy of HH detection*

fewer counters to reduce memory usage, lower network overhead and send counters more frequently to the controller to report in a short time-scale.

2.3.2 Why Dynamic Allocation?

Given limited resources and the need to support concurrent measurement tasks, it is important to efficiently allocate TCAM and SRAM resources for measurement.

Leverage: Diminishing returns in accuracy for measurement. The *accuracy* of a measurement task depends on the resources allocated to it [112, 150]. For example, for heavy hitter (HH) detection, *recall*, the fraction of true HHs that are detected, is a measure of accuracy. Figure 2.2 shows the result of our HH detection algorithm on a CAIDA traffic trace [20] with a threshold of 8 Mbps (See Section 2.6.1 for implementation details).

The figure shows that more counters leads to higher recall. For example, doubling counters from 512 to 1024 increases recall from 60% to 80% (Figure 2.2a). There is a point of diminishing returns for many measurement tasks [38, 109, 97, 95] where additional resource investment does not lead to proportional accuracy improvement. The accuracy gain becomes smaller as we double the resources; it only improves from 82% to 92% when doubling the number of counters from 1024 to 2048, and even 8K counters are insufficient to achieve an accuracy of 99%. Furthermore, the precise point of diminishing returns depends on the task type, parameters (*e.g.,* heavy hitter threshold) and traffic [112].

Another important aspect of the relationship between accuracy and resource usage of algorithms is that, beyond the point of diminishing returns, additional resources yield less significant outcomes, on average. For example, the heavy hitters detected with additional resources are intuitively "less important" or "smaller" heavy hitters and the changes detected by a change detection algorithm are smaller, by nearly a factor of 2 on average (we have empirically confirmed this).

This observation is at the core of our approach: *we assert that network operators will be satisfied with operating these measurement tasks at, or slightly above, the point of diminishing returns, in exchange for being able to concurrently execute more measurement tasks.*[1] At a high-level, our approach permits operators to dynamically instantiate three distinct kinds of measurement tasks (discussed later) and to specify a target accuracy for each task. It then allocates flow and sketch counters to these tasks to enable them to achieve the specified accuracy, adapts TCAM and SRAM allocations as tasks leave or enter or as traffic changes. Finally, our approach performs admission control because the accuracy bound is inelastic and admitting too many tasks can leave each task with fewer resources than necessary to achieve the target accuracy.

Leverage: Temporal and Spatial Resource Multiplexing. The TCAM and SRAM resources required for a task depends on the properties of monitored traffic. For example, as the number of heavy hitters increases, we need more resources to detect them. Prior work [150] has proposed tuning the size of sketches based on their resource-accuracy trade-off at *task instantiation* to maintain a required accuracy. However, these resource-accuracy trade-offs are for the *worst-case*. For example, based on the formulation of Count-Min sketch, to not detect items sending less than 9 Mbps for a threshold of 10 Mbps in a 10 Gbps link ($e_{cm} = 1$ Mbps), we need about 27 K counters of 4 bytes for each row; with 3 rows and a prefix tree with 32 levels, this works out to 5.5 MB.[2] However, the total traffic T of a link may not always reach the link capacity. In addition, Cormode [38] showed that the bound is loose for skewed traffic (a not uncommon case in real world) and the sketch can be exponentially smaller when sized for known skew. For the above

[1] Indeed, under resource constraints, less critical measurement tasks might well return very interesting/important results even well below the point of diminishing returns. We have left an exploration of this point in the design space to future work.

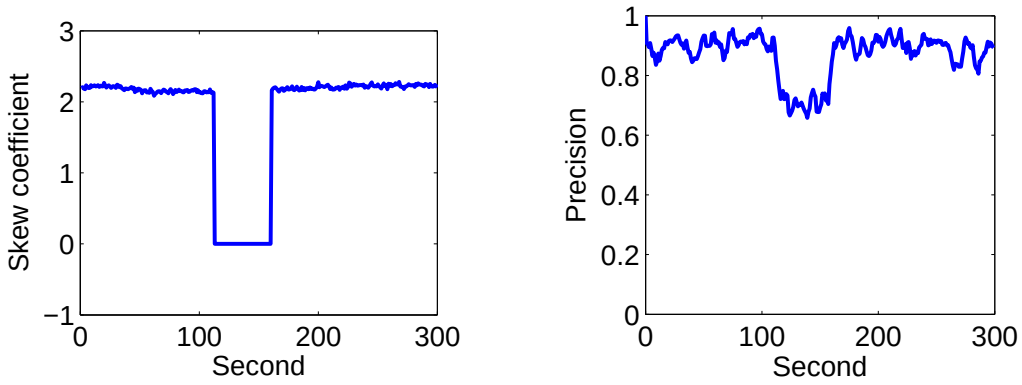
[2] The number of hash functions, d, is usually fixed to 3 or 4 to simplify hardware and because of reduced marginal gains for larger values. The total is smaller than $27K \times 4 \times 3 \times 32$ because the sketches for the top layers of the prefix tree can be smaller [36].

example, if the link utilization is 20% in average with a skew factor of 1.3 [38], each row will need only 1040 counters which require 260 KB of SRAM. Other sketches also exhibit traffic-dependent accuracy [33, 95]. In summary, these static trade-off formulations, when exist, are loose because the optimal resource requirement of a sketch for a given accuracy depends on the *traffic* that changes over time.

Here we show this dependence of required resources on traffic with two examples for flow-based tasks and sketch-based tasks. Figure 2.2a shows that the recall of HH detection task using flow-based counters with 256 TCAM entries decreases in the presence of more HHs (after time 100s) and we need more resources to keep its recall above 50%. For the second example, Figure 2.3 shows the variable accuracy of a heavy hitter detection task using Count-Min sketch over time. Figure 2.3a shows the skew of traffic from source IP addresses in CAIDA trace [20] over time. (Our skew metric is the *slope* of a fitted line on the log-log diagram of traffic volume from IP addresses vs. their rank (ZipF exponent).) The skew decreases from time 110 to 160 because of a DDoS attack. Figure 2.3b shows the accuracy of heavy hitter (HH) source IP detection of Count-Min sketch with 64 KB memory over time. Heavy hitter detection accuracy, *precision* (the fraction of detected true HHs over detected ones), decreases from 90% to 70% for less skewed traffic, which means that the sketch needs more counters *only* at this time period in order to maintain 90% accuracy.

Therefore, if we allocate fixed resources to each task, we would either over-provision the resource usage and support fewer tasks, or under-provision the resource usage and obtain low accuracy. However, this also presents an opportunity to *statistically multiplex* resources across tasks on a single switch: while a heavy hitter task on a switch may see many heavy hitters at a given time, a concurrent change detection task may see fewer anomalies at the same instant, and so may need fewer resources.

Measurement tasks also permit *spatial statistical multiplexing*, since the task may need resources from multiple switches. For example, we may need to find heavy hitter source IP addresses on flows of a prefix that come from multiple switches. Figure 2.2b shows the *recall* of heavy hitters found on two switches monitoring different traffic: the recall at each switch is defined by the portion of detected heavy hitters on



(a) *Skew* **(b)** *Sketch accuracy*

Figure 2.3: *Variation of traffic skew and sketch accuracy over time*

this switch over true heavy hitters. The graph shows that with the same amount of resources, the switches exhibit different recall; conversely, different amounts of resources may be needed at different switches.

These leverage points suggest that it may be possible to efficiently use TCAM and SRAM resources to permit multiple concurrent measurement tasks by (a) permitting operators[3] to specify desired accuracy bounds for each task, and (b) adapting the resources allocated to each task in a way that permits temporal and spatial multiplexing. This approach presents two design challenges.

Challenge: Estimating resource usage for a task with a desired accuracy. Given a task and target accuracy, we need to determine the resources to allocate to the task. If we knew the dependence of accuracy on resources, we could solve the resource allocation problem as an optimization problem subject to resource constraints. However, it is impossible to characterize the resource-accuracy dependence a priori for flow-based tasks because it depends on the traffic, the task type, the measurement algorithms, and the parameters [112]. Also, for sketch-based tasks, an allocation mechanism has to find and quantify the effect of different traffic properties on the resource-accuracy trade-off of each task, and run parallel measurement tasks to find the value of traffic properties (*e.g.,* skew, which can be used to tighten the accuracy bound for Count-Min sketch [92]). However, quantifying the effect of traffic properties (*e.g.,* skew parameters) on the accuracy is complicated, and dynamically estimating them may require significant resources [92].

[3]Operators may not wish to express and reason about accuracy bounds. Therefore, a deployed system may have reasonable defaults for accuracy bounds, or allow priorities instead of accuracy bounds, and translate these priorities to desired accuracy bounds. We have left an exploration of this to future work.

Furthermore, if we knew the current accuracy, we could then compare it with the desired accuracy and increase/decrease the resource usage correspondingly. Unfortunately, it is also impossible to know the current accuracy because we may not have the ground-truth during the measurement. For example, when we run a heavy hitter detection algorithm online, we can only know the heavy hitters that the algorithm detects (which may have false positives/negatives), but require offline processing to know the real number of heavy hitters.

To address this challenge, we need to *estimate* accuracy and then dynamically increase or decrease resource usage until the desired accuracy is achieved. For example, to estimate recall (a measure of accuracy) for heavy hitter detection, we can compute the real number of heavy hitters by estimating the number of missed heavy hitters using the collected counters. In Figure 2.1, for example, the task cannot miss more than two heavy hitters by monitoring prefix 0* because there are only two leaves under node 0* and its total volume is less than three times the threshold. In Section 2.7, we use similar intuitions to describe accuracy estimators for other measurement tasks.

Challenge: Spatial and temporal resource adaptation. As traffic changes over time, or as tasks enter and leave, an algorithm that continually estimates task accuracy and adapts resource allocation to match the desired accuracy (as discussed above) will also be able to achieve temporal multiplexing. In particular, such an algorithm will allocate minimal resources to measurement tasks whose traffic does not exhibit interesting phenomena (*e.g.,* heavy hitters), freeing up resources for other tasks that may incur large changes, for example. However, this algorithm alone is not sufficient to achieve spatial multiplexing, since, for a given task, we may need to allocate different resources on different switches to achieve a desired *global* accuracy. For example, a task may wish to detect heavy hitters within a prefix P, but traffic for that prefix may be seen on switch A and B. If the volume of prefix P's traffic on A is much higher than on B, it may suffice to allocate a large number of TCAM resources on A, and very few TCAM resources on B. Designing an algorithm that adapts network-wide resource allocations to achieve a desired global accuracy is a challenge, especially in the presence of traffic shifts between switches.

DREAM and SCREAM leverage both the global estimated accuracy and a measure of accuracy estimated for each switch to decide on which switch a task needs more resources in order to achieve the desired global accuracy. In addition, we change the task implementations in Section 2.6 to work with different and dynamic number of counters from different switches.

2.4 System Overview

DREAM and SCREAM enable resource-aware Software-defined Measurement. They support dynamic instantiation of measurement tasks with a specified accuracy, and automatically adapt TCAM and SRAM resources allocated to each task across multiple switches. DREAM and SCREAM can also be extended to other tasks and sketches for which it is possible to estimate accuracy.

Architecture and API: DREAM and SCREAM implement a collection of algorithms (defined later) running on an SDN controller. Users submit *measurement tasks* to the system. DREAM and SCREAM periodically report measurement results to users, who can use these results to reconfigure the network, install attack defenses, or increase network capacity. A user of our systems can be a network operator, or a software component that instantiates tasks and interprets results.

Our current implementation supports four *types* of measurement tasks, each with multiple parameters:

Heavy Hitter (HH): A heavy hitter is a traffic aggregate identified by a packet header field that exceeds a specified volume. For example, heavy hitter detection on source IP finds source IP addresses contributing large traffic in the network.

Hierarchical Heavy Hitter (HHH): Hierarchical heavy hitters (HHHs), used for detecting DDoS [132], are defined by the longest prefixes that exceed a certain threshold, θ, in aggregate traffic volume even after *excluding* any HHH descendants in the prefix tree [39]. For example, if a prefix 10.1.0.0/16 has traffic volume more than θ, but all the subnets within the prefix have traffic volume less than θ, we call the prefix a HHH. In contrast, if one of its subnets, say 10.1.1.0/24, has traffic more than θ, but the rest of the IPs collectively do not have traffic more than θ, we view 10.1.1.0/24 as a HHH, but 10.1.0.0/16 is not a HHH.

Change Detection (CD): Traffic anomalies such as attacks often correlate with significant changes in some aspect of a traffic aggregate (*e.g.,* volume or number of connections). For example, large changes in traffic volume from source IP addresses have been used for anomaly detection [155].

Super source or destination (SSD): A super source is a source IP that communicates with a more than a threshold number of *distinct* destination IP/port pairs. A super destination is defined in a similar way. SSDs are used for detecting worms, port-scans, P2P super nodes or DDoS targets.

We implemented HH, HHH and CD tasks in DREAM and HH, HHH and SSD tasks in SCREAM. Each of these tasks takes four parameters: a *flow filter* specifying the traffic aggregate to consider for the corresponding phenomenon (HH, HHH, CD or SSD); a *packet header field* on which the phenomenon is defined (*e.g.,* source IP address); a *threshold* specifying the minimum volume constituting a HH, HHH, CD or SSD; and a user-specified *accuracy bound* (usually expressed as a fraction). For example, if a user specifies, for a HH task a flow filter $< 10/8, 12/8, *, *, * >$, source IP as the packet header field, a threshold of 10Mb and an accuracy of 80%, DREAM measures, with an accuracy higher than 80%, heavy hitters in the source IP field on traffic from 10/8 to 12/8, where the heavy hitter is defined as any source IP sending more than 10Mb traffic in a measurement epoch. The user does not specify the switch to execute the measurement task; multiple switches may see traffic matching a task's flow filter, and it is DREAM's and SCREAM's responsibility to install measurement rules at all relevant switches.

Workflow. Figure 2.4 shows the workflow, illustrating both the interface to DREAM and the salient aspects of its internal operation. SCREAM also uses a similar workflow. A user instantiates a task and specifies its parameters (step 1). Then, DREAM decides to accept or reject the task based on available resources (step 2). For each accepted task, DREAM initially configures a default number of counters at one or more switches (step 3). DREAM also creates a *task object* for each accepted task: this object encapsulates the resource allocation algorithms run by DREAM for each task.

Periodically, DREAM retrieves counters from switches and passes these to task objects (step 4). Task objects compute measurement results and report the results to users (step 5). In addition, each task object contains an *accuracy estimator* that measures current task accuracy (step 6). This estimate serves as input

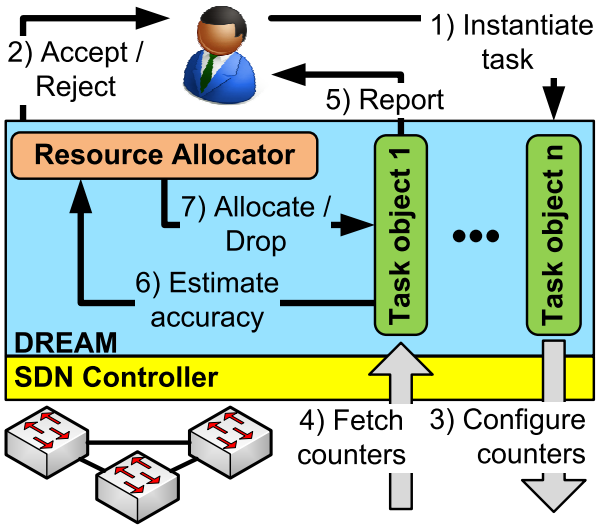


Figure 2.4: *DREAM system overview*

to the *resource allocator* component of DREAM, which determines the number of TCAM counters to allocate to each task and conveys this number to the corresponding task object (step 7). The task object determines how to use the allocated counters to measure traffic, and may re-configure one or more switches (step 3). If a task is dropped for the lack of resources, DREAM removes its task object and de-allocates the task's TCAM counters.

Generality: We have used the framework for two measurement primitives (flow-based counters and sketches) and implemented four task types. These *network-wide* measurement tasks have many applications in data centers and ISP networks. For example, they are used for multi-path routing [5], optical switching [27], network provisioning [47], threshold-based accounting [46], worm detection [145], P2P seed server detection [21], port scan [21], anomaly detection [154, 155, 89] and DDoS detection [132].

Moreover, although we have implemented measurement tasks in SCREAM based on Count-Min sketch and its variants, SCREAM can support other sketches for a different resource-accuracy trade-off or for other measurement tasks. If a given sketch's accuracy depends on traffic properties, it also benefits from our dynamic resource allocation algorithm. For example, the error of Count-Sketch [24], that can also support our three tasks, depends on the variation of traffic (compared to the error of Count-Min sketch which depends on the size of traffic). However, its theoretical bound is still loose for a skewed distribution [33].

Kumar [95] proposed a sketch to compute the flow size distribution, but the accuracy of this sketch also depends on the traffic properties (number of flows). In order to add new tasks to DREAM or SCREAM (for example using the above sketches), we need to a) implement the task in a way that handles dynamic and different amount of resources across switches and b) design and implement an accurate algorithm to estimate their global and local accuracy (See Section 2.5 for their definitions).

There are three main challenges in DREAM and SCREAM, discussed in subsequent sections: the design of the resource allocator, the design of tasks that can use dynamic and different amount of resources on switches, and the design of task-specific accuracy estimators.

2.5 Dynamic Resource Allocation

DREAM and SCREAM allocate TCAM and SRAM resources to measurement tasks on multiple switches. In this section, we describe our dynamic resource allocation algorithm in the context of DREAM, but the algorithm is general and is used in SCREAM with no changes.

Let $r_{i,s}(t)$ denote the amount of TCAM resources allocated to the i-th task on switch s at time t. Each task is also associated with an instantaneous global accuracy $g_i(t)$. Recall that the accuracy of a task is a function of the task type, parameters, the number of its counters per switch and the traffic matching its flow filter on each switch. DREAM allocates TCAM resources to maintain high average task *satisfaction*, which is the fraction of time where a task's accuracy $g_i(t)$ is greater than the operator specified bound. More important, at each switch, DREAM must respect switch capacity: the sum of $r_{i,s}(t)$ for all i must be less than the total TCAM resources at switch s, for all t.

To do this, DREAM needs a *resource allocation* algorithm to allocate counters to each task (*i.e.,* the algorithm determines $r_{i,s}(t)$). DREAM also needs an *admission control* algorithm; since the accuracy bound is inelastic (Section 2.3), admitting tasks indiscriminately can eventually lead to zero satisfaction as no task receives enough resources to achieve an accuracy above the specified bound.

Strawman approaches: One approach to resource allocation is to apply a convex optimization periodically, maximizing the number of satisfied tasks by allocating $r_{i,s}(t)$ subject to switch resource constraints. This

optimization technique requires a characterization of the resource-accuracy curve, a function that maps target accuracy to resources needed. The same is true for an optimization technique like simulated annealing which requires the ability to predict the "goodness" of a neighboring state. As discussed in Section 2.3.2, however, it is hard to characterize this curve a priori, because it depends upon traffic characteristics, and the type of task.

An alternative approach is to perform this optimization iteratively: jointly (for all tasks across all switches) optimize the increase or decrease of TCAM resources, measure the resulting accuracy, and repeat until all tasks are satisfied. However, this joint optimization is hard to scale to large numbers of switches and tasks because the combinatorics of the problem is a function of product of the number of switches and the number of tasks.

If the total resource required for all tasks exceeds system capacity, the first approach may result in an infeasible optimization, and the second may not converge. These approaches may then need to *drop* tasks after having admitted them, and in these algorithms admission control is tightly coupled with resource allocation.

Solution Overview: DREAM adopts a simpler design, based on two key ideas. First, compared to our strawman approaches, it loosely decouples resource allocation from admission control. In most cases, DREAM can reject new tasks by carefully estimating spare TCAM capacity, and admitting a task only if sufficient spare capacity (or *headroom*) exists. This headroom accommodates variability in aggregate resource demands due to traffic changes. Second, DREAM decouples the decision of *when* to adapt a task's resources from *how* to perform the adaptation. Resource allocation decisions are made when a task's accuracy is below its target accuracy bound. The *task accuracy computation* uses global information. However, in DREAM, a *per-switch resource allocator* maps TCAM resources to tasks on each switch, which increases/decreases TCAM resources locally at each switch step-wise until the overall task accuracy converges to the desired target accuracy. This decoupling avoids the need to solve a joint optimization for resource allocation, leading to better scaling.

Below, we discuss three components of DREAM's TCAM resource management algorithm: the task accuracy computation, the per-switch resource allocator, and the headroom estimator. We observe that these components are generic and do not depend on the types of tasks (HH, HHH, or CD) that the system supports.

Task Accuracy Computation: As discussed above, DREAM allocates additional resources to a task if its current accuracy is below the desired accuracy bound. However, because DREAM tasks can see traffic on multiple switches, it is unclear what measure of accuracy to use to make this decision per switch. There are two possible measures: *global* accuracy and *local* accuracy on each switch. For example, if a HH task has 20 HHs on switch A and 10 HHs on switch B, and we detect 5 and 9 true HHs on each switch respectively, the global accuracy will be 47% and the local accuracy will be 25% for A and 90% for B.

Let g_i be the global accuracy for task i, and $l_{i,s}$ be its local accuracy at switch s. The goal is to keep g_i above the accuracy bound. Thus adding measurement resources to task i at switch s is unnecessary if g_i is already above the accuracy bound even if $l_{i,s}$ is low. For the above example, we do not want to increase resources on switch A when the accuracy bound is 40%. Therefore, we only add resources if g_i is below the accuracy bound, but adding resources at all switches could be wasteful: At switch s, $l_{i,s}$ may already be above the accuracy bound, so it may be expensive to add additional resources to task i at switch s. This is because many measurement tasks reach a point of diminishing returns in accuracy as a function of assigned resources. In the above example, we do not want to increase resources on switch B when the accuracy bound is 80%. Thus, when g_i is low, we only add resources at switches that have a local accuracy, $l_{i,s}$, below the bound. This requires two properties for local accuracies in accuracy estimation algorithms: a) if global accuracy is low, at least one local accuracy must be low too; b) the switch that causes more error must have lower accuracy than others. The challenge is how to compute local accuracies when an item (*e.g.,* heavy hitter) can have traffic on multiple switches. In Section 2.7, we propose accuracy estimation algorithms that follow the above properties for different measurement tasks.

The discussion above motivates the use of an *overall accuracy* $a_{i,s} = max(g_i, l_{i,s})$ to decide when to make resource allocation decisions. Of course, this quantity may itself fluctuate because of traffic changes

and estimation error. To minimize oscillations due to such fluctuations, we smooth the overall accuracy using an EWMA filter. In what follows, we use the term overall accuracy to refer to this smoothed value. The overall accuracy for a task is calculated by its task object in Figure 2.4.

The Per-switch Resource Allocator: The heart of DREAM is the per-switch resource allocator (Figure 2.4), which runs on the controller and maps TCAM counters to tasks for each switch.[4] It uses the overall accuracy $a_{i,s}(t)$ to redistribute resources from *rich tasks* (whose overall accuracy are above the accuracy bound) to *poor tasks* (whose overall accuracy is below the accuracy bound) to ensure all tasks are above the accuracy bound. DREAM makes allocation decisions at the granularity of multiple measurement epochs, an *allocation epoch*. This allows DREAM to observe the effects of its allocation decisions before making new allocations.

Ensuring Fast Convergence with Adaptive Step Sizes: The allocator does not a priori know the number of necessary TCAM counters for a task to achieve its target accuracy (we call this the *resource target*, denoted by $R_{i,s}$). The resource target for each task may also change over time with changing traffic. The key challenge is to quickly converge to $R_{i,s}(t)$; the longer $r_{i,s}(t)$ is below the target, the less the task's satisfaction.

Because $R_{i,s}$ is unknown and time-varying, at each allocation epoch, the allocator iteratively increases or decreases $r_{i,s}(t)$ in *steps* based on the overall accuracy (calculated in the previous epoch) to reach the right amount of resources. The size of the step determines the convergence time of the algorithm and its stability. If the step is too small, it can take a long time to move resources from a rich task to a poor one; on the other hand, larger allocation step sizes enable faster convergence, but can induce oscillations. For example, if a satisfied task needs 8 TCAMs on a switch and has 10, removing 8 TCAMs can easily drop its accuracy to zero. Intuitively, for stability, DREAM should use larger step sizes when the task is far away from $R_{i,s}(t)$, and smaller step sizes when it is close.

[4] In practice, an operator might reserve a fixed number of TCAM counters for important measurement tasks, leaving only a pool of dynamically allocable counters. DREAM operates on this pool.

Since $R_{i,s}$ is unknown, DREAM estimates it by determining when a task changes its status (from poor to rich or from rich to poor) as a result of a resource change. Concretely, DREAM's resource allocation algorithm works as follows. At each measurement epoch, DREAM computes the sum of the step sizes of all the poor tasks s_P, and the sum of the step sizes of all the rich tasks s_R.[5] If $s_P \leq s_R$, then each rich task's $r_{i,s}$ is reduced by its step size, and each poor task's $r_{i,s}$ is increased in proportion to its step size (*i.e.,* s_R is distributed proportionally to the step size of each poor task). The converse happens when $s_P > s_R$. If we increase or decrease $r_{i,s}$ during one allocation epoch, and this does not change the task's rich/poor status in the next epoch, then, we increase the step size to enable the task to converge faster to its desired accuracy. However, if the status of task changes as a result of a resource change, we return TCAM resources (but use a smaller step size) to converge to $R_{i,s}$.

Figure 2.5 illustrates the convergence time to $R_{i,s}(t)$ for different increase/decrease policies for the step size. Here, multiplicative (M) policies change step size by a factor of 2, and additive (A) policies change step size by 4 TCAM counters every epoch. We ran this experiment with other values and the results for those values are qualitatively similar. Additive increase in AM and AA has slow convergence when $R_{i,s}(t)$ changes since it takes a long time to increase the step size. Although MA reaches the goal fast, it takes long for it to decrease the step size and converge to the goal. Therefore, we use multiplicative increase and decrease (MM) for changing the step size; we have also experimentally verified its superior performance.

As an aside, note that our problem is subtly different from fair bandwidth allocation (*e.g.,* as in TCP). In our setting, different tasks can have different $R_{i,s}$, and the goal is to keep their allocated resources, $r_{i,s}$, above $R_{i,s}$ for more tasks, but fairness is a non-goal. By contrast, TCP attempts to converge to a target fair rate that depends upon the bottleneck bandwidth. Therefore, some of the intuitions about TCP's control laws do not apply in our setting. In the language of TCP, our approach is closest to AIAD, since our step size is independent of $r_{i,s}$. In contrast to AIAD, we use large steps when $r_{i,s}$ is far from $R_{i,s}$ for fast convergence, and we use small step sizes otherwise for saving resources by making $r_{i,s}$ close to $R_{i,s}$.

[5] In our implementation, a task is considered rich only if $a_{i,s} > A + \delta$, where A is the target accuracy bound. The δ is a hysteresis threshold that prevents a task from frequently oscillating between rich and poor states.

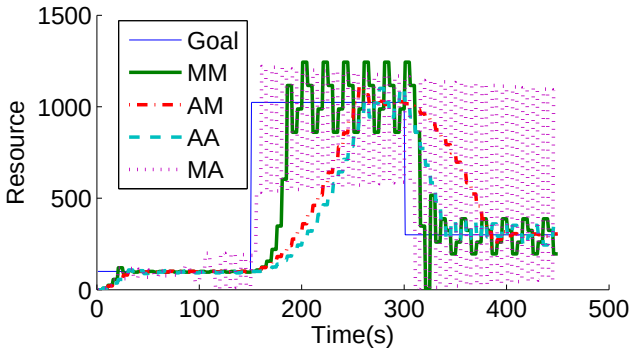


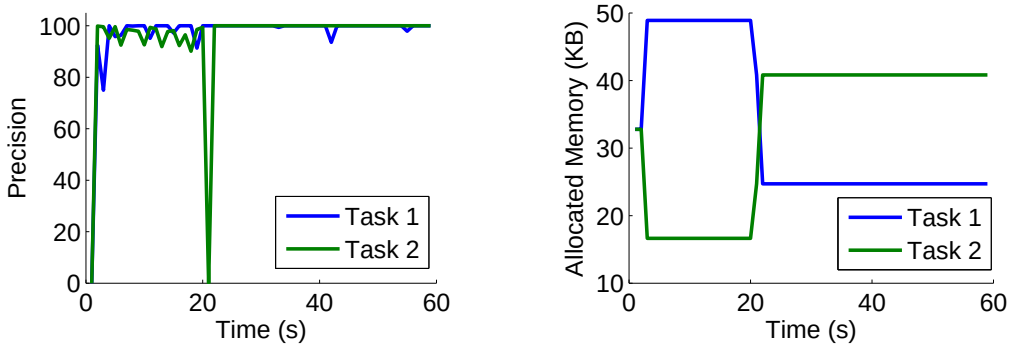
Figure 2.5: *Comparing step updates algorithms*

Spare TCAM capacity, or *headroom*: Since $R_{i,s}$ can change over time because of traffic changes, running the system close to the capacity can result in low task satisfaction. Therefore, DREAM maintains *headroom* of TCAM counters (5% of the total TCAM capacity in our implementation), and immediately rejects a new task if the headroom is below a target value on any switch for the task. This permits the system to absorb fluctuations in total resource usage, while ensuring high task satisfaction.

However, because it is impossible to predict traffic variability, DREAM may infrequently drop tasks when headroom is insufficient.[6] In our current design, operators can specify a *drop priority* for tasks. DREAM lets the poor tasks with low drop priority (*i.e.,* those that should be dropped last) steal resources from those tasks with high drop priority (*i.e.,* those that can be dropped first). When tasks with high drop priority get fewer and fewer resources on some switches, and remain poor for several consecutive epochs, DREAM drops them, to ensure that they release resources on *all* switches.

DREAM does not literally maintain a pool of unused TCAM counters as headroom. Rather, it always allocates enough TCAM counters to all tasks to maximize accuracy in the presence of traffic changes, but then calculates *effective headroom* when a new task arrives. One estimate of effective headroom is $s_R - s_P$ (the sum of the step sizes of the rich tasks minus that of the poor tasks). However, this can under-estimate headroom: a rich task may have more resources than it needs, but its step size may have converged to a small value and may not accurately reflect how many resources it can give up while still maintaining accuracy. Hence, DREAM introduces a *phantom* task on each switch whose resource requirement is equal

[6]Here, we assume that perimeter defenses are employed (*e.g.,* as in data centers), so malicious traffic cannot trigger task drops. In future work, we plan to explore robustness to attack traffic.



(a) *Estimated precision* **(b)** *Allocation*

Figure 2.6: *Resource allocation example*

to the headroom. Rich tasks are forced to give up resources to this phantom task, but, when a task becomes poor due to traffic changes, it can steal resources from this phantom task (this is possible because the phantom task is assigned the lowest drop priority). In this case, if r_{ph} is the phantom task's resources, the effective headroom is $r_{ph} + s_R - s_P$, and DREAM uses this to determine if the new task should be admitted.

We now illustrate how the resource allocator works using a simple example in SCREAM (Figure 2.6). We ran an experiment on two heavy hitter (HH) detection tasks on source IP using Count-Min sketch on a single switch. Each task monitors a chunk of a packet trace [20] starting from time 0. Figure 2.6a shows the estimated accuracy in terms of *precision* (the fraction of detected true HHs over detected ones) of tasks over time, and Figure 2.6b shows the allocated resources per task. In the beginning, both tasks get equal resources (32 KB), but task 1 cannot reach the 80% accuracy bound at time 3 while task 2 has very high accuracy. Therefore, the resource allocator takes memory resources (16 KB) from task 2 and gives to task 1. At time 20, we increase the skew of volume of traffic from source IPs for task 1 and decrease it for task 2. As a result, task 2 requires more resources to reach 80% accuracy bound, thus its estimated accuracy degrades at time 20. The resource allocator responds to the accuracy decrease by iteratively allocating more resources to task 2 (first 8 KB then 16 KB) until it exceeds the bound.

The dynamic resource allocation algorithm needs tasks to give an estimate of their current accuracy globally and per switch. To understand that, we first need to know how each task works in detail as discussed in the next section.

2.6 Task Implementation

In DREAM and SCREAM, task objects implement individual task instances. In this section, we describe how we improved the implementation of tasks to handle dynamic and different amount of resources from different switches and how we generalized the algorithms over multiple tasks.

2.6.1 Flow-based Tasks

We begin by describing a generic algorithm that captures task object functionality. An important component of this algorithm is a *task-independent* iterative algorithm for configuring TCAM counters across multiple switches. This algorithm leverages TCAM properties, and does not depend upon details of each task type (*i.e.,* HH, HHH or CD). We conclude the section with a discussion of the *task-dependent* components of the generic algorithm.

This deliberate separation of functionality between generic, task-independent, and task dependent parts enables easier evolution of DREAM. To introduce a new task type, it suffices to design new algorithms for the task-dependent portion, of which the most complicated is the accuracy estimator.

2.6.1.1 A Generic Algorithm for Task Objects

A DREAM task object implements Figure 2.7; each task object runs on the SDN controller. This algorithm description is generic and serves as a design pattern for any task object independent of the specific functionality it might implement (*e.g.,* HH, HHH or CD).

Each newly admitted task is initially allocated one counter to monitor the prefix defined by the task's flow filter (Section 2.4). Task object structure is simple: at each *measurement interval*, a task object performs six steps. It fetches counters from switches (line 2), creates the report of task (line 3) and estimates its accuracy (line 4). Then, it invokes the per-switch allocator (line 5, Section 2.5) and allows the task to update its counters to match allocations and to improve its accuracy (line 6). Finally, the task object installs the new counters (line 7).

```
1 foreach measurement iteration do
2 |   counters=fetchCounters(switches)
3 |   report = createReport(counters)
4 |   (global, locals) = t.estimateAccuracy(report, counters)
5 |   allocations = allocator.getAllocations(global, locals)
6 |   counters = configureCounters(counters, allocations)
7 |   saveCounters(counters,switches)
```

Figure 2.7: *DREAM task object implementation*

Of these six steps, one of them `configureCounters()` can be made *task-independent*. This step relies on the capabilities of TCAMs alone and not on details of how these counters are used to measure HHs, HHHs or significant changes. Two other steps are *task-dependent*: `createReport`, and `estimateAccuracy`.

2.6.1.2 Configuring Counters for TCAMs

Overview: After the resource allocator assigns TCAM counters to each task object on each switch, the tasks must decide how to *configure* those counters, namely which traffic aggregates to monitor on which switch using those counters (`configureCounters()` in Figure 2.7). A measurement task cannot monitor every flow in the network because, in general, it will not have enough TCAM counters allocated to it. Instead, it can measure traffic aggregates, trading off some accuracy. For example, TCAM-based measurement tasks can count traffic matching a traffic aggregate expressed as a wild-card rule (*e.g.,* traffic matching an IP prefix).

The challenge then becomes choosing the right set of prefixes to monitor for a sufficiently accurate measurement while bounding resource usage. A task-independent iterative approach works as follows. It starts by measuring an initial set of prefixes in the prefix trie for the *packet header field* (source or destination IP) that is an input parameter to our tasks. Figure 2.8 shows an example prefix trie for four bits.

Then, if the count on one of the monitored prefixes is "interesting" from the perspective of the specific task (*e.g.,* it reveals the possibility of heavy hitters within the prefix), it *divides* that prefix to monitor its children and use more counters. Conversely, if some prefixes are "uninteresting", it *merges* them to free counters for more useful measurements.

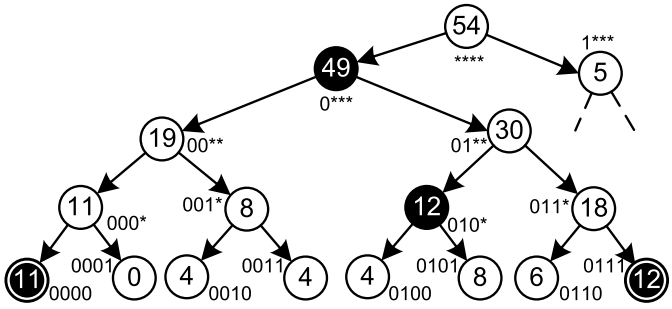


Figure 2.8: *A prefix trie of source IPs where the number on each node shows the bandwidth used by the associated IP prefix in Mb in an epoch. With threshold 10, the nodes in double circles are heavy hitters and the nodes with shaded background are hierarchical heavy hitters.*

While this approach is task-independent, it depends upon a task-dependent component: a prefix *score* that estimates how "interesting" the prefix is for the specific task. Finally, DREAM can only measure network phenomena that last long enough for this iterative approach to complete (usually, on the order of several seconds).

Divide-and-Merge: Figure 2.9 describes this *divide-and-merge* algorithm in detail. The input to this algorithm includes (a) the current configuration of counters allocated to the task and (b) the new resource allocation. The output of this algorithm is a new configuration describing the prefixes to be monitored in the next measurement interval.

```
1  computeScores(counters)
2  while F = {over-allocated switches} ≠ Φ do
3  |   merge(cover(F, counters, allocations))
4  repeat
5  |   m = maxIndex(counters.score)
6  |   F = toFree(m, allocations)
7  |   solution.cost=0
8  |   if F ≠ Φ then
9  |   |   solution=cover(F, counters, allocations)
10 |   if solution.cost<m.score then
11 |   |   divide(m)
12 |   |   merge(solution)
13 until no counter to divide
```

Figure 2.9: *Divide-and-merge algorithm to update counters in flow-based tasks*

In the first step, the algorithm invokes a task-dependent function that returns the score associated with each prefix currently being monitored by the task object (line 1). We describe prefix scoring later in this section, but scores are non-negative, and the *cost* of merging a set of prefixes is the sum of their score. Now,

if the new TCAM counter allocation at some switches is lower than the current allocation (we say that these switches are *overloaded*), the algorithm needs to find prefixes to merge. It iteratively finds a set of prefixes with minimum cost that can be merged into their ancestors, thereby freeing entries on overloaded switches (lines 2-3). We describe how to find such candidate prefixes (`cover()` function) below. After merging, the score of the new counter will be the total score of merged counters.

Next, the algorithm iteratively divides and merges (lines 4-13). First, it picks the counter with maximum score to divide (line 5) and determines if that results in overloaded switches, designated by the set F (line 6). If F is empty, for example, because the resource allocator increased the allocation on all switches, no merge is necessary, so the merge cost is zero. Otherwise, we use the `cover()` function to find the counters to merge (line 9). Next, if the score of the counter is worth the cost, we apply divide and merge (lines 10-12). After dividing, the score of children will be half of the parent's score [154]. The algorithm loops over all counters until no other counter is worth dividing.

A similar algorithm has been used for individual measurement tasks (*e.g.,* HH [89], HHH [112], and traffic changes [155]). In this chapter, we provide a general task-independent algorithm, ensuring that the algorithm uses bounded resources and adapting to resource changes on multiple switches.

On multiple switches: We now explain how divide-and-merge works across multiple switches. We consider tasks that measure phenomena on a single packet header field, and we leave to future work extensions to multiple fields. For ease of exposition, we describe our approach assuming that the user specifies the source IP field for a given task. We also assume that we know the ingress switches for each prefix that a task wishes to monitor; then to monitor the prefix, the task must install a counter on all of its ingress switches and later sum the resulting volumes at the controller.

Dividing a prefix may need an additional entry on *multiple* switches. Formally, if two sibling nodes in the prefix trie, A and B, have traffic on switch sets S_A and S_B, monitoring A and B needs one entry on each switch in S_A and one on each switch in S_B, but monitoring the parent P needs one entry on $S_P = S_A \cup S_B$ switches. Therefore, merging A and B and monitoring the parent prefix frees one entry on $S_A \cap S_B$. Conversely, dividing the parent prefix needs one additional entry on switches in $S_A \cap S_B$. For

example, suppose that $S_{0000} = \{1,2\}$ and $S_{0001} = \{2,3\}$ in Figure 2.8 where set elements are switch ids. Merging S_{0000} and S_{0001} saves an entry on 2.

The challenge is that we may need to merge more than two sibling prefixes to their ancestor prefix to free an entry in a switch. For example, suppose that $S_{0010} = \{3,4\}$ and $S_{0011} = \{4,1\}$. To free an entry on switch 3, we must merge S_{0001} and S_{0010}. Therefore, we merge all four counters to their common ancestor $00**$. [7]

To generalize, suppose that for each internal node j in the prefix trie (ancestor of counters), we know that merging all its descendant counters would free entries on a set of switches, say T_j. Furthermore, let the cost for node j be the sum of the scores of the descendant monitored prefixes of j. The function `cover()` picks those T_j sets that cover the set of switches requiring additional entries, F, with minimum total cost. There are fast greedy approximation algorithms for Minimum Subset Cover [144].

Finally, we describe how to compute T_j for each internal node. For each node j, we keep two sets of switches, S_j and T_j. S_j contains the switches that have traffic on j and is simply $S_{j_{left}} \cup S_{j_{right}}$ when j has two children j_{left}, j_{right}. T_j contains the switches that will free at least one entry if we merge all its descendant counters to j. Defining it recursively, T_j includes $T_{j_{left}}$ and $T_{j_{right}}$, and contains (by the reasoning described above) the common entries between the switches having traffic on the left and right children, $S_{j_{left}} \cap S_{j_{right}}$. T_j is empty for prefixes currently being monitored.

2.6.1.3 Task-dependent Algorithms

Beyond these task-independent algorithms, each task object implements three task-dependent algorithms. We present the task-dependent algorithms for HH, HHH, and CD tasks. A key task-dependent component is accuracy estimation and will be discussed in Section 2.7.1. The other task-dependent algorithms for these tasks are summarized in Table 2.1, but we discuss some of the non-trivial algorithms below.

Heavy hitter (HH): A heavy hitter is a traffic aggregate that exceeds a specified volume. For example, we can define heavy hitters as the source IP addresses whose traffic exceeds a threshold θ over a measurement

[7] Although we could just merge those two to $00**$, this creates overlapping counters that makes the algorithm more complex and adds delay in saving rules at switches.

Task	Create report	Score
HH	Report exact counters with $volume > \theta$	$\frac{volume}{\#wildcards+1}$
HHH	Traverse prefix trie bottom-up and report a prefix h if $volume_h - \sum_i volume_i > \theta$ where i is a descendant detected HHH of h [39]	$volume$
CD	Report exact counters with $\lvert volume - mean \rvert > \theta$	$\frac{\lvert volume-mean \rvert}{\#wildcards+1}$

Table 2.1: *Task dependent methods for flow-based tasks*

epoch. Figure 2.8 shows an example of bandwidth usage for each IP prefix during an epoch. With a threshold of $\theta = 10Mb$, there are a total of two leaf heavy-hitters shown in double circles. Our divide-and-merge approach iteratively drills-down to these two leaves.

Reporting: The task object simply outputs counters of exact IP addresses (trie leaves) whose traffic volume exceeds the threshold.

Prefix Scoring: Recall that a prefix score is used to determine when to divide a prefix. It is insufficient to simply assign a score to a prefix proportional to its traffic. Intuitively, prefixes closer to the leaves should be assigned higher scores, since doing so will drive the algorithm towards solutions (since HHs are defined on exact IP addresses) [89]. To achieve this depth-first drill-down, we use $\frac{volume}{\#wildcards+1}$ for the score of a prefix as a heuristic where the number of wildcard bits is a metric of how far a prefix is from the leaves.

Hierarchical heavy hitters (HHH): A variant of heavy hitters, called *Hierarchical Heavy Hitters (HHHs)* [39] is useful for anomaly detection [154] and DDoS detection [132]. A HHH is (recursively) defined by the longest IP prefixes that contribute traffic exceeding a threshold θ of total traffic, after *excluding* any HHH descendants in the prefix trie. For example in Figure 2.8, prefix 010* is a HHH as IPs 0100 and 0101 *collectively* have large traffic, but prefix 01** is not a HHH because excluding descendent HHHs (010* and 0111), its traffic is less than the threshold.

Reporting: Given the monitored traffic volumes at different prefixes in the trie, we simply use the recursive definition above to identify and report HHH prefixes.

Prefix Scoring: We use the volume of a monitored prefix as its score as discussed in [112].

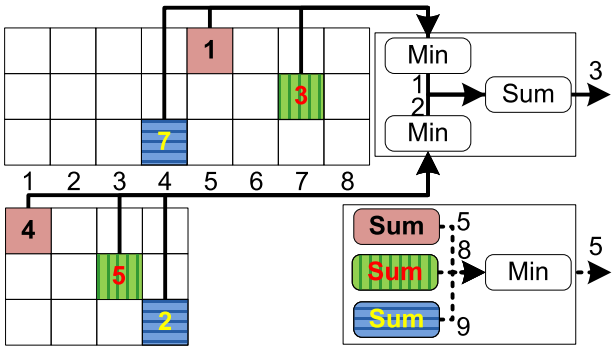


Figure 2.10: *Merging two Count-Min sketches with different sizes*

Change detection (CD): A simple way to define the traffic change of a prefix is to check if the difference between its current volume and a moving average of its volume exceeds a specified threshold. In this sense, change detection is similar to HH detection: a change is significant if $|volume - mean| > \theta$. Thus, for change detection, reporting and prefix scoring are similar to those for HH tasks (Table 2.1): wherever volume is used in HH tasks, $|volume - mean|$ is used for CD.

2.6.2 Sketch-based Tasks

In SCREAM, tasks at the controller configure sketch counters at switches, fetch counters and prepare reports. In order to prepare reports, tasks need to find instances of HHs, HHHs, or SSDs. In this section, we first describe how to approximate traffic counts for HHs and HHHs, and connection counts for SSDs using Count-Min and HyperLogLog sketches on multiple switches. These algorithms execute at the controller, and are specific to a given task type. Then, we describe an algorithm independent of task type that, from the derived counts, estimates and reports instances of HHs, HHHs and SSDs that exceed the specified threshold.

Although there have been many sketch-based algorithms [150], we improve upon them in the following ways. We introduce novel techniques to merge sketches with different sizes from multiple switches, leverage hierarchical grouping with adjustable overhead to find instances of HHs, HHHs and SSDs, and adapt the design of the SSD task to be unbiased and provide stable accuracy. We describe these improvements below.

Heavy Hitter (HH): If a prefix has traffic on one switch, we estimate traffic size by the minimum of counts from different rows of the counter array in the Count-Min sketch approximation algorithm (Section 2.2.2). However, a heavy hitter prefix may have traffic from multiple switches. One approach [3] in this case is to simply sum up the Count-Min sketch arrays fetched from different switches into a single array ($A_{new} = \sum_s A_s$ for each switch s) and run the algorithms as if there is only one sketch. However, in SCREAM, the resource allocator sizes the sketches at each switch differently, so each sketch may have an array of different widths and cannot be summed. For example, Figure 2.10 shows the counter arrays for two Count-Min sketches with three rows and different widths that cannot be directly summed.

A natural extension for sketches of different sizes is to find the corresponding counter for each prefix at each row and sum the counters at similar rows across sketches. The approximated count will be their minimum: $min_i(\sum_s A_s[i, h_i(x)])$. For example, say an item on the first sketch maps to counters with index 5, 7, and 4, and on the second sketch maps to 1, 3, and 4. The approximation will be: $min(A_1[1,5]+A_2[1,1],\ A_1[2,7]+A_2[2,3],\ A_1[3,4]+A_2[3,4])$. In Figure 2.10 (right bottom) we connect counters with the same color/pattern to the corresponding sum boxes to get 5 as the final result.

However, because Count-Min sketch always over-approximates due to hash collisions, we can formulate a method that generates smaller, thus more accurate, approximations. The idea is to take the minimum of corresponding counters of a prefix inside each sketch and then sum the minima: $\sum_s min_i(A_s[i, h_i(x)])$. For the above example, this will be $min(A_1[1,5],A_1[2,7],A_1[3,4]) + min(A_2[1,1],A_2[2,3],A_2[3,4])$ (Figure 2.10, the top merging module with solid lines to counter arrays which approximates the size as 3 instead of 5). This approximation is always more accurate because the sum of minimums is always smaller than minimum of sums for positive numbers. In all of this, we assume that each flow is monitored only on one switch (*e.g.,* at the source ToR switches).

Hierarchical Heavy Hitter (HHH): Recall that HHHs are defined by the longest prefixes exceeding a certain threshold in aggregate volume after *excluding* any HHH descendants in the prefix tree. Figure 2.8 shows an example of a prefix tree for four bits. With a threshold of $\theta = 10Mb$, prefix 010* is a HHH as IPs

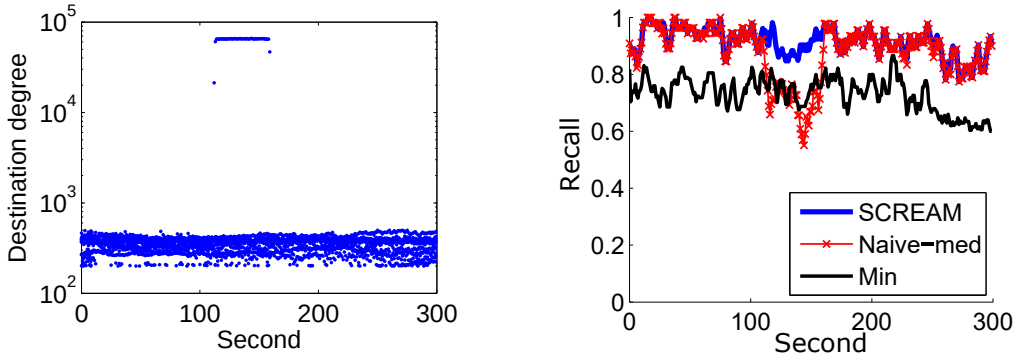
0100 and 0101 collectively have large traffic, but prefix 01** is not a HHH because excluding descendent HHHs (010* and 0111), its traffic is less than the threshold.

We leverage multiple sketches to find HHHs [39]. We need a sketch for each layer of the prefix tree to estimate the size of prefixes at different levels. For a HHH without any descendant HHH, the approximation function works the same as HH detection task. However, for other HHHs, we need to exclude the size of descendant HHHs. Thus, during a bottom up traversal on the tree, SCREAM tracks the total size of descendant HHHs already detected and subtracts that size from the approximation for the current prefix [39].

Super Source/Destination (SSD): SSD detection needs to count distinct items instead of the volume of traffic, so we replace each counter in the Count-Min sketch array with a distinct counter [35]. Therefore, each sketch has $w \times d$ distinct counters; we used the HyperLogLog [51] distinct counter because its space usage is near-optimal and it is easy to implement [69]. However, distinct counters may under-approximate or over-approximate with same probability, so picking the minimum can cause under-approximation and result in many missing items even with a large Count-Min array. For example, suppose that there is no collision in a Count-Min sketch, we have d distinct counters for a source IP that have Gaussian error, and we pick the minimum, it is more likely to pick the one that under-approximates. Figure 2.11b shows the recall (detected fraction of true SSDs) of SSD detection given fixed resources in simulation over a CAIDA traffic trace [20]. The under-approximation of picking the minimum resulted in missing more than 20% of SSDs. Unfortunately, this will become worse for larger Count-Min sketches with fewer collisions. Thus, the SSD approximation, even for a single sketch, cannot use the minimum of corresponding counters.

To counter this bias, unlike prior work [35, 150, 63], we pick the median instead of minimum. Then, to remove the median's bias towards Count-Min hash collisions, we remove the average error of the sketch from it in Equation 2.1 where A is the Count-Min sketch array of width w and T is the sum of distinct items of prefixes. Equation 2.1 is unbiased since we can interpret the over/under-approximations of distinct counters as random positive and negative updates on the sketch and use the proof in [94].

$$\frac{median_i(A[i,h(x)]) - T/w}{1 - 1/w} \tag{2.1}$$



(a) *Destination degree* **(b)** *Recall*

Figure 2.11: *Unbiasing detection of destination IPs contacted by* > 200 *source IPs on Count-Min sketch* $(w = 1320, d = 3)$ *plus HyperLogLog* $(m = 16)$

However, when the number of sources per destination IP is highly skewed (*e.g.,* in a DDoS attack) removing the average error (T/w) in Equation 2.1 can result in missing SSDs. For example, Figure 2.11a shows the degree of super destinations over time in the ground-truth where a specific destination has a large number of distinct sources from time 110 to 160. Figure 2.11b shows that the recall for an approach that uses Equation 2.1, named Naive-med, drops during the DDoS attack. These missed SSDs result from the fact that Equation 2.1 compensates for the average Count-Min collision from every counter, but for skewed traffic a few large items that increase average error significantly only collide with a few counters. Thus, reducing this large error from every median approximation causes an under-approximation of the total count, and results in missing true SSDs.

Instead, we refine the average error estimate by removing those very large prefixes from it, but must first detect them using two steps. In the first step, we use the average error just to detect very large prefixes, set L (as mentioned before, this causes an under-approximation, but is still sufficient to detect very large SSDs.). In the second round, we reduce the adjusted average error, $\frac{T - \sum_{k \in L} c_k^{est}}{w}$, from the medians, where c_k^{est} is the estimated count for item k. This results in high recall, independent of traffic skew (Figure 2.11b). This does not come at the cost of increased false positives, and SCREAM's precision is also high.

```
1 Function approximate(prefix, sketches)
2     for i = 1...d do
3         M_{new,*} = 0
4         for s in sketches do
5             DC_{s,i} =distinct counter in A_s[i, h(prefix)]
6             for k = 1...m do
7                 M_{new,k} = max(M_{new,k}, M_{DC_{s,i},k})
8         c_i^{est} = hyperloglog(M_{new,*}).approximate()
9     return unbias(median_i(c_i^{est}))
```

Figure 2.12: *Approximate function for SSD*

Multiple switches: If a prefix has traffic from multiple sketches, summing the number of distinct items from different sketches over-approximates the number of distinct items because two distinct counters may have counted similar items. For example, two switches that forward traffic from a source IP prefix may see traffic to a common destination IP. However, the common destination IP should only be counted once in the degree of the source IP prefix. We can combine two HyperLogLog distinct counters, DC_1 and DC_2, with m replica counters by taking the maximum of each corresponding replica counter to make a new distinct counter [51]: $M_{new,k} = max(M_{DC_1,k}, M_{DC_2,k})$ for $k = 1 \ldots m$.

To leverage this, we keep a fixed number of replica in distinct counters of different sketches and only change the width of the array in Count-Min sketch (w) based on the allocated resources. Again, having Count-Min sketches with different widths, we cannot use the traditional approach of merging distinct counters with the same index in Count-Min counter array [63]. Instead, we find corresponding distinct counters for each specific query in each row and merge them.

Figure 2.12 summarizes how we use this idea to approximate the degree of each prefix when Count-Min sketches may have different widths. For each d rows of Count-Min sketches, we find the corresponding distinct counters for a prefix in each sketch (lines 2-5). Then, we merge replicas from these distinct counters (lines 6-7) and approximate the number of distinct items using the new replica counters similar to a single HyperLogLog sketch [51] (line 8). Now similar to the case of a single switch, we approximate the degree of the SSD using the unbiased median approach (line 9).

2.6.2.1 Reporting HHs, HHHs and SSDs

So far, we have presented ways of approximating traffic volumes and connection counts. However, we also need an efficient way of determining which IP prefixes contain HHs, HHHs or SSDs. In the data plane of each switch, SCREAM uses Count-Min sketches to count traffic. A single Count-Min sketch can only approximate the count given a prefix. Exploring all prefixes at the controller is impossible, so SCREAM uses a hierarchy of Count-Min sketches to identify the actual prefixes [36]. It employs a Count-Min sketch for each level of prefix tree (*e.g.,* 16 sketches for a task with flow filter of 10.5/16), where the sketch on level l (from leaves) ignores l least significant IP bits.[8] Note that to find HH/SSD IP prefixes that are not exact, we can start the tree from a level > 0.

Figure 2.13 shows an algorithm that does not depend on the task type. In line 2, the algorithm `approximates` the size of a prefix tree node by combining multiple sketches (using algorithms described above). Then, it traverses the prefix tree (lines 3-6). If the approximation is above the threshold, it goes deeper to find items for output. This algorithm relies on the observation that if a prefix's size is not over the threshold, its ancestors sizes are not too.

For example in Figure 2.8, it starts from the root and goes in the left branches until it finds heavy hitter 0000, but later when it reaches prefix 001*, it does not need to check its children. The `updateOutput` function for HH/SSD detection is simply to add the prefix for a leaf node (path from the root in the prefix tree) to the output. However, for HHH detection, we only add the prefix into output if its size remains larger than threshold after gathering its descendant HHHs and excluding their size.

Many techniques are proposed to identify items to query (reverse a sketch) [37, 19, 129]. At the core, all use multiple sketches and apply group testing to reverse the sketch, but their grouping is different. We use hierarchical grouping [37] because it is enough for our tasks, is fast and simple and has tunable overhead comparing to some alternatives. For example, OpenSketch used Reversible sketch [129] with

[8] It is possible to have a sketch for each $g > 1$ levels of the tree but with more overhead at the controller to enumerate 2^g entries at each level. Our implementation is easily extendible for $g > 1$.

```
1 Function createReport (prefix, output)
2     e=approximate(prefix, prefix.sketches)
3     if e≥threshold then
4         foreach child of prefix do
5             createReport(child, output)
6         updateOutput(prefix, output)
```

Figure 2.13: *Generic algorithm to create output for sketch-based tasks*

fixed high memory usage of 0.5 MB. Our work generalizes prior work that has used hierarchical grouping for HHs and HHHs, but not for SSDs [36, 37].

2.7 Accuracy Estimation

To support dynamic resource allocation, we need algorithms that can estimate the instantaneous accuracy for individual tasks, even when the traffic for a task spans multiple switches. In addition to informing resource allocation, our accuracy estimates can give operators some understanding of the robustness of the reports. Our accuracy estimators discussed in this section consider two accuracy metrics: *precision*, the fraction of retrieved items that are true positives; and *recall*, the fraction of true positives that are retrieved. For these definitions, an item refers to a HH, HHH, change or super source/destination. Depending on the type of measurement task, DREAM and SCREAM estimate one of these accuracy measures to determine resource allocation.

The key challenge is that we do not have an *a priori* model of traffic and it takes too much overhead to understand traffic characteristics by measuring traffic. Instead, our accuracy estimator only leverages the collected counters of the task.

2.7.1 Flow-based Tasks

Our key idea for estimating the accuracy for flow-based tasks is to leverage the measurement counters to find tight bounds on the error in the output of tasks.

Heavy hitter detection: For our TCAM-based algorithm, all detected HHs are true, which means the precision is always one in this algorithm. For this reason, we use recall as a measure of accuracy for HH detection. Doing so requires an estimate of the number of true HHs the algorithm misses. We use the

smaller of the following two bounds to estimate the missed heavy hitters under a non-exact prefix. First, a prefix with b wildcard bits cannot miss more than 2^b heavy hitters. For example, prefix $0***$ in Figure 2.8 has 8 heavy hitters at most. Second, if the volume of the prefix is v, there can only be $\lfloor \frac{v}{\theta} \rfloor$ missed heavy hitters. This bound for prefix $0***$ will be 4.

Finally, we need to estimate both local and global recall (Section 2.5). We compute the local recall for a switch based on detected HHs, and we estimate missed HHs from prefixes that have traffic on the switch. However, there are cases where only a subset of switches is bottlenecked (*i.e.,* they have used all available counters, so it is not possible to further divide prefixes). In this case, we only consider missed HHs on these switches.

Hierarchical heavy hitter detection: For HHHs, our algorithm estimates precision by determining whether a detected HHH is a true positive or a false positive. Our algorithm assigns a precision value to each detected HHH: the value is either 0 if it is a false positive, 1 if a true positive, or fractional if there is ambiguity in the determination, as discussed below. The overall accuracy estimate is an average of these values. The method for making these value assessments is different for HHHs without and with detected descendant HHHs.

If a detected HHH h has no detected descendant HHHs (*e.g.,* 0000, 010*, 0111 in Figure 2.8), it is a false positive HHH if it has been detected instead of one of its descendants. So, for it to be a true positive HHH, we need to ensure that none of its descendants could have been a HHH. There are three cases. (1) h is an exact IP. (2) We monitored the descendants of h and their volume is below the threshold θ. For example, if we monitor 0100 and 0101, we can confirm that the detected HHH 010* is a true one. In these two cases, it is easy to tell h is a true HHH. (3) We only monitored h and do not know about its descendants. If h has a count larger than 2θ, then h cannot be a true HHH, because the volume of at least one of its children must be above θ. If the volume is smaller than 2θ, either the detected prefix or one of its sub-prefixes is HHH, so we set its precision value to 0.5.

For an HHH h with detected descendant HHHs, the error in the detected descendant HHHs can make h a false HHH. For example in Figure 2.8, suppose that we report 0000, 010* and 011* as HHHs. Now,

the volume for 0*** excluding descendant HHHs will be 8 because of false detection of 011*. Therefore, instead of 0***, we detect **** as HHH. In this scenario, we have over-approximated the traffic from descendant HHHs of ****. In the worst case, the over-approximated traffic has been excluded from a child of the detected HHH. Thus, for each child prefix, we find if adding up these over-approximations could make them a HHH. If any child with a new volume becomes HHH, the parent cannot be, so as a heuristic, we halve the precision weight of h. The over-approximation for a HHH confirmed to be true is 0, and the over-approximation for other HHHs can be at most $volume - \theta$.

The global precision is the average precision value of detected HHHs. To compute the local precision per switch, we compute the average precision value of HHH prefixes from each switch. If a HHH has traffic from multiple switches, we give the computed precision value only to bottleneck switches, and precision 1 to other switches.

For HHH tasks, recall can be calculated similar to HH tasks. We have experimentally found that, for HHH, recall is correlated with precision.

Change detection: change detection is similar to HH detection: a change is significant if $|volume - mean| > \theta$. Thus, for change detection, accuracy estimation algorithm is similar to the one for HH tasks: wherever volume is used in HH tasks, $|volume - mean|$ is used for CD.

2.7.2 Sketch-based Tasks

There are two key ideas in our accuracy estimator for sketches: (1) applying probabilistic bounds on *individual* counters of detected prefixes, and (2) tightening the bounds by *separating* the error due to large items from the error due to small items.

Heavy hitter detection: Count-Min sketch always over-approximates the volume of a prefix because of hash collisions; therefore, its recall is 1. We compute precision by averaging the probability that a detected HH j is a true HH, p_j. We start with the case where each HH has traffic from one switch and later expand it for multiple switches. The strawman solution is to estimate the probability that an item could remain a HH even after removing the collision error of *any* other item from its *minimum* counter. The resulting

estimated accuracy under-estimates the accuracy by large error mainly because, for skewed traffic, a few large items make the probabilistic bound on the error loose since the few large items may only collide on a few counters. Our approach treats the counters in each row separately and only uses the probabilistic bound for the error of small undetected items.

A strawman for estimating p_j. p_j is the probability that the real volume of a detected HH is larger than the threshold θ, $p_j = P(c_j^{real} > \theta)$. In other words, an item is a true HH, if the estimated volume remains above the threshold even after removing the collision error. We can estimate the converse (when the collision error is larger than the difference between estimated volume and threshold (Equation 2.2) using the Markov inequality. To do this, we observe that each counter has an equal chance to match traffic of every item, so the average traffic on each counter of each row is $\frac{T}{w}$ (T is the total traffic, and w is the number of counters for each hash function) [37]. Using the Markov inequality, the probability that the collision exceeds $c_j^{est} - \theta$ is smaller than $\frac{T}{w(c_j^{est}-\theta)}$. However, since Count-Min sketch picks the minimum of d independent counters, the collisions of all counters must be above the bound. Putting this together, we get Equation 2.3:

$$P(c_j^{real} > \theta) = P(c_j^{est} - e_{cm} > \theta) = 1 - P(e_{cm} \geq c_j^{est} - \theta) \tag{2.2}$$

$$P(c_j^{real} > \theta) > 1 - (\frac{T}{w(c_j^{est} - \theta)})^d \tag{2.3}$$

Unfortunately, as Figure 2.14 shows, the resulting estimated precision is far from the actual precision, which leads to inefficient resource allocation. The reason is that, for skewed traffic, a few large items can significantly increase average error $\frac{T}{w}$, but only collide with a few counters.

Our solution: separate the collision of detected HHs on each counter. We can leverage individual counters of detected HHs in two ways to tighten the p_j estimation. First, instead of using the final estimated volume for a HH (c_j^{est}) that is the smallest in all rows, we use the individual counters for each hash function h_i separately ($c_{i,j}^{est}$) that can be larger and provide tighter bounds in Equation 2.4.

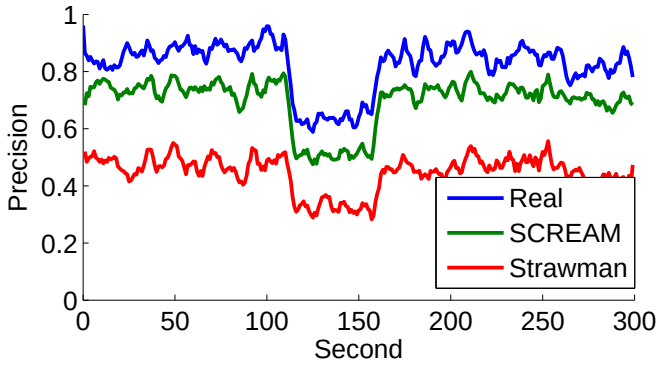


Figure 2.14: *HH detection accuracy estimation for Count-Min sketch ($w = 340, d = 3$)*

$$P(c_j^{real} > \theta) > 1 - \prod_{i=1}^{d} \frac{T}{w(c_{i,j}^{est} - \theta)} \tag{2.4}$$

Second, we know the counter indices for detected HHs and can find if they collide with each other. Therefore, we separate the collisions of detected HHs from collisions with other small items. Using this, we can lower the estimate for average collision traffic in the Markov inequality by removing the traffic of detected HHs, resulting in a tighter estimate.[9] We now describe the details of this technique.

There are two cases where a detected HH is not a real HH: (1) when detected HHs collide with each other; (2) when a detected HH does not collide with other detected HHs, but collides with multiple IPs with low counts, which together inflate the traffic count above the threshold. For case (1), we can easily check if a detected HH collides with other detected HHs by checking if the index of its counter is hit by another HH. If $B_{i,j}$ is the set of other HHs that collide on the ith counter of HH j, we just remove the estimated volume of those HHs from the counter by using $c_{i,j}^{est} - \sum_{k \in B_{i,j}} c_k^{est}$ instead of $c_{i,j}^{est}$. The estimated volume of HHs in set $B_{i,j}$ may be an over-approximation and removing them from $c_{i,j}^{est}$ makes our p_j estimate conservative. For case (2), we use the Markov inequality to bound the collision of undetected small items. However, instead of T, now we should use the traffic of only undetected small items. Let A be the set of detected HHs whose estimation is not affected by other HHs (no hit on minimum counter). Replacing T with $T - \sum_{k \in A} c_k^{real}$ in Equation 2.4, we can estimate p_j in Equation 2.5. However, we do not know

[9] Cormode [38] also used this technique to find a resource-accuracy trade-off for Count-Min sketch assuming the skew of traffic is known, but our goal is to estimate p_j for each HH without assuming a model of traffic.

$c^{real}_{k\in A}$ because of the over-approximations of counts in the sketch. Thus, as an estimate, we use $c^{est}_{k\in A}$ after reducing the average collision error of only small items from it. Figure 2.14 shows that our estimation based on Equation 2.5 is close to the real precision, even under traffic dynamics. In Section 2.8.3.3, we have validated that this improvement applies across all the traces we have used in our evaluations, and that this improvement is essential for SCREAM.

$$P(c^{real}_j > \theta) > 1 - \prod_{i=1}^{d} \frac{T - \sum_{k\in A} c^{real}_k}{w(c^{est}_{i,j} - \sum_{k\in B_{i,j}} c^{est}_k - \theta)} \tag{2.5}$$

Multiple switches: As described in Section 2.5, our resource allocator estimates a global accuracy for the task, as well as a per-switch local accuracy (Section 2.5). It uses these to add/remove resources from switches. Similar to the single switch case, we compute the global precision by finding p_j for each detected HH.

Markov's inequality is too loose when a HH has traffic from a set of switches, so the single-switch accuracy estimator does not work well. The reason is that the network-wide collision (a random variable) is the sum of collisions at individual switches (sum of random variables) [29]. However, since the collision on a sketch is independent from the collision on another, we can replace Markov's bound with Chernoff's bound [29] to get a more accurate estimation of p_j, and use the Markov's inequality to estimate precision if a HH has traffic from a single switch.

If HH j has traffic on a set of sketches S, SCREAM returns $\sum_{s\in S} c^{est_s}_j$ as an approximation of its volume. In Equation 2.6, however, we prove that this approximation is always smaller than $c^{real}_j + min_i(\sum_{s\in S} e^{cm_s}_{i,j})$ where $e^{cm_s}_{i,j}$ is the Count-Min sketch error on sketch s in row i for HH j.

$$c^{est}_j = \sum_{s\in S} c^{est_s}_j = \sum_{s\in S} min_i(c^{est_s}_{i,j}) \leq min_i(\sum_{s\in S} c^{est_s}_{i,j}) = min_i(\sum_{s\in S} c^{real_s}_{i,j} + e^{cm_s}_{i,j}) = c^{real}_j + min_i(\sum_{s\in S} e^{cm_s}_{i,j}) \tag{2.6}$$

Now for each row of Count-Min sketch, i, errors, $e_{i,j}^{cm_s}$, are independent random variables on different switches. We apply Chernoff's bound on their sum to compute the complement probability of p_j in Equation 2.7 where μ is the summation of average errors on sketches and $(1+\delta)\mu = c_j^{est} - \theta$. The exponent d in Equation 2.7 is because of the minimum over d summations in Equation 2.6.

$$P(c_j^{real} > \theta) \leq 1 - P(min_i(\sum_{s \in S} e_{i,j}^{cm_s}) \geq c_j^{est} - \theta) = 1 - exp(-\frac{\delta_j^2}{2+\delta_j}\mu d) \tag{2.7}$$

Once we calculate p_j, we compute local accuracies by attributing the estimated precision p_j to each switch. If a given HH j has traffic at a single switch, the p_j is only used for the local accuracy of that switch. Otherwise, we attribute precision proportionally to each switch based on its average error as, intuitively, the switch that has smaller average error compared to others must have higher precision.

Hierarchical heavy hitters detection: If a detected HHH has no descendant HHH (*e.g.,* 0000, 010*, 0111 in Figure 2.8), its p_j can be easily calculated using the Markov or Chernoff bound. However, if a detected HHH has descendant HHHs, we cannot just apply those equations to c_j^{est} (volume excluding descendant HHHs) as its p_j depends on the p_j of descendent HHHs, because even if the sketch approximated the volume of a HHH accurately, the over-approximation of the descendant HHHs can make it a false HHH. For example in Figure 2.8, if we detected 0000, 010*, and 0111 as HHHs and over-approximated only the volume of 010* as 17, the weight for 0*** excluding descendant HHHs will be $49-40=9$ and will not be detected. Instead, we detect **** as a HHH with volume $54-40=14$. In this scenario, although we may have approximated the volume of **** correctly, it will be incorrectly detected as a HHH. Thus, we need to find if the sum of over-approximations in a set of descendants could make a true descendant HHH below j and avoid j to become a true HHH.

Instead, SCREAM uses a simpler but conservative approach. First, we notice that in the worst case, the over-approximated traffic has been excluded from one of children of the detected HHH. For each child prefix, we check if these over-approximations could make it a HHH. If any child with a new volume

becomes HHH, the parent cannot be, so as a heuristic, we halve p_j. Second, we find a conservative bound for the over-approximations of *each* descendant HHH and add them up instead of going through the probability distribution of the sum of over-approximations. The over-approximation error bound, say $\hat{e}^{cm}_{D(j)}$, for each descendant HHH of j, $D(j)$, is the upper bound on its error, $e^{cm}_{D(j)}$: $P(e^{cm}_{D(j)} < \hat{e}^{cm}_{D(j)}) > 0.1$. [10] We find this upper bound using Markov's inequality for HHHs originated from a single switch and Chernoff's bound otherwise. For example, Equation 2.8 derived from Equation 2.5 shows the maximum error that a descendant HHH at a single switch at level l can have while keeping $p_j \geq 0.1$.

$$e^{cm}_{D(j)} \leq \frac{T - \sum_{k \in A_l} c^{real}_k}{w \sqrt[d]{0.9}} \tag{2.8}$$

Multiple switches: For the global accuracy, we just replace the Markov inequalities in HH tasks and Equation 2.8 with Chernoff's bound. Finding the local accuracy on each switch is similar to HH with one difference: when the p_j of a HHH decreases because of its descendants, we need to consider from which switch the data for descendants come and assign lower accuracy to them. So in these cases, we also consider the average error of sketches per descendant for each switch and attribute the accuracy proportionally across switches.

Finally, we have found in our experiments (Section 2.8.3) with realistic traffic that, for HHH, recall is correlated with precision. Our intuition is that because the total size of detected HHHs is smaller than T [39] and no non-exact HHH prefix can have a size $\geq 2\theta$ (Section 2.6.1.3), detecting a wrong HHH (low precision) will also be at the cost of missing a true HHH (low recall).

Super source or destination detection: The p_j of a SSD depends on both the distinct counter error (e^{dc}) and hash collisions (e^{cm}) because their errors add up [63]. For a false positive SSD, j, that has counter $c^{est}_{i,j}$ for ith hash function, the error, $e^{dc}_{i,j} + e^{cm}_{i,j}$ must have been greater than $c^{est}_{i,j} - \theta'$ where θ' is computed based on the threshold θ and our version of Equation 2.1: $\theta' = (1 - \frac{1}{w})\theta + \frac{T - \sum_{k \in L} c^{est}_L}{w}$. If the SSD has $d' \leq d$ such counters (remember we choose median instead of minimum), p_j is computed using Equation 2.9.

[10] In practice, we found 0.1 a reasonable value.

$$P(c_j^{real} > \theta') = 1 - \prod_{i=1}^{d'} P(e_{i,j}^{dc} + e_{i,j}^{cm} \geq c_{i,j}^{est} - \theta') \tag{2.9}$$

Now, we show that we can compute p_j, based on the *individual* error distributions of Count-Min sketch and the distinct counter. We can simplify the probability that a counter for SSD j in the ith hash function has large enough error to make it a false positive (the probability expression inside Π). The event happens if sum of Count-Min error and distinct counter error, $e_{i,j}^{dc} + e_{i,j}^{cm}$, is greater than $c_{i,j}^{est} - \theta'$. We can rewrite this probability based on an indicator function (I) and the joint probability distribution of $e_{i,j}^{dc}$ and $e_{i,j}^{cm}$ in Equation 2.10. As the two errors are independent, we can separate the joint distribution in Equation 2.11. Now running the integration on the indicator function and the probability distribution of Count-Min sketch error, $e_{i,j}^{cm}$, we reach Equation 2.12. Then we can calculate each term of Equation 2.12 separately: The error of HyperLogLog sketch has the Gaussian distribution with mean zero and relative standard deviation of $1.04/\sqrt{m}$ when it has m replica counters [51]. The collision error because of Count-Min sketch is also bounded using Markov inequality as before.

$$P(e_{i,j}^{dc} + e_{i,j}^{cm} \geq c_{i,j}^{est} - \theta') = \int_t \int_s I_{t+s \geq c_{i,j}^{est} - \theta'} \, f_{e_{i,j}^{dc}, e_{i,j}^{cm}}(t,s) \, dt \, ds \tag{2.10}$$

$$= \int_t \int_s I_{t+s \geq c_{i,j}^{est} - \theta'} \, f_{e_{i,j}^{dc}}(t) \, f_{e_{i,j}^{cm}}(s) \, dt \, ds \tag{2.11}$$

$$= \int_t f_{e_{i,j}^{dc}}(t) \, P(e_{i,j}^{cm} \geq c_{i,j}^{est} - \theta' - t) \, dt \tag{2.12}$$

In contrast to HH tasks, the recall of SSD is not 1 because the distinct counters can under-approximate. However, the probability of missing a true SSD can be calculated based on the error of the distinct counter [63]. The error of HyperLogLog distinct counter depends on the number of its replica counters, and we can configure it based on the user requirement just at the task instantiation.

Multiple switches: We merged distinct counters on different switches into one distinct counter for each row of Count-Min sketch. Thus, for SSDs, accuracy estimation on multiple switches is the same as one

switch. To compute local accuracies, we use the average error of sketches from different switches to attribute the computed global accuracy, p_j, proportionally across switches.

2.8 Evaluation

We have implemented a complete prototype of DREAM, and use this to evaluate our approach and compare it with alternatives. We then use simulations to explore the performance of DREAM on larger networks, and also study its parameter sensitivity. Sketches are not available in switches yet, so we use simulations driven by realistic traffic traces to show that SCREAM performs significantly better than OpenSketch, and is comparable to an oracle both on a single switch and on multiple switches.

2.8.1 Evaluation Methodology

DREAM Implementation: We have implemented the DREAM resource allocator and the task objects in Java on the Floodlight controller [50]. Our implementation interfaces both with hardware OpenFlow switches, and with Open vSwitch [115]. We have also implemented alternative resource allocation strategies, described below. Our total implementation is nearly 20,000 lines of code.

Simulator: Our event-based simulator runs flow-counters and sketches on switches and reports to the controller every second. Tasks at the controller generate task reports and estimate accuracy, and the resource allocator re-assigns resources among tasks.

Parameter Settings: We use a one second measurement interval and a two second allocation interval. We set the headroom to 5% of the switch capacity and drop tasks if their global accuracy is below the bound for 6 consecutive allocation iterations. The sensitivity to these parameters is explored in Section 2.8.2.3.

Tasks: Our workload consists of the three types of tasks, HH, HHH and CD for DREAM and HH, HHH and SSD for SCREAM both individually and in combination. We choose 80% as the default accuracy bound for all tasks since we have empirically observed that to be the point of diminishing returns for many tasks, but also explore DREAM's performance for other choices of accuracy bounds. We smooth the local and global accuracies using EWMA with history weight of $\alpha = 0.4$. The flow filters for the tasks are

chosen randomly from prefixes with 12 wildcard bits to fit all our tasks. The default threshold for the above tasks is 8Mb and the threshold for SSD tasks is 200 sources per destination IP. For change detection we also use the history weight of $\alpha = 0.8$. Our default drop priority is to drop the most recent task first.

By controlling the mapping of prefixes to switches, we create different scenarios of tasks on switches. For example, a tenant can own a subnet of /12, and its virtual machines in this subnet can be located on different switches. If we assign multiple /10 prefixes to switches (*i.e.,* each switch sees traffic from many tenants), each task will have traffic from one switch. However, if we assign /15 prefixes to switches (*i.e.,* one tenant sends traffic from many switches), each task monitors traffic from 8 switches at most. Tasks observe dynamically varying traffic as each task picks a /4 prefix of a 5-min chunk of trace and maps it to their /12 filter. Thus, our workload requires dynamic resource adjustment because of traffic properties variations and task arrival/departure. In a data center, our systems would monitor traffic on the source switches of traffic (the ToRs), thus network topology is irrelevant to our evaluation.

Tasks run for an average of 5 minutes. For evaluations on our prototype, 256 tasks having traffic from 8 switches arrive based on a Poisson process during 20 minutes. For the large-scale simulation, 4096 tasks having traffic from 8 out of 32 switches arrive during 80 minutes. We note that these are fairly adversarial settings for task dynamics, and are designed to stress test DREAM, SCREAM and other alternatives.

Finally, we use a 5-hour CAIDA packet trace [20] from a 10Gbps link with an average 2Gbps load. We divide it into 5-min chunks, each of which contains 16 /4 prefixes, of which only prefixes with $>1\%$ total traffic are used. Each task randomly picks a /4 prefix which is mapped to its /12 filter

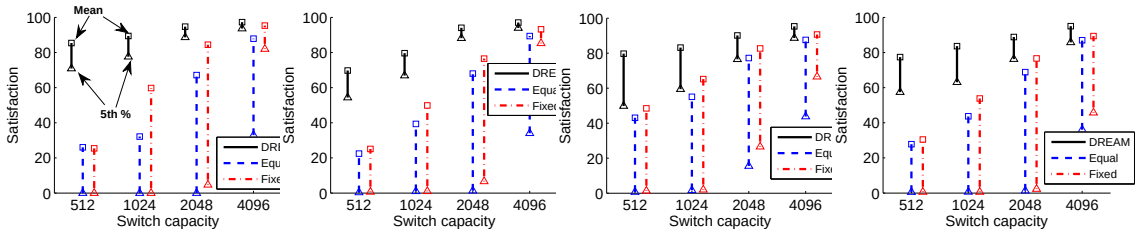
Evaluation metrics: We evaluate DREAM, SCREAM and other alternatives using three metrics. The *satisfaction* of a task is the percentage of time a task has an accuracy above the bound when the task was active. In results from our prototype, we use *estimated* accuracy because delays in installing TCAM counters in the actual experiment make it difficult for us to assess the ground-truth in the traffic seen by a switch. We have found in our evaluations that the estimated accuracy consistently under-estimates the real accuracy by 5-10% on average, so our prototype results are a conservative estimate of the actual satisfaction that tasks would see in practice. In our simulation results, we use the real accuracy.

We show both the average and 5^{th} percentile for this metric over all tasks. The latter metric captures the tail behavior of resource allocation: a 5-th percentile of 20 means that 95% of tasks had an accuracy above the bound for 20% of their lifetime. The *drop* and *rejection* ratios measure the percentage of tasks that are dropped and rejected, respectively. While the rejection ratios can be a function of the workload and can be high in highly overloaded conditions, we expect drop ratios to be small for a viable scheme (*i.e.,* it is desirable that a task, once admitted, is not dropped, but may be rejected before admission).

2.8.2 DREAM Evaluation

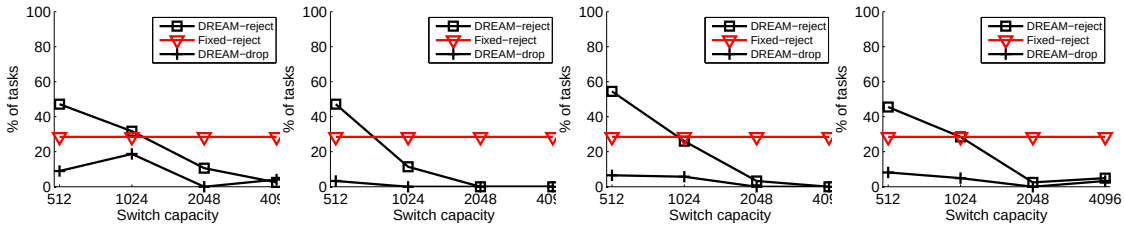
Alternative strategies: One alternative we explore is to reserve a *Fixed* fraction of counters on each switch for a task, and reject tasks for which this fixed allocation cannot be made. While we evaluated fixed allocation with different fractions, here we only show the results for the scenario that allocates $\frac{1}{32}$ of the resources on a switch per task. Larger allocations result in higher satisfaction for fewer tasks and a higher rejection ratio, and smaller fixed allocations accept more tasks at the expense of lower satisfaction (See Section 2.8.2.3 for the result of Fixed allocation with different fractions). A more complex algorithm is to give *Equal* amounts of resources to each task. When a task joins, it gets an equal share of counters as other tasks on the switches it has traffic from. The allocations are also updated when a task leaves, and *Equal* does not reject tasks.

Experimental setup: We replay the CAIDA traffic on 8 switches. We attempted to evaluate DREAM on a modern hardware switch (the Pica8 3290 [120]) but its delay for rule installation is unacceptably high: 256 rules take 1 second, and 512 rules take 10 seconds. We believe better engineering will result in improved installation times in the future; indeed, for applications with tight control loops like ours, it is essential to improve installation times in hardware switches. Our evaluations are conducted on software switches [115] that can delete and save 512 rules in less than 20ms. We also reduce control loop delay by using *incremental update* of TCAM counters and associated rules, updating at each epoch only the rules that have changed from the previous epoch. We show below that this strategy results in acceptable rule installation performance (Section 2.8.2.4). In our experiments, the DREAM prototype runs on a Floodlight



(a) *HH* **(b)** *HHH* **(c)** *CD* **(d)** *Combined workload*

Figure 2.15: *Satisfaction in prototype*



(a) *HH* **(b)** *HHH* **(c)** *CD* **(d)** *Combined workload*

Figure 2.16: *Rejection and drop in prototype*

controller [50] on a quad core 2.4 Ghz Xeon processor connected to the switches through a 1Gbps shared link with ping delay of 0.25ms.

2.8.2.1 Results from Prototype

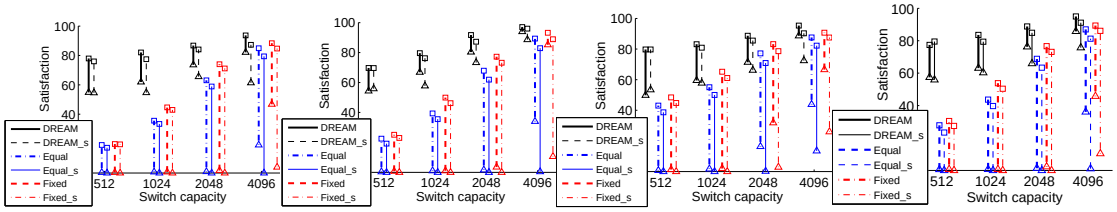
Figure 2.15 shows, for different switch capacities, the 5^{th} percentile and mean satisfaction of tasks for HHH, HH, and CD separately, as well as a combined workload that runs a mixture of these tasks. *The mean value is the upper end of each vertical bar, and the 5^{th} percentile is the lower end.* These figures demonstrate DREAM's superior performance compared to the alternatives, both in terms of the mean and the 5^{th} percentile.

Large capacity switches. For large switches, DREAM can keep almost all tasks satisfied by temporally and spatially multiplexing TCAM resources without rejecting or dropping any task (Figure 2.16). For example, Figure 2.15b shows that 95% of tasks were satisfied for more than 94% of their lifetime. By contrast, a high mean and a dramatically lower 5^{th} percentile (about 40%, or nearly $2\times$ less than DREAM) for Equal indicate that this scheme has undesirable tail behavior: it keeps tasks that require fewer resources satisfied, but leaves large tasks unsatisfied. This is undesirable in general: larger tasks are where the action happens, in a manner of speaking, and cannot be left dissatisfied. The Fixed approach achieves high

average satisfaction, but has two drawbacks: poor tail performance, and a high rejection ratio of about 30%.

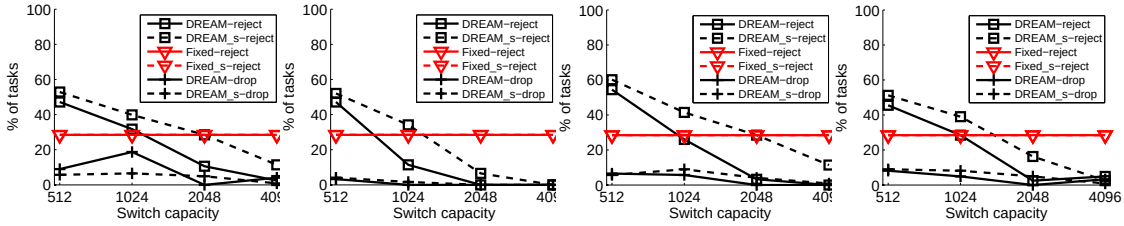
Highly resource-constrained switches. For smaller switches, where our workload overloads the resources on switches, DREAM leverages rejection to limit the load and keep active tasks satisfied by multiplexing resources. For example, in Figure 2.15a for a switch with 512 counters, DREAM rejects about 50% of tasks, but can keep 95% of tasks satisfied for more than 70% of their lifetime. By contrast, in this setting, Equal performs pathologically worse: its average satisfaction is 20% and 5% of tasks under Equal get nearly zero satisfaction. This is because Equal does not perform admission control and it under-provisions resources in small switches and thus gets low satisfaction. We emphasize that this is an adversarial workload and represents a high degree of overload: DREAM has to reject nearly 50% of the tasks, and drop about 10% in order to satisfy the remaining tasks. Also, DREAM's mean and 5^{th} percentile satisfaction is a little lower than for the larger switch capacity case, mostly because the adaptation required to fit tasks into the smaller switch requires more allocation epochs to converge.

Across different task types, the results are qualitatively consistent, save for two exceptions. First, the drop ratio for HH detection (Figure 2.16a) increases from switch capacity of 512 to 1024, which is likely because of a decrease in the rejection ratio. Moreover, its drop rate is higher than other tasks, which we believe is because we under-estimate the accuracy of tasks. Remember that to calculate the number of missed HHs under a prefix we used the bound of $\frac{volume}{\theta}$. However, the missed HHs could have larger volumes than θ and make this upper bound loose. This loose upper bound ensures better 5^{th} percentile performance than other schemes we have tried, so it seems a better design choice: it drops more tasks to favor satisfying more. Second, the average satisfaction of Equal and Fixed is higher than other task types for change detection (but the tail performance is poor). This is because, in our dataset, not all epochs have a significant change, thus the tasks are satisfied in those epochs even with very small resources.



(a) *HH* **(b)** *HHH* **(c)** *CD* **(d)** *Combined workload*

Figure 2.17: *Satisfaction for simulator validation using prototype*



(a) *HH* **(b)** *HHH* **(c)** *CD* **(d)** *Combined workload*

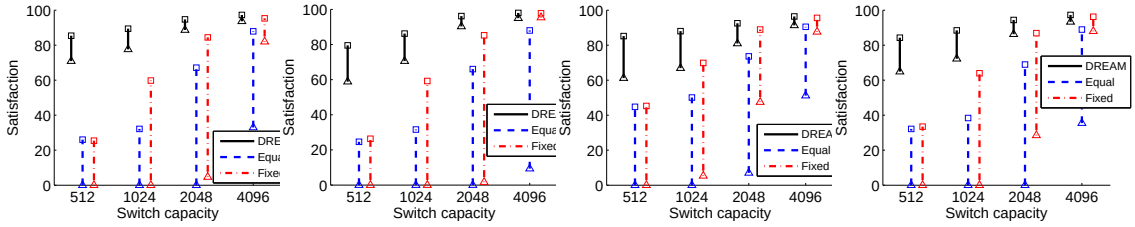
Figure 2.18: *Rejection and drop for simulator validation using prototype*

2.8.2.2 Results from Simulation

Simulator validation. We use simulation to study the performance of DREAM and other alternatives at larger scale. Our simulator uses the same code base as the prototype.

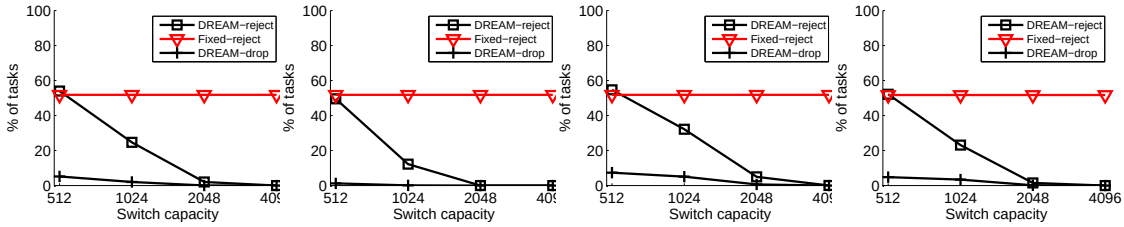
Figures 2.17 and 2.18 compare the performance of DREAM on the simulator and our prototype, under identical configurations. The mean satisfaction matches closely, while 5^{th} percentile is smaller for large switches in simulation. We believe this difference is because the simulator does not model the control loop delay (the time from retrieving counters from the switch to the time new rules are installed), while tasks may miss some traffic during this period in the prototype. For a similar reason, the rejection ratio in the prototype is lower, since the prototype may miss a few events, leaving enough headroom for DREAM.

Results at scale. DREAM's superior performance is also evident in larger networks. Figures 2.19 and 2.20 compare the satisfaction and rejection ratio for the 32 switch network with 4096 tasks. In this much larger setting, the superior tail satisfaction of DREAM at low load (high capacity) and the superior average satisfaction at high load (low capacity) are strikingly evident. As with smaller networks, DREAM has a small drop ratio (less than 5%) at high load.



(a) *HH* **(b)** *HHH* **(c)** *CD* **(d)** *Combined workload*

Figure 2.19: *Satisfaction in large scale simulation*



(a) *HH* **(b)** *HHH* **(c)** *CD* **(d)** *Combined workload*

Figure 2.20: *Rejection and drop in large scale simulation*

2.8.2.3 Parameter Sensitivity Analysis

To understand how sensitive our results are to changes in various parameters, we conduct several experiments with a switch capacity of 1024 TCAM entries, but vary several other parameters. We note that for our baseline workload, 1024 TCAM entries represent a constrained setting. For this set of results, we show results for a specific type of task (HHH), rather than using results from a combined workload, as this makes it easier to interpret the results (Figures 2.21 and 2.22). The qualitative behavior of other tasks is similar.

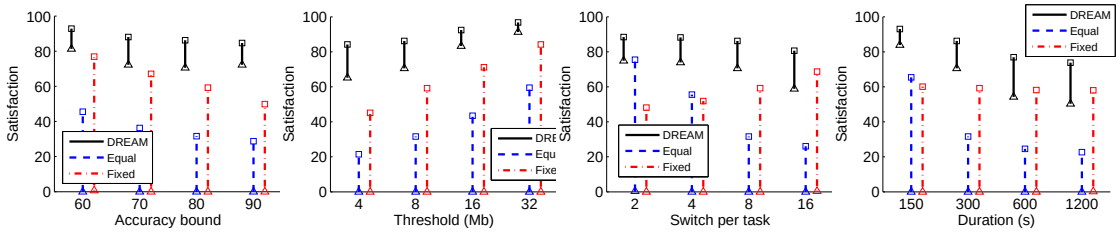
DREAM keeps tasks satisfied for different accuracy bounds. With higher accuracy bounds the allocation becomes harder, since tasks in general need more resources, but DREAM can keep more tasks satisfied with a smaller rejection rate compared to Fixed allocation (Figures 2.21a and 2.22a). Besides, DREAM is also uniformly better than Equal allocation because it effectively multiplexes resources across tasks.

DREAM can handle different task thresholds. Finding smaller HHHs (*i.e.,* HHH tasks with a lower threshold) requires more resources and thus increases the load on the system. DREAM limits this load by

rejecting more tasks to keep the admitted tasks satisfied, in contrast to Equal allocation. On the other hand, for larger threshold values, it satisfies more tasks with higher satisfaction compared to the Fixed scheme (Figures 2.21b and 2.22b).

DREAM multiplexes resources spatially. Task filters can match the traffic from many switches; however, a task's accuracy may depend on receiving resources from only a few switches because of spatial diversity. If we increase the number of switches that see traffic for a task, each per-switch allocator observes more tasks and Equal allocation allocates smaller share of resources to tasks, thus its performance drops (Figure 2.21c). On the other hand, Fixed allocation reserves resources on many switches and thus its rejection rate increases fast with the number of switches per task (Figure 2.22c). Instead, DREAM identifies the switches that the task needs more resources on, and can satisfy tasks while rejecting fewer tasks than Fixed.

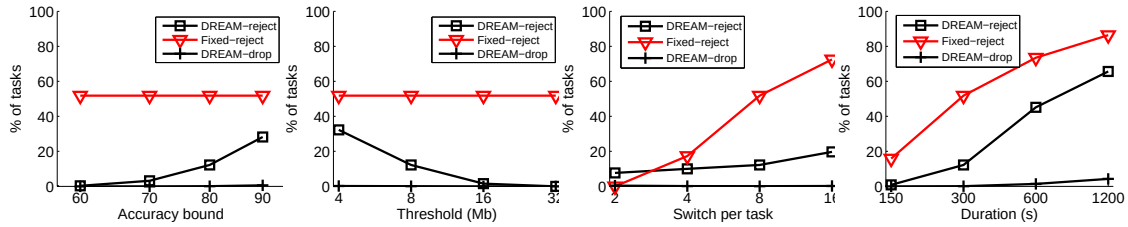
DREAM handles diverse workloads. Increasing job duration and increasing arrival rate also increases the load on the network. Figures 2.21d and 2.22d show the satisfaction, drop and rejection of dream for different job duration. Compared to increasing the threshold, increasing the duration of tasks increases the rejection rate of the Fixed allocation, because if a task is admitted it holds on to the resources for a longer duration. DREAM has lower rejection rate and provides higher satisfaction. Changing the arrival rate also results in similar graphs (Figure 2.23).



(a) *Accuracy bound* **(b)** *Threshold* **(c)** *Switch per task* **(d)** *Duration*

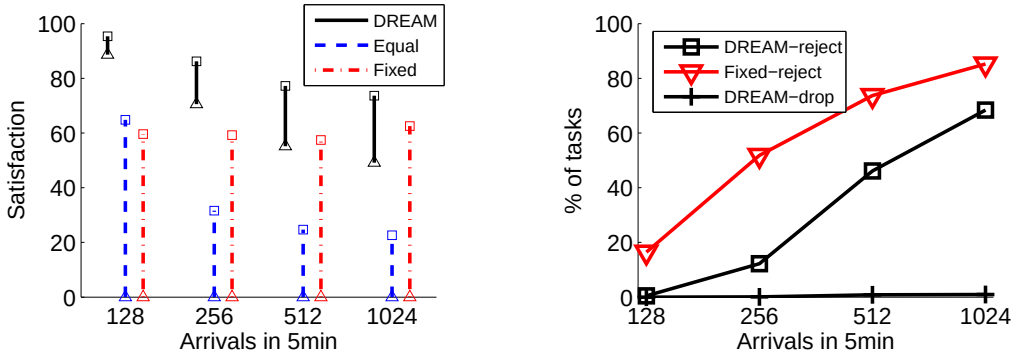
Figure 2.21: *Satisfaction for parameter sensitivity analysis*

Headroom is important to keep drop rate low. If DREAM does not reject tasks a priori, many tasks will starve just after joining the system. For example, Figure 2.24b shows a drop rate of 30% for DREAM when there is no headroom at an allocation interval of 2s. Interestingly, the level of headroom does not



(a) *Accuracy* **(b)** *Threshold* **(c)** *Switch per task* **(d)** *Duration*

Figure 2.22: *Rejection and drop for parameter sensitivity analysis*



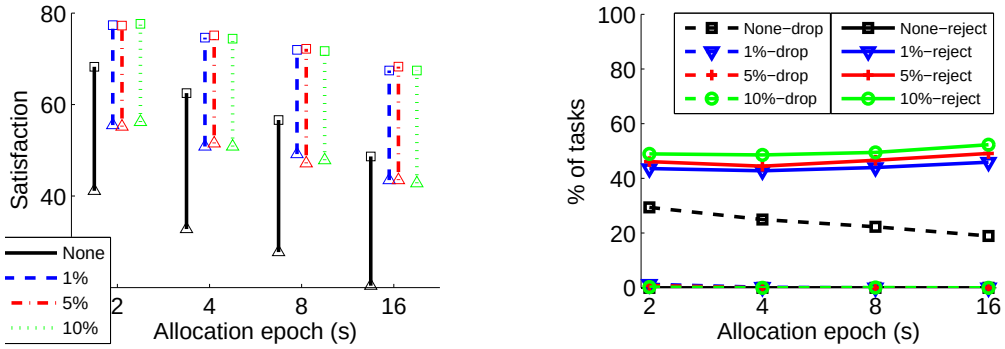
(a) *Satisfaction* **(b)** *Rejection and drop*

Figure 2.23: *Arrival rate parameter sensitivity analysis*

seem to make a significant difference in the statistics of satisfaction, but can affect drop rates. With a 5% and 10% headroom, drop rates are negligible.

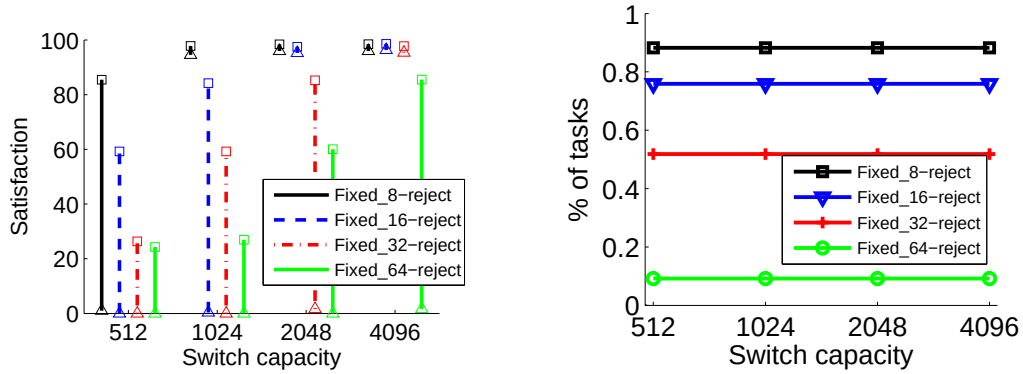
Other DREAM parameters include allocation interval, drop threshold, and the MM algorithm multiplicative factor. Figure 2.24a shows that allocating resources infrequently with a larger allocation interval results in lower satisfaction because DREAM cannot adapt resources quickly enough. Smaller drop threshold increases the drop rate and satisfaction, and increasing the multiplicative factor of the MM algorithm causes higher rejection rate because poor tasks overshoot their goal by large change step sizes and thereby reduce headroom. Note that a smaller multiplication factor requires a larger drop threshold to avoid unnecessary drops in under-loaded cases.

Fixed allocation parameter sensitivity analysis. Figure 2.25 compares the performance metrics for the fixed allocation with different allocation fractions from $\frac{1}{32}$ that we used. Larger allocations result in higher satisfaction for fewer tasks and a higher rejection ratio. For example, the satisfaction of Fixed_8 approach that allocates $\frac{1}{8}$ of switch capacity to each task is always close to 100% even for 5^{th} percentile for many



(a) *Satisfaction* **(b)** *Rejection and drop*

Figure 2.24: *Headroom and allocation epoch (combined workload)*



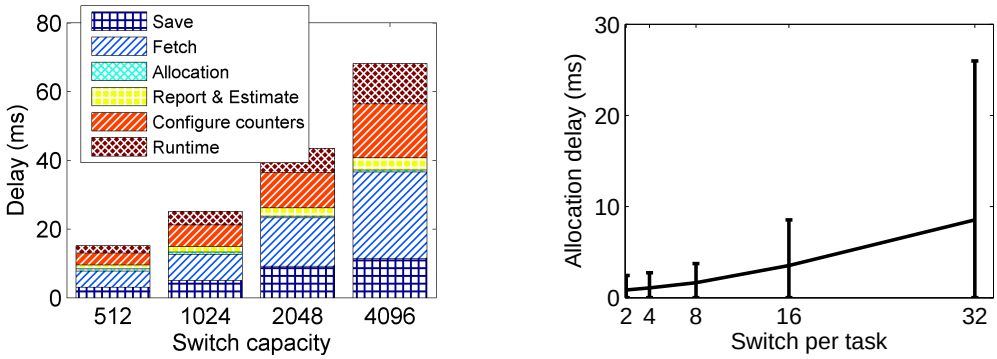
(a) *Satisfaction* **(b)** *Rejection and drop*

Figure 2.25: *Fixed allocation configurations*

switch configurations. However, it rejects about 90% of tasks. Smaller fixed allocations accept more tasks at the expense of lower satisfaction. For example, Fixed_64 that reserves $\frac{1}{64}$ of switch capacity for each task, rejects only 10% of tasks but its average satisfaction rate is lower than other configurations and its 5^{th} percentile is always close to zero.

2.8.2.4 Control Loop Delay

The delay of the control loop – the process of configuring counters and updating rules in TCAMs – can affect the accuracy of real prototypes because important events can be missed while these counters are being updated. We calculate the control loop delay by calculating the average delay between fetching the counters from the switches to receiving the OpenFlow barrier reply from all switches after installing incremental rules on the prototype.



(a) *Prototype delay* **(b)** *Allocation delay*

Figure 2.26: *Control loop delay (combined workload)*

Figure 2.26a breaks down the delay of control loop into: saving the incremental rules, fetching the counters, allocating resources, creating the report and estimating its accuracy, configuring counters through divide and merge algorithm and the runtime overhead for combining counter statistics from multiple switches and creating a counter on multiple switches for all tasks. The interesting points are: (1) the allocation delay (the overhead of computing new allocations) is negligible compared to other delays; (2) the average (95^{th}%) allocation delay decreases with increasing switch size from 0.65 (3.1) ms to 0.5 (1.3) ms, because for larger switches, fewer tasks are dissatisfied although more tasks have been admitted; (3) fetch times dominate save times (although it takes longer to save or delete a counter than to fetch one) because we fetch all counters, but only delete and save incrementally. For example, for the case of switches with 1024 TCAM capacity, on average in each epoch 90% of the counters did not change. This number increases for larger switch capacities as tasks need to configure their counters less frequently because (a) they already have an accurate view of network (b) their allocations changes rarely as more tasks are satisfied with more resources.

Finally, the DREAM controller scales to many tasks because it is highly parallelizable; each task can run on a core and each per-switch allocator can run separately. The per-switch resource allocator does more work as we increase the number of switches per task, since each switch sees more tasks. Figure 2.26b shows that the mean, and 95^{th} percentile of allocation delay in the large scale simulation environment (on

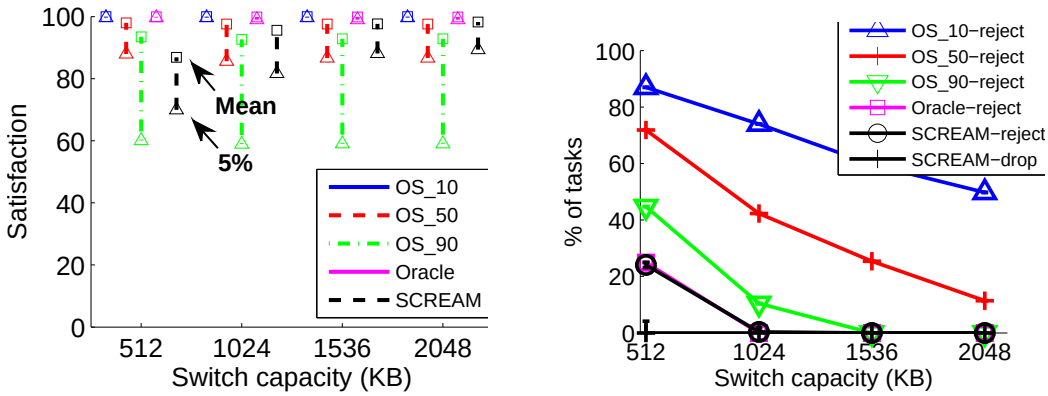
a 32 core machine) increases for larger number of switches per task, but the mean is still less than 10ms and the control loop delay is still dominated by other (unavoidable) latencies in the system.

2.8.3 SCREAM Evaluation

Alternative strategies: Sketches often provide a provable trade-off between resources and accuracy. It means that given the worst-case traffic properties, we can allocate enough resources per task to guarantee its accuracy. Therefore, to evaluate SCREAM, we use an alternative strategy introduced in OpenSketch [150] that is more advanced than the fixed algorithm. OpenSketch can allocate SRAM to each task instance considering its type and parameters at its instantiation. Also, because sketches can find the right set of flow properties in the data plane *without* any iterative reconfiguration from the controller, it is feasible to implement an oracle solution that knows the amount of required resources for each task on each switch for every epoch by running the experiment many times. We describe details of the two alternatives in the following paragraphs.

OpenSketch: OpenSketch [150] allocates resources to a task based on worst-case traffic to reach a given relative error at a single switch. To bound the error to $x\%$ of the threshold θ on a HH/HHH detection task that has an average traffic of T, it configures a Count-Min sketch with $w = \frac{eT}{x\theta}$. For example, if a task has 128 Mbps traffic, a sketch with $w = 435$ and $d = 3$ can guarantee that the relative error is lower than 10% of the threshold $\theta = 1$ MB with probability $1 - e^{-3} = 0.95$. In our experiments, we fix the number of rows, d, to 3 and find w based on the error rate. For SSD detection, OpenSketch solves a linear optimization to find the best value for distinct counter parameter (m) and Count-Min sketch width (w) that minimizes the size of sketch [150].

At task arrival, OpenSketch finds the required amount of resources for the relative error guarantee and reserves its resources if there are enough free resources; otherwise, it rejects the task. We run OpenSketch with a range of relative errors to explore the trade-off between satisfaction and rejection. OpenSketch was originally proposed for the single-switch case, so for comparison on multiple switches, we propose an extension as follows. We run a separate OpenSketch allocator for each switch and if a task cannot get



(a) *Satisfaction* **(b)** *Rejection and drop*

Figure 2.27: *Comparison for OpenSketch (different relative error%) and the oracle at a single switch*

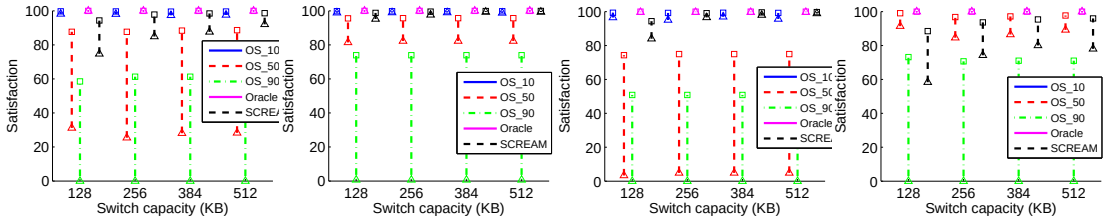
resources on any switch it will be rejected. However, setting the resources based on the worst case traffic for all switches would require too many resources as some tasks may have most of their traffic from a single switch. Therefore, given the total traffic T for a task, we configure the sketch at each switch based on the average traffic across switches ($T/8$).

Oracle: We also evaluate an *oracle* that knows, at each instant, the exact resources required for each task in each switch. In contrast, SCREAM does not know the required resources, the traffic properties or even the error of accuracy estimates. We derive the oracle by actually executing the task, and determining the resources required to exceed the target accuracy empirically. Thus, the oracle always achieves 100% satisfaction and never drops a task. It may, however, reject tasks that might be dropped by SCREAM, since the latter does not have knowledge of the future.

2.8.3.1 Performance at a Single Switch

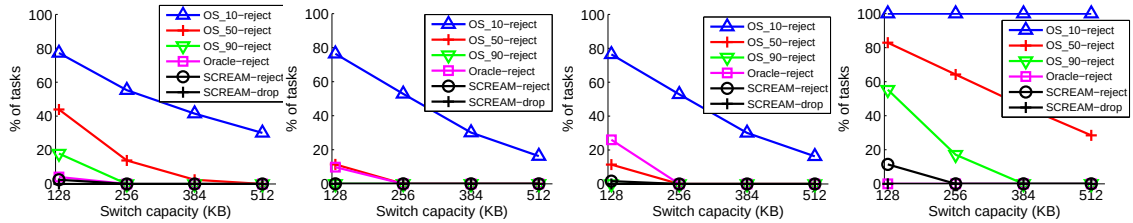
SCREAM supports more accurate tasks than OpenSketch. Figure 2.27 compares SCREAM to OpenSketch with three different relative error percentages for a combination of different types of tasks over different switch memory sizes. [11] We ran the experiment 5 times with different task arrival patterns and show the error bars in Figure 2.27b which are very tight. Note that OpenSketch uses the worst-case guarantee and even a relative error of 90% of threshold can result in high satisfaction. SCREAM has higher

[11] Switch memory size and network bandwidth overhead are linearly related and both depend on the number of counters in each sketch.



(a) *Combine* **(b)** *HH* **(c)** *HHH* **(d)** *SSD*

Figure 2.28: *Satisfaction comparison for OpenSketch (different relative error%) and the oracle on multiple switches*



(a) *Combine* **(b)** *HH* **(c)** *HHH* **(d)** *SSD*

Figure 2.29: *Drop & reject comparison for OpenSketch (different relative error%) and the oracle on multiple switches*

satisfaction rate than OpenSketch with high relative error (90%), but its rejection rate is lower. SCREAM can support 2 times more tasks with comparable satisfaction (*e.g.,* the curve 50% error on switch capacity of 1024 KB). Finally, OpenSketch needs to reject up to 80% of tasks in order to get much higher satisfaction than SCREAM (in 10% relative error curve).

SCREAM can match the oracle's satisfaction for switches with larger memory. For switches with larger memory, SCREAM can successfully find the resource required for each task and reach comparable satisfaction as the oracle while rejecting no tasks (Figure 2.27). For switches with smaller memory, SCREAM has similar rejection rate as the oracle, but its satisfaction rate is smaller than the oracle. The reason is that, in this case, SCREAM needs to dynamically adjust task resources over time more frequently than for larger switches and waits until tasks are dropped, during which times some tasks may not be satisfied.

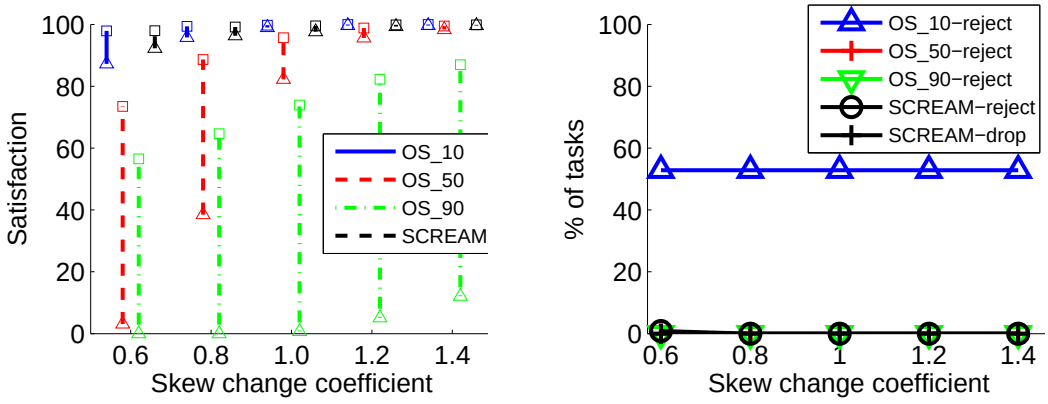
2.8.3.2 Performance on Multiple Switches

Figures 2.28a and 2.29a show that SCREAM can keep all task types satisfied on 8 switches. However, OpenSketch either has high rejection for strict error guarantee that over-allocates (OS_10), or cannot keep

tasks accurate for relaxed error guarantees that admit more tasks (OS_50, OS_90). Note that the satisfaction of OpenSketch for multiple switches is lower than its satisfaction on a single switch especially for $5^{th}\%$ because OpenSketch uses the error bounds that treat every switch the same. (OpenSketch is not traffic aware, and cannot size resources at different switches to match the traffic). As a result, it either sets low allocation for a task on all switches and reaches low satisfaction or sets high allocation and wastes resources. However, SCREAM can find the resource requirement of a task on each switch and allocate just enough resources to reach the required accuracy. Other diagrams in Figures 2.28 and 2.29 also show the superiority of SCREAM over OpenSketch for each individual task type. Note that SSD tasks need more resources and OpenSketch with 10% relative error cannot run any task on the range of switch capacity we tested.

Like the single switch case, SCREAM can also achieve a satisfaction comparable to the oracle for multiple switches (Figure 2.28). In Figure 2.29, SCREAM has a lower rejection rate than oracle for small switches but still has no drop. This is because SCREAM does not know the future resource requirements of tasks, thus it admits them if it can take enough headroom of free resources from highly accurate tasks. But if later it cannot support them for *a few epochs*, it drops them. Thus, SCREAM can tolerate tasks to be less accurate for a few epoch and does not reject or drop them. However, the oracle is strict and rejects these tasks at task instantiation; hence, it has higher rejection.

SCREAM supports more accurate tasks than OpenSketch over different traffic traces. Above, we showed SCREAM's superior performance for tasks with different traffic traces. Now, we explore the effect of traffic skew by changing the volume of traffic from each source IP at each second: Recall that the frequency (denoted by volume) of elements (source IPs) of rank i in ZipF distribution with exponent z is defined by i^{-z}. Thus to change the ZipF exponent to $s \times z$, it is enough to raise to the power of s the traffic volume from each source IP in each measurement epoch. Note that we keep the total traffic volume the same by normalizing the traffic volume per source IP by the ratio of new total volume over old total volume. Figure 2.30 shows that SCREAM can keep tasks satisfied in a wide range of skew. For example, if we reduce the skew to 60%, the mean ($5^{th}\%$) of satisfaction is 98% (92%) and no task is rejected or



(a) *Satisfaction* **(b)** *Rejection and drop*

Figure 2.30: *Changing skew for HH detection at multiple switches with capacity 64 KB*

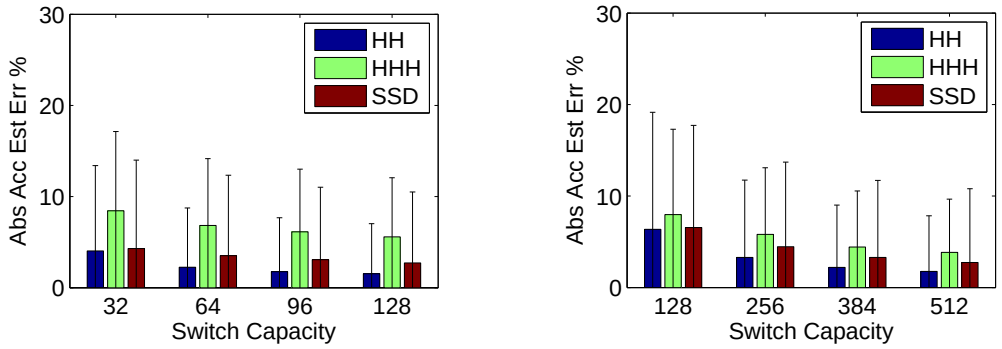
dropped. However, as OpenSketch considers the worst case irrespective of traffic properties, it either ends up with low satisfaction for less skewed traffic (OS_50, OS_90) or over-provisioning and rejecting many tasks (OS_10).

2.8.3.3 Accuracy Estimation

SCREAM's superior performance requires low accuracy estimation error. Our experiments show that our accuracy estimation has within 5% error on average. Although we define accuracy based on precision, SCREAM achieves high recall in most cases.

SCREAM accuracy estimation has low errors. We calculated the accuracy estimation error (the percentage difference of the estimated accuracy and the real accuracy) of tasks for the single switch and multiple switches cases. Figure 2.31 shows that SCREAM can estimate the accuracy of tasks with about 5% error on average. As an aside, using the strawman accuracy estimator for HH detection (Section 2.7.2), resulted in about 15%(40%) error in average (std) and forced SCREAM to reject or drop all tasks in this scenario.

The error of our accuracy estimator varies across different task types but goes down for switches with larger capacity. The reason is that the error of our accuracy estimators decreases for higher accuracies and with larger switches more and more tasks can reach higher accuracies. We found that the cases with high error in accuracy estimation usually only have very few detected items that are close to the threshold. For



(a) *Single switch* **(b)** *Multiple switches*

Figure 2.31: *Accuracy estimation error*

such items with small margin over the threshold, Markov inequality is loose, resulting in error in accuracy estimation.

Tasks in SCREAM have high recall. Figure 2.32 shows that the satisfaction of HHH detection tasks based on recall is higher than that of OpenSketch. This is because the recall of HHH detection is correlated with its precision (see Section 2.7.2). Hence, the curves for satisfaction based on recall (Figure 2.32) are similar to the curves for satisfaction based on precision (Figure 2.28c).

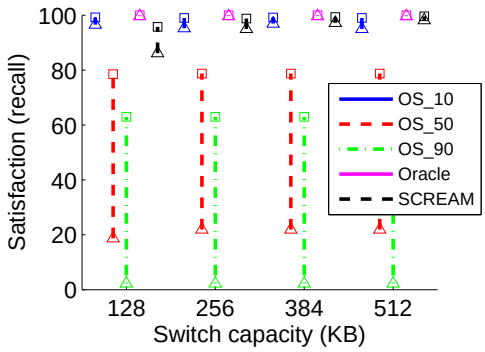


Figure 2.32: *HHH satisfaction based on recall on multiple switches*

As mentioned in Section 2.7.2, the recall of HH detection is always 1, and the recall for SSD detection depends on the number of replica counters in the distinct counter. In our experiments, the average recall was above 80% for all switch sizes.

2.9 Conclusions

Measurement is fundamental for network management systems. Flow-based counters and sketches allow measurement tasks to find flows and flow aggregates with certain properties inside the network. DREAM and SCREAM enable operators and cloud tenants to flexibly specify their measurement tasks in a network, and dynamically allocates SRAM and TCAM resources to these tasks based on the resource-accuracy trade-offs for each task and ensure a user-specified minimum accuracy. We proposed algorithms for estimating the instantaneous accuracy of tasks to dynamically adapt to the required resources for each task on multiple switches. By multiplexing resources among network-wide measurement tasks, SCREAM and DREAM support more accurate tasks than current practice, Fixed allocation and OpenSketch.

Chapter 3

Trumpet: Timely and Precise Triggers in Data Centers

As data centers grow larger and strive to provide tight performance and availability SLAs, their monitoring infrastructure must move from passive systems that provide aggregated inputs to human operators, to active systems that enable programmed control. In this Chapter, we propose Trumpet, an event monitoring system that leverages CPU resources and end-host programmability, to monitor every packet and report events at millisecond time-scales. Trumpet users can express many *network-wide events*, and the system efficiently detects these events using *triggers* at end-hosts. Using careful design, Trumpet can evaluate triggers by inspecting every packet at full line rate even on future generations of NICs, scale to thousands of triggers per end-host while bounding packet processing delay to a few microseconds, and report events to a controller within 10 milliseconds, even in the presence of attacks. We demonstrate these properties using an implementation of Trumpet, and also show that it allows operators to describe new network events such as detecting correlated bursts and loss, identifying the root cause of transient congestion, and detecting short-term anomalies at the scale of a data center tenant.

3.1 Introduction

Data center network management tasks range from fault diagnosis to traffic engineering, network planning, performance diagnosis, and attack prevention and require network monitoring. Commercial network monitoring tools (*e.g.,* SNMP, NetFlow) produce highly aggregated or sampled statistics (*e.g.,* per port count

in SNMP, sampled NetFlow records) at relatively coarse time-scales (seconds to minutes), often provide input to dashboards designed to present aggregate network views to human operators, and require humans to interpret their output and initiate network control.

As data centers evolve to larger scales, higher speed, and higher link utilization, monitoring systems must detect *events* (such as transient congestion or server load imbalance) more precisely (inspecting every packet), at fine time-scales (on the order of milliseconds) and in a programmable fashion (so that the set of events can be dynamically determined). In this Chapter, we argue for a shift of monitoring infrastructure from *passive* to *active*: Instead of presenting aggregated network views to human operators, monitoring infrastructure should both allow for interactive drill down based on current conditions and for automated reaction to pre-defined, fine-grained packet events at the scope of entire fabrics rather than individual links or servers.

The challenge in achieving this goal is scale along many dimensions: the number of endpoints, the high aggregate traffic in the network, and, as data center networks move to support large numbers of tenants and services, the number of events of interest. To address this challenge, we leverage the relatively plentiful processing power at end-hosts in a data center. These end-hosts are programmable, already need to process every packet once, and can monitor all packets without sampling using their powerful CPUs.

Contributions: In this Chapter, we present the architecture and implementation of Trumpet: an event monitoring system in which users define network-wide events, a centralized controller installs *triggers* at end-hosts where triggers test for local conditions, and the controller aggregates these signals and tests for the presence of specified network-wide events. This architecture demonstrates the benefits of pushing, to end-hosts, the substantial scalability challenges faced by datacenter-scale per-packet monitoring. The key to Trumpet's efficacy is carefully optimized per-packet processing inline on packets being demultiplexed by a software switch.

We demonstrate the benefits of Trumpet with the following contributions. First, Trumpet introduces a *simple network-wide event definition language* which allows users to define network-wide events by specifying a *filter* to identify the set of packets over which a *predicate* is evaluated; an *event* occurs when

the predicate evaluates to true. Users can set a predicate's time and flow granularity. Despite this simplicity, the language permits users to capture many interesting events: identifying if flows trigger congestion on other flows, when aggregate traffic to hosts of a service exceeds a threshold, or when connections experience a burst of packet loss.

Our second contribution is scaling event processing at end-hosts while still being able to detect events at time-scales of a few milliseconds. Upon receiving an event definition, the central controller installs *triggers* corresponding to events at end-hosts. A trigger determines whether the user-specified event has occurred at an end-host; if so, the end-host sends the trigger's output to the controller. The controller collects the trigger results from individual end-hosts to determine if the event definition has been satisfied across the network. The key challenge is to support thousands of triggers at full line rate without requiring extra CPU cores on a software switch, and without (a) dropping any packet, (b) missing any events, and (c) delaying any packet by more than a few μs.

To achieve these properties, Trumpet processes packets in two phases in the end-hosts: A match-and-scatter phase that matches each incoming packet and keeps per 5-tuple flow statistics; and a gather-test-and-report phase which runs at each trigger's time granularity, gathers per-trigger statistics, and reports when triggers are satisfied. We also design and implement a suite of algorithms and systems optimizations including caching to reduce computation, pre-fetching and data structure layout to increase memory access efficiency, careful use of virtual memory, and a queue-length adaptive scheduling technique to steal cycles from packet processing. Our design degrades gracefully under attack, and scales naturally to higher-speed NICs likely to be deployed in datacenters in the next few years.

Our third contribution is an *evaluation of Trumpet* across a variety of workloads and event descriptions. We evaluated Trumpet running on a single core with 4K triggers and (a) 64-byte packets on a 10G NIC at line rate and (b) 650-byte packets on a 40G NIC at line rate. Trumpet sustains this workload without packet loss, or missing a single event. More generally, we characterize the feasible set of parameters, or the *feasibility region*, in which Trumpet finishes every sweep, and never loses a packet. Each of our optimizations provides small benefits, but because at full line-rate we have almost no processing headroom,

every optimization is crucial. Trumpet can report network-wide events within 1ms after the first trigger is matched at one end-host. Its matching complexity is independent of the number of triggers, a key reason we are able to scale the system to large numbers of triggers. Finally, Trumpet degrades gracefully under attack: with the appropriate choice of parameters (which can be determined by an accurate model), it can sustain a workload in which 96% of the packets are DoS packets, at the expense of not being able to monitor flows of size smaller than 128 bytes.

Fourth, we demonstrate the expressivity of Trumpet with three use cases: a) pacing traffic, using a tight control loop, when that traffic causes a burst of losses; b) automatically identifying flows responsible for transient congestion and c) detecting, using a network-wide event, services whose combined volume exceeds a threshold.

Getting precise and accurate visibility into data centers is a crucial problem, and Trumpet addresses an important part of this space. In practice, a spectrum of solutions is likely to be necessary, including approaches (Section 5.3) that use NICs and switches. NIC offloading saves end-host CPU and can monitor the flows that bypass the hypervisor (*e.g.,* using SR-IOV and RDMA). Other schemes that mirror packets or packet headers to a controller, where an operator or a script can examine headers or content, can help drill down on events triggered by Trumpet. As an aside, these schemes might need to process packets at high rates at the controller, and Trumpet's processing optimizations might be applicable to these.

3.2 The Case for Trumpet

Today's monitoring infrastructure is designed for human monitoring, so coarse time-scales and high levels of aggregation are sufficient. Modern data centers need real-time, fine-grained, and precise monitoring to feed a variety of control systems. In this section, we elaborate on this observation, which motivates the design of Trumpet.

Problems of today's monitoring systems: Network management systems in data centers rely on detecting network *events*. An event often indicates a network condition that may require a corrective action,

mostly through a computer-assisted reaction. Events are often defined in terms of packets dropped, delayed, or delivered by the network, and can range in topological scope from a specific flow, to traffic aggregates (*e.g.,* all network traffic to a service), and in temporal scope from being extremely short-lived to long-lived events.

Traditional monitoring systems only provide coarse-grained views of the network at larger time-scales, which are sufficient for humans to understand network state, but may be insufficient to capture events precisely or at fine time-scales. For example, SNMP provides per port counters every few minutes, too coarse-grained for traffic engineering or performance diagnosis. OpenFlow provides counters for aggregated flows (due to limited TCAM sizes (Chapter 2)) and reports the updated counters every few seconds [42, 119], which cannot capture sub-second traffic events. sFlow [148] uses packet sampling which cannot capture transient events (*e.g.,* transient packet losses), track connection states (*e.g.,* congestion window), and correctly estimate link load [126]. These shortcomings in monitoring systems can lead to significant loss of network availability: a traffic surge in Google's Compute Engine was detected 3 minutes after causing 90% packet loss between two regions [58].

Moreover, today's monitoring systems do not scale well to larger networks with higher capacities and higher utilization. Higher link speed and larger scales mean more packets to monitor; higher network utilization requires more timely event reporting, because delayed reports of an outage can affect larger traffic volumes. Unfortunately, higher utilization also leaves fewer network resources for monitoring. For example, the reporting frequency of OpenFlow counters is inversely proportional to the number of connections managed (increasing new connections from 150 to 250 per second requires reducing reporting frequency from once per second to once per 5 seconds [42]). As a result, the precision of a network monitoring system can suffer at higher scales and utilization and therefore adversely impact the goal of achieving service level objectives and high utilization, especially in data center networks.

Data centers need precise, fine time-scale event monitoring. Data centers need novel kinds of event monitoring capabilities to capture a variety of network misbehaviors (*e.g.,* misconfiguration, transient

looping, anomalies, drops) and as input for network management decisions (*e.g.,* traffic engineering, load balancing, VM migration). We describe a few examples here and list more in Section 3.3.

Identify losses caused by traffic bursts Traffic bursts are common in data centers and can improve application performance [61, 156]. For example, NIC offloading [87] sends packets in batches to reduce processing overhead, and distributed file systems read and write in bulk to maximize disk throughput. However, bursty traffic may cause losses when they traverse shallow buffered switches [136], which significantly affects application performance. To improve performance, it may be necessary to detect lost packets (*e.g.,* by retransmissions) and packets in bursts (*e.g.,* by tracking packet timestamps), and identify their correlations (*e.g.,* by correlating packet sequence numbers). All these detections are not possible using packet sampling or flow level counters.

Identify root cause of congestion Transient network congestion (*e.g.,* incast [28]) is hard to diagnose. For example, a MapReduce reducer may see significant performance degradation caused by network congestion, even though its aggregate demand may be well below switch capacity. This is because another application sending through the same switch can trigger transient incast losses, increasing the reducer's processing time and therefore the job finish time. Such congestion is often caused by short bursts of traffic at short time-scales (10 ms). This may not be detectable on aggregate counters in today's switches (which have $>$ 1s granularity). To diagnose this behavior, it is important to identify TCP flows with high loss and correlate them with heavy hitters going through the bottleneck.

Monitor server load balance and load burst Maintaining the service-level agreements (SLAs) for different applications requires careful provisioning and load balancing in cloud services [117] because imperfect load balance and short request bursts can lead to long tail latency [76, 72]. A good way to track service load is to monitor the network traffic [13]. If we can identify volume anomalies in short time-scales, we can identify inefficient load balancing across servers, provisioning issues or DDoS attacks on some servers. For example, operators can query whether the long tail latency is because the service sees bursts of requests, or if more than 50% of VMs of a service see a traffic surge as a result of a DDoS attack.

Trumpet: These kinds of event detection are beyond the capabilities of today's deployed systems. In this Chapter, we consider a qualitatively different point in the design space of monitoring systems. We ask: Does there exist a design for a monitoring system which can detect and report *thousands* of events within a few *milliseconds*, where event detections are *precise* because the system processes every packet (rather than, say, sampling), and event specifications can be *flexible* (permitting a range of spatial and temporal scopes of event definition)? Such a monitoring system would be especially useful for automatic diagnosis and control at millisecond time-scales on timely reporting of traffic anomalies, fine-grained flow scheduling, pacing, traffic engineering, VM migration and network reconfigurations.

3.3 Defining Events in Trumpet

Users of Trumpet define *events* using a variant of a match-action language [106], customized for expressing events rather than actions. An event in this language is defined by two elements: a *packet filter* and a *predicate*. A *packet filter* defines the set of packets of interest for the specific event, from among all packets entering, traversing, or leaving a data center (or part thereof) monitored by Trumpet. Filters are expressed using wildcards, ranges or prefix specifications on the appropriate packet header fields. If a packet matches multiple filters, it belongs to set of packet for each of the corresponding events. A *predicate* is simply a Boolean formula that checks for a condition defined on the set of packets defined by a packet filter; when the predicate evaluates to true, the event is said to have occurred. A predicate is usually defined over some aggregation function expressed on *per-packet variables*; users can specify a *spatio-temporal granularity* for the aggregation.

Elements of the event definition language: Each packet in Trumpet is associated with several variables. Table 3.1 shows the variables used for use cases discussed in this Chapter. It is easy to add new variables, and we have left an exploration of the expansion of Trumpet's vocabulary to future work.

The predicate can be defined in terms of mathematical and logical operations on these variables, and aggregation functions (`max`, `min`, `avg`, `count`, `sum`, `stddev`) of these variables. To specify the granularity at which the aggregation is performed, users can specify (a) a `time_interval` over which the predicate is

Variable	Description
volume	the size in bytes of the packet payload
ecn	if the packet is marked with ECN
rwnd	the actual receiver advertised window
ttl	the packet's time-to-live field
rtt	the round-trip time experienced by the packet (as measured from the returning ACK)
is_lost	if the packet was retransmitted at least once
is_burst	if the packet occurred in a burst (packets from a flow with short inter-arrival time [87])
ack	latest ack
seq	maximum sequence number
dup	an estimate of the number of bytes sent because of duplicate acks

Table 3.1: *Packet variables*

Example	Event
Heavy flows to a rack with IP range 10.0.128.0/24	dstIP=10.0.128.0/24, sum(volume)>125KB, 5-tuples, 10ms
Large correlated burst & loss in any flow of a service on 10.0.128.0/24	srcIP=10.0.128.0/24, sum(is_lost & is_burst)>10%, 5-tuples, 10ms
Congestion of each TCP flow	Protocol=TCP, 1 - (ack - ack_lastepoch + dup) / (seq_lastepoch - ack_lastepoch)>0.5, 5-tuples, 10ms
Load spike on a service at 10.0.128.0/24 port:80	(dstIP=10.0.128.0/24 and dstPort=80), sum(volume)>100MB, dstIP/24, 10ms
Reachability loss between service A on 10.0.128.0/24 to B 10.20.93.0/24	(srcIP=10.0.128.0/24 and dstIP=10.20.93.0/24), sum(is_lost)>100, (srcIP and dstIP), 10ms
Popular service dependencies for a tenant on 10.0.128.0/20	srcIP=10.0.128.0/20, sum(volume)>10GB, (dstIP/24 and srcIP/24), 1s

Table 3.2: *Event definitions of examples (Filter, Predicate, Flow_granularity, Time_interval)*

to be evaluated, and (b) a `flow_granularity` that specifies how to bucket the universe of packets, so that the aggregation function is computed over the set of packets in each bucket within the last `time_interval`.

The `flow_granularity` can take values such as `5-tuple`[1] or any aggregation on packet fields (such as `srcIP`,`dstIP` and `srcIP/24`). Figure 3.1 shows two examples of counting the volume of four flows at two different flow granularities (`srcIP`,`dstIP` and `srcIP/24`). At `srcIP`,`dstIP` granularity, the first two flows are bucketed together, while the third flow is in a separate bucket. At `srcIP/24` granularity, the first three flows are placed in the same bucket. The `flow_granularity` specification makes event descriptions much more concise. For example, to find chatty VM pairs in a service, the operator can define an event at the granularity of `srcIP`,`dstIP` without needing to explicitly identify the actual IP addresses of the communicating pairs. In some cases, these IP addresses may not be known in advance:

[1] 5-tuple fields include source and destination IP, source and destination port, and protocol. Although all examples here use 5-tuple, Trumpet flow granularity specifications can include other packet header fields (*e.g.,* MAC, VLAN).

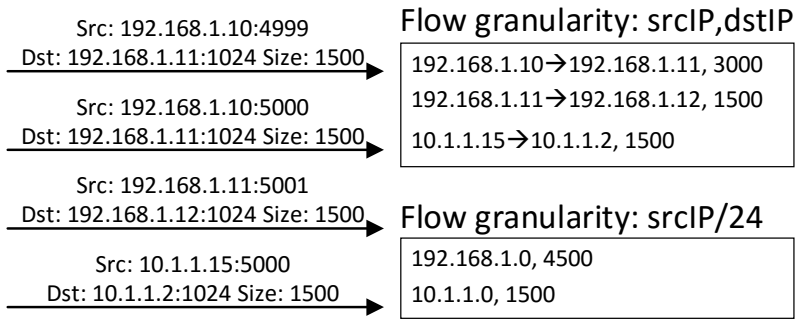


Figure 3.1: *Flow granularity in Trumpet event definition*

e.g., all the users of a datacenter internal service. Additional examples of event definitions with different `flow_granularity` are given below.

Example event definitions: Table 3.2 gives examples of some event definitions based on the discussion in Section 3.2.

The first example is to find heavy hitter flows that send a burst of packets to a rack (say `10.0.128.0/24`). The user of Trumpet would define the IP block as the filter, define a predicate that collects the total number of bytes and checks if the sum exceeds a threshold (say 125 KB as 10% of a 1G link capacity) in a *10 ms time interval* and *5-tuple flow granularity*. Notice that this event definition is expressed at the scale of a rack (say `10.0.128.0/24`), and can flag *any* 5-tuple flow whose behavior matches the predicate.

The second example detects a large correlated burst of losses in any flow of a service whose servers use a given IP block (say `10.0.128.0/24`). The user defines a predicate that counts the number of packets for which `is_lost`[2] and `is_burst` [87] is simultaneously true and checks when the count passes the threshold (the predicate is shown in Table 3.2). Events with `flow_granularity` definitions over multiple flows can be used for coflow scheduling [30], guiding SDN rule placement (Chapter 4) and estimating the demand of FlowGroups for rate limiting [96].

The next event in the table detects when a TCP connection experiences congestion. Specifically, the event predicate checks if a connection does not receive most of the acks for which it was waiting from the beginning of a measurement *epoch* whose duration is the `time_interval` of the predicate. This event

[2]The number of retransmissions over-estimates the number of lost packets. A more accurate solution is more complex and needs careful RTO estimation [9]

tracks `seq`, `ack` and `dup`[3] variables for *all TCP flows in both directions* for the current and previous epochs. It computes the size of acked bytes in the current epoch using the `ack` and `dup` of the other side of connection and compares it against outstanding bytes in the last epoch based on the `ack` and `seq`. Similar events can be used to detect the violation of bandwidth allocation policies [12] at short time-scales, debug new variants of TCP congestion control algorithms, and detect unresponsive VMs to ECN marks and RTT delays.

A slightly different example (the fourth in Table 3.2) detects if there is a load spike on a distributed service (as defined by an IP address block `10.0.128.0/24`). In this example, the predicate evaluates to true if the total volume of traffic to all destination servers within this IP block over a 10 ms time interval exceeds this threshold. For this event, the `flow_granularity` is `dstIP/24`: the whole service.[4]

Beyond the above examples, Trumpet can also be used on other management tasks such as: (a) diagnosing reachability problems between the VMs of two services by counting the total packet losses among any source and destination pair (similar queries can find black holes [61, 156] and transient connectivity problems in middleboxes [123]) and (b) finding popular service dependencies of a tenant by checking if any set of servers in a specific source IP/24 collectively send more than 10GB per second of traffic to a set of servers in a destination IP/24 (a service IP range) (useful, for example, to migrate their VMs to the same rack [74] or put load balancing rules on ToRs with chatty services [54]).

3.4 An Overview of Trumpet

Trumpet is designed to support thousands of dynamically instantiated concurrent events which can result in thousands of triggers at each host. For example, a host may run 50 VMs of different tenants each communicating with 10 services. To support different management tasks (*e.g.,* accounting, anomaly detection, debugging, *etc.*), we may need to define triggers on 10 different per-packet variables (*e.g.,* in Table 3.1) and over different time-intervals and predicates.

[3] Duplicate acks show that a packet is received although it is not the one expected. dup increases by 1460 bytes for each dup-ack and decreases based on acked bytes for each regular ack.

[4] To IP addresses of a service cannot be expressed in a single range, Trumpet allows event specifications with multiple filters.

Trumpet consists of two components (Figure 3.2): the Trumpet Event Manager (TEM) at the controller and a Trumpet Packet Monitor (TPM) at each end-host.

Users submit event descriptions (Section 3.3) to the TEM, which analyzes these descriptions statically and performs the following sequence of actions. First, based on the filter description and on network topology, it determines which end-hosts should monitor the event. Second, it generates *triggers* for each end-host and installs the triggers at the TPM at each host. A trigger is a customized version of the event description that includes filters and predicates in flow and time granularity. However, the predicate in a trigger can be slightly different from the predicate in the corresponding event definition because the event description may describe a network-wide event while a trigger captures a host local event. Third, TEM collects trigger *satisfaction* reports from host TPMs. TPMs generate satisfaction reports when a trigger predicate evaluates to true. For each satisfied trigger, TEM may poll other TPMs that run triggers for the same event in order to determine if the network-wide predicate is satisfied.

An important architectural question in the design of Trumpet is where to place the monitoring functionality (the TPMs). Trumpet chooses end-hosts for various reasons. Other architectural choices may not be as expressive or timely. In-network packet inspection using measurement capabilities at switches lack the flexibility to describe events in terms of per-packet variables; for example, it is hard to get visibility into variables like loss, burst, or RTT at the granularity of individual packets since switches see higher aggregate traffic than end-hosts. Alternatively, sending all packets to the controller to evaluate event descriptions would result in significantly high overhead and might not enable timely detection.

Trumpet's TPM monitors packets in the hypervisor where a software switch passes packets from NICs to VMs and vice versa. This choice leverages the programmability of end-hosts, the visibility of the hypervisor into traffic entering and leaving hosted VMs, and the ability of new CPUs to quickly (*e.g.,* using Direct Data I/O [78]) inspect all packets at fine time-scales. Finally, Trumpet piggybacks on the trend towards dedicating CPU resources to packet switching in software switches [65]: a software switch often runs all processing for each packet in a single core before sending the packet to VMs because it is expensive to mirror packets across cores due to data copy and synchronization overhead [43]. For exactly

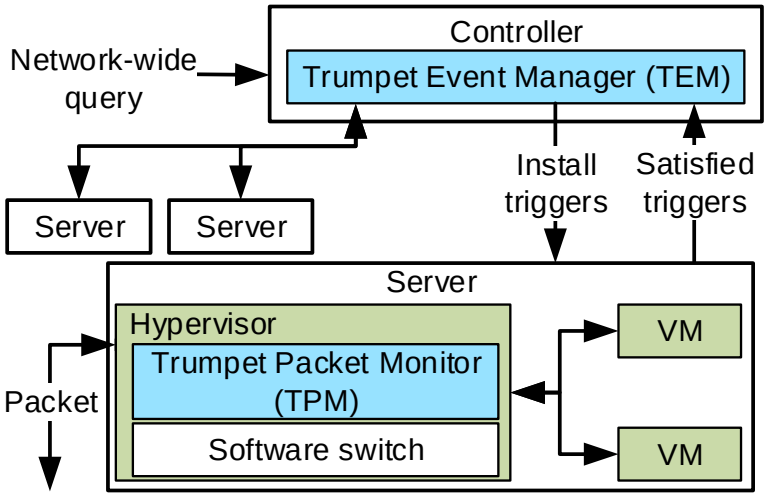


Figure 3.2: *Trumpet system overview*

the same reason, and because it needs complete visibility into all traffic entering or leaving a host, TPM is co-located with the software switch on this core. This trend is predicated on increasing core counts in modern servers.

Trumpet is designed for data centers that use software packet demultiplexing, leveraging its programmability for supporting complex event descriptions, extensibility to new requirements, and resource elasticity to handle load peaks. However, Trumpet can also be used in two ways in data centers where the NIC directly transfers traffic to different VMs (*e.g.,* using kernel bypass, SR-IOV, or receive-side scaling). The first is mirroring traffic to the hypervisor. New NICs allow mirroring traffic to a separate queue that is readable by the hypervisor, using which Trumpet can evaluate trigger predicates. Although this has CPU overhead of processing packets twice (in the hypervisor and VMs), this still preserves the goal of reducing packet latency at VMs. Moreover, because trigger evaluation is not on the packet processing path, Trumpet is not constrained by bounds on packet delay, so it may not need a dedicated core. We have evaluated Trumpet with the mirroring capability in Section 3.7.2. The second is NIC offloading. With the advent of FPGA (*e.g.,* SmartNIC[48]) and network processors at NICs [14], Trumpet can offload some event processing to NICs. For example, it can offload trigger filters in order to selectively send packet headers to the hypervisor. As NIC capabilities evolve, Trumpet may be able to evaluate simpler predicates within

the NIC, leaving CPU cores free to perform even more complex processing tasks (*e.g.,* understanding correlations across multiple flows) for which some of our techniques will continue to be useful.

Trumpet also depends upon being able to inspect packet headers, both IP and TCP, so header encryption could reduce the expressivity of Trumpet.

The design of both TPM and TEM present significant challenges. In the next two sections, we present the design of these components, with a greater emphasis on the systems and scaling challenges posed by precise and timely measurement at the TPM. We also discuss the design of TEM, but a complete design of a *highly scalable* TEM using, for example, techniques from [118, 156], is beyond the scope of this Chapter.

3.5 Trumpet Packet Monitor

The primary challenge in Trumpet is the design of the Trumpet Packet Monitor (TPM). TPM must, at line rate: (a) determine which trigger's filter a packet matches, (b) update statistics of per-packet variables associated with the trigger's predicate and (c) evaluate the predicate at the specified time granularity, which can be as low as 10 milliseconds. For a 10G NIC, in the worst-case (small packet) rate of 14.8 Mpps, these computations must fit within a budget of less than 70ns per packet, on average.

3.5.1 Design Challenges

The need for two-phase processing: Our event definitions constrain the space of possible designs. We cannot perform all of the three steps outlined above when a packet is received. Specifically, the step that checks the predicate must be performed at the specified time-granularity. For instance, a predicate of the form (#packets for a flow during 10ms is *below* 10), can only be evaluated at the end of the 10ms interval. Thus, TPM must consist of two phases: a computation that occurs per-packet (*phase one*), and another that occurs at the predicate evaluation granularity (*phase two*). Now, if we target fine time-scales of a few milliseconds, we cannot rely on the OS CPU scheduler to manage the processing of the two phases because scheduling delays can cause a phase to exceed the per-packet budget discussed above. Hence, Trumpet

piggybacks on the core dedicated to software switching, carefully managing the two phases as described later.

Strawman approaches: To illustrate the challenges underlying the TPM design, consider two different strawman designs: (a) *match-later*, in which phase one simply records a history of packet headers and phase two matches packets to triggers, computes statistics of per-packet variables and evaluates the predicate, and (b) *match-first*, in which phase one matches each incoming packet to its trigger and updates statistics, and phase two simply evaluates the predicate.

Neither of these extremes performs well. With 4096 triggers each of which simply counts the number of packets from an IP address prefix at a time-granularity of 10ms, both options drop 20% of the packets at full 10G packet rate of 14.8 Mpps. Such a high packet rate at a server is common in NFV applications [45], at higher bandwidth links and in certain datacenter applications [102]. A monitoring system cannot induce loss of packets destined to services and applications, and packet loss also degrades the efficacy of the monitoring system itself, so we consider these solutions unacceptable. Moreover, these strawman solutions do not achieve the *full expressivity of our event definitions*: for example, they do not track losses and bursts that require keeping per flow state (Section 3.3). Modifying them to do so would add complexity, which might result in much higher losses at line rates.

Design requirements: These results and the strawman designs help identify several requirements for TPM design. First, both match-first and match-later, even with a state-of-the-art matching implementation, induce loss because the overhead of matching the packet to a filter often exceeds the 70ns budget available to process a single packet at the rate of 14.8 Mpps 64-byte packets on a 10G NIC. This implies that more *efficient per-packet* processing (phase one) is necessary. We also impose the requirement that the system should *degrade gracefully under DoS attacks*, since an attack can disrupt monitoring and attacks on cloud providers are not uncommon [108]. Moreover, with match-first, any delay in processing a packet will add queueing delay to processing subsequent packets: a monitoring system should, ideally, impose *small and bounded delay*.

Second, we observed that match-first scales worse than match-later because it incurs 3× higher TLB and 30% higher cache misses (match-later exhibits much more locality because it performs all packet related actions at once). Thus, *cache and TLB efficiency* is a crucial design requirement for being able to scale trigger processing to full line rate.

3.5.2 Our Approach

In Trumpet, TPM splits the monitoring functions into the following two phases: (a) *Match and scatter* in which incoming packets are matched to a 5-tuple flow[5] (the finest `flow_granularity` specification allowed), which stores per packet counts/other statistics per flow (this scatters statistics across flows), and (b) *Gather-test-and-report*, which runs at the specified trigger time granularity, gathers the statistics to the right `flow_granularity`, evaluates the predicate and reports to TEM when the predicate evaluates to true.

Partitioning the processing in this manner enables Trumpet to satisfy the requirements discussed in the previous subsection:

- As we discuss below, this partitioning permits the design of data structures that promote cache and TLB efficiency, which, as we discuss above, is crucial for performance. Furthermore, the balance of CPU overhead between the two phases permits efficient packet processing, without compromising expressivity: in the first phase, we can minimize matching overhead by caching lookups.
- Small and bounded delays can be achieved by co-operatively scheduling these phases (which avoids synchronization overhead): the second phase is *queue-adaptive* and runs only when the NIC queue is empty.
- Match and scatter per 5-tuple flows allows us to track per flow statistics such as loss and burst and separating that from the gather-test-and-report phase let us compute the statistics once and share these among multiple triggers matching the same flow, but at different flow granularities.

[5]Unlike match-first, which matches against triggers defined at multiple granularities.

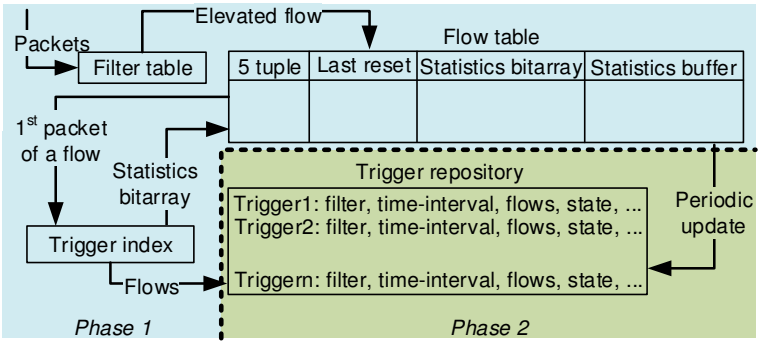


Figure 3.3: *Two-stage approach in Trumpet*

- This partitioning also localizes the two processing bottlenecks in the system: packet processing and gathering statistics. As we discuss later, this allows us to design safeguards for the system to degrade gracefully under attacks, avoiding packet loss completely while maintaining the fidelity of statistics gathering.

3.5.3 Data Structures in Trumpet

Trumpet uses four data structures (Figure 3.3): a *flow table* to store statistics of per-packet variables for the flow, a *trigger repository* contains all the triggers, a *trigger index* for fast matching, and a *filter table* for DoS-resilience. We describe the latter two data structures in detail later.

The *flow table* is a hash table, keyed on the flow's 5-tuple, that keeps, for each flow, only the statistics required for the triggers that match the flow (Figure 3.3). Thus, for example, if a flow only matches a single trigger whose predicate is expressed in terms of volume (payload size), the flow table does not track other per-packet variables (loss and burst, round-trip times, congestion windows, *etc.*). The variables can be shared among triggers, and the TEM tells the TPM which variables to track based on static analysis of event descriptions at trigger installation. The flow table maintains enough memory to store most per-packet statistics, but some, like loss and burst indicators are stored in dynamically allocated data structures. Finally, the TPM maintains a statically allocated overflow pool to deal with hash collisions. The *trigger repository* not only contains the definition and state of triggers, but also tracks a list of 5-tuple flows that match each trigger. In a later section, we discuss how we optimize the layout of these data structures to increase cache efficiency.

```
1  Function processPackets(packetBatch)
2      foreach Packet p do
3          prefetch(p)
4      foreach Packet p do
5          if p.flow != lastpacket.flow then
6              p.hash = calculateHash()
7              prefetchFlowEntry(p)
8      foreach Packet p do
9          if p.flow != lastpacket.flow then
10             e = flowTable.find(p)
11             if e == NULL then
12                 e = flowTable.add(p)
13                 triggers = triggerMatcher.match(p)
14                 e.summaryBitarray = bitarray(triggers.summaries.id)
15         if e.lastUpdate < epoch then
16             e.resetSummaries()
17         e.updateSummaries(p)
18         forwardPacket(p)
```

Figure 3.4: *Processing packets in Trumpet*

3.5.4 Phase 1: Match and Scatter

In this phase, Figure 3.4 runs over every packet. It looks up the packet's 5-tuple in the flow-table and updates the per-packet statistics. If the lookup fails, we use a *matching* algorithm (lines 11-14) to match the packet to one or more triggers (a flow may, in general, match more than one trigger).

Fast trigger matching using tuple search: Matching a packet header against trigger filters is an instance of multi-dimensional matching with wildcard rules. For this, we build a *trigger index* based on the tuple search algorithm [138], which is also used in Open vSwitch [119]. The tuple search algorithm uses the observation that there are only a limited number of patterns in these wildcard filters (*e.g.,* only 32 prefix lengths for the IP prefix). The trigger index consists of multiple hash tables, one for each pattern, each of which stores the filters and the corresponding triggers for each filter. Searching in each hash table involves masking packet header fields to match the hash table's pattern (*e.g.,* for a table defined on a /24 prefix, we mask out the lower 8 bits of the IP address), then hashing the result to obtain the matched triggers. Tuple search memory usage is linear to the number of triggers, and its update time is constant.

Performance optimizations: Since packet processing imposes a limited time budget, we use several optimizations to reduce computation and increase cache efficiency. For performance, software switches

often read a *batch* of packets from the NIC. When we process this batch, we use two forms of *cache prefetching* to reduce packet processing delay: (1) prefetching packet headers to L1 cache (lines 2-3) and (2) prefetching flow table entries (lines 4-7). Data center applications have been observed to generate a burst of packets on the same flow [87], so we cache the result of the last flow table lookup (lines 5, 9). To minimize the impact of TLB misses, we store the flow table in huge pages. In Section 3.7, we demonstrate that each optimization is critical for Trumpet's performance.

3.5.5 Phase 2: Gather, Test, and Report

This phase gathers all statistics from the flow table entries into the `flow_granularity` specified for each trigger (recall that the flow-table stores statistics at 5-tuple granularity, but a trigger may be defined on coarser flow granularities, like `dstIP/24`). Then, it evaluates the predicate, and reports all the predicates that evaluate to true to the TEM.

The simplest implementation, which runs this entire phase in one sweep of triggers, can result in large packet delays or even packet loss, since packets might be queued or dropped in the NIC while this phase is ongoing. Scheduling this phase is one of the trickier aspects of Trumpet's design.

At a high-level, our implementation works as follows. Time is divided into *epochs* of size T, which is the greatest common factor of the time granularities of all the triggers. Trumpet supports a T as small as 10 milliseconds. In each flow table entry, we double-buffer the statistics (like volumes, loss, burst, *etc.*): one buffer collects statistics for the odd-numbered epochs, another for even-numbered epochs. In the i-th epoch, we gather statistics from the $i-1$-th epoch. Thus, double-buffering gives us the flexibility to interleave trigger sweeps with packet processing.

```
1 Function mainLoop(timeBudget)
2     if time to sweep then
3         startSweep()
4     if last sweep is not finished then
5         sweep(timeBudget)
6     while Packets in NIC queue do
7         processPackets(batches of 16 packets)
```

Figure 3.5: *Trumpet main loop*

We schedule this gathering sweep in a *queue-adaptive* fashion (Figure 3.5). When the NIC queue is empty, we run a sweep step for a bounded time (Figure 3.6). Because of Trumpet's careful overall design, it is always able to stay ahead of incoming packets *so that these sweeps are never starved* (Section 3.7). This bound determines the delay imposed by the measurement system, and can be configured. In our experiments, we could bound delay to less than 10 μs (Figure 3.7). Each invocation of this algorithm processes some number of triggers from the trigger repository (lines 4-5 in Figure 3.6). This processing essentially gathers all the statistics from the flow entries (recall that each trigger entry has a list of matched flow entries).

```
1  Function sweep(timeBudget)
2      b = timeBudget / 10
3      foreach t = nextTrigger() do
4          foreach t.FlowList do
5              flowNum = processFlowList(t)
6              t.flowNum += flowNum
7              b = b - (t.avgUpdateTime × flowNum)
8              if b ≤0 then
9                  if passedTime ≥ timeBudget then
10                     saveSweepState()
11                     updateAvgUpdateTime(passedTime)
12                     return
13                 b = timeBudget / 10
14         if epoch - t.lastUpdate ≥ t.timeInterval then
15             if t.condition(t) then
16                 report(t)
17             reset(t)
18             t.lastUpdate = epoch
19     updateAvgUpdateTimes(passedTime)
```

Figure 3.6: *Periodic triggers sweeping*

Once all the flows for a trigger have been processed, the algorithm tests the predicate and reports to the TEM if the predicate evaluates to true (lines 14-18 in Figure 3.6). However, while processing a trigger, the processing bound may be exceeded: in this case, we save the sweep state (line 10), and resume the sweep at that point when the NIC queue is next empty. For each trigger, during a sweep step, instead of computing elapsed time after processing each flow entry, which can be expensive ($\sim$100 cycles), we only compute the actual elapsed time when the estimated elapsed time exceeds a small fraction of the budget (lines 7-13). The estimate is based on the number of flows processed for this trigger so far.

Our approach assumes we can complete sweeping all of the triggers within one epoch (10ms): in Section 3.7.2, we demonstrate that we can achieve this for 4K triggers on one core for a 10G NIC with small packets at full line rate even under DoS attack, and show that removing some of our optimizations (discussed below) can result in *unfinished sweeps*. We also demonstrate that, in many cases, shorter epoch durations cannot be sustained in at least one current CPU.

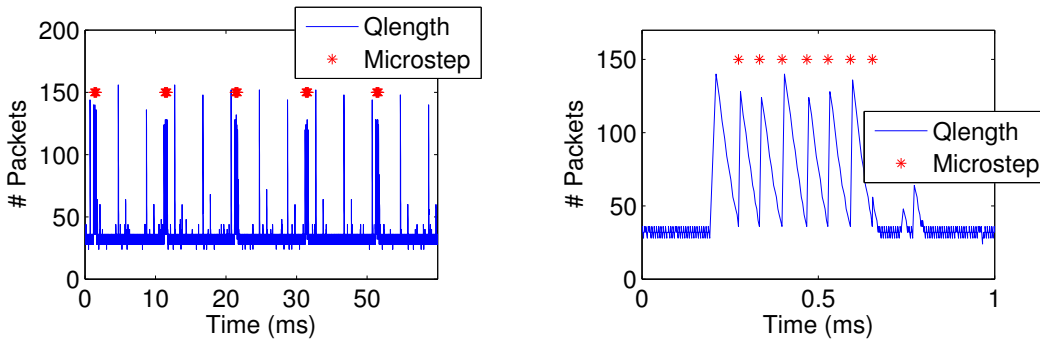
Performance optimizations: We lazily reset counters and remove old flows. In each flow table entry, we store the epoch number when the entry was reset. Then, when we update an entry or read its data during trigger sweep, if the stored epoch number does not match the current, we reset the statistics (lines 15-16 in Figure 3.4).

Trumpet also incorporates several memory optimizations for this phase. Trigger entries are stored contiguously in memory to facilitate hardware cache prefetching. We store the trigger repository in a huge page to reduce TLB misses and store the list of flow entries that match a trigger in a chunked linked list [116] to improve access locality. Each chunk contains 64 flow entries, and these are allocated from a pre-allocated huge page, again for TLB efficiency. For triggers with many flows, chunking reduces the cost of pointer lookups comparing to a linked list of flows at the cost of slightly higher memory usage.

Our final optimization involves efficiently gathering statistics at the required flow granularity. For example, to support a trigger that reports if the volume of traffic to a host (A) from any source IP (at flow-granularity of source IP) is larger than a threshold, we must track the volume per source IP across flows. We can track this with a hash table per trigger, but this adds memory and lookup overhead. Instead, we dynamically instantiate a new instance of the trigger: when a packet from source IP X matches the trigger with a filter dstIP=A, we create a new trigger with filter srcIP=X and dstIP=A.

3.5.6 Degrading Gracefully Under DoS Attacks

It is important for Trumpet to be robust to DoS attacks that aim at exhausting resources in the monitoring system and either cause packet loss, or prevent trigger sweep completion (which would prevent accurate



(a) *Multiple sweeps* **(b)** *Zoomed in View*

Figure 3.7: *Queue-adaptive sweep scheduling strategy is effective at keeping queue sizes below 150 packets or 10 μs.*

detection of events). The expensive steps in Trumpet are matching new flows against triggers in the match-and-scatter phase and updating a trigger based on its flow list in the gather-test-and-report phase.

We assume the attacker knows the design of the system. To attack the system by triggering the largest possible number of expensive match operations, the attacker should send one minimal-sized packet per flow. With this, sweeps might not complete, so trigger reports might not be correctly reported. To mitigate this, when we detect unfinished sweeps, we impose a *DoS threshold*: matching is invoked only on flows whose size in bytes exceeds this threshold in the filter table, and flows will be removed from the flow table if they send fewer bytes than the threshold as new flows collide with them in the flow table or triggers read them at sweep. The DoS threshold can be predicted by profiling the cost of each packet processing operation and each matching operation (Section 3.6). This design comes at the cost of not monitoring very small flows: we quantify this trade-off in Section 3.7.

3.5.7 Summary

Trumpet balances several conflicting goals: event expressivity, tight processing and delay budgets, and efficient core usage. It achieves this by carefully partitioning trigger processing over two phases (match-and-scatter and gather-test-and-report) to keep data access locality per packet processing, keep per flow statistics efficiently, and access trigger information once per epoch. To make match-and-scatter phase efficient, Trumpet uses tuple search matching, NIC polling (DPDK), batching packets, cache prefetching,

huge pages (fewer TLB misses) and caching last flow entry. To minimize processing delay during gather-test-and-report, we proposed a queue-adaptive multi-step sweep. We optimized the sweep to finish within the query time interval using lazy reset (bringing less data into cache), accessing data in a linear fashion (leveraging cache prefetching), checking the time less often, chunking flow entries (fewer pointer jumps) and using huge pages (fewer TLB misses). While some of these are well-known, others such as the adaptive sweep, our two-phase partitioning, and our approach to DDoS resiliency are novel. Moreover, the combination of these techniques is crucial to achieving the conflicting goals. Finally, our experiences with cache and TLB effects and data structure design can inform future efforts in fast packet processing.

3.6 Trumpet Event Manager

Trumpet Event Manager (TEM) translates network-wide events to triggers at hosts, collects satisfied triggers and statistics from hosts, and reports network-wide events. Figure 3.8 shows the detailed process using an example: Consider an event expressed as a predicate over the total traffic volume received by a service on two hosts where the predicate is true if that quantity exceeds a threshold of 10Mbps. In step 1, TEM statically analyzes the event description to determine the trigger predicate, finds the hosts to install the triggers on based on the event filter defined by the service IP addresses and the mapping of IP addresses to hosts. Then, it divides the threshold among the triggers and installs them in step 2. For the example, we set the threshold for each host as half of the event threshold (5Mbps). If neither of the host triggers exceed 5Mbps, their sum cannot be larger than 10Mbps. In step 3, TPMs at hosts measure traffic and

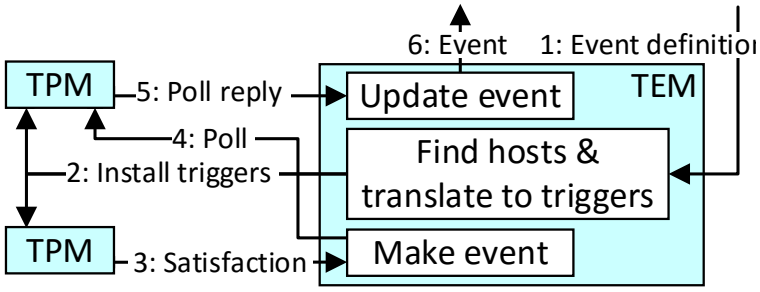


Figure 3.8: *TEM interactions with TPMs*

send trigger satisfaction messages to TEM specifying the event and epoch when evaluating triggers in the gather-test-and-report phase.

In step 4, upon receiving the first satisfaction report for the event, TEM polls the other hosts for their local value of the quantity at that epoch. For our example, a TPM may have sent a satisfaction with value 7Mbps, thus TEM asks for the value at the other TPM to check if its value is above 3Mbps or not. In step 5, TPMs respond to the controller polls after finishing phase 2 when they can steal cycles from packet processing; TPM allows triggers to keep the history of a few last epochs to answer polls. Finally in step 6, TEM evaluates the event after receiving all poll responses. For this to work, TEM relies on time synchronized measurement epochs; the synchronization accuracy of PTP [77] should be sufficient for TEM.

Network wide queries: The approach of dividing a network-wide threshold for an aggregate function can accommodate many kinds of aggregate functions. For example, it is possible to design thresholds for triggers to bound the error on any convex function on the average of quantities distributed among hosts [133] or their standard deviation [52]. Beyond these, Trumpet can also support other kinds of network-wide queries. For example: a) Gather statistics from many events: To find if 50% of VMs (or a certain number of VMs) of a tenant receive traffic exceeding a threshold, we add events for each VM in Trumpet, gather their trigger satisfaction every epoch and report if more than 50% of the related events happened. b) Drill down based on events: TEM can install events conditionally, for example, install a heavy hitter detection event only when another event (*e.g.,* loss) happens. c) Estimating network-wide statistics: We can monitor standard deviation of traffic volume to a set of VMs with a bounded error, ε by defining two events over the standard deviation ($std < std_{old} - \varepsilon$ and $std > std_{old} + \varepsilon$) and updating them accordingly if one of them is satisfied. d) Relative predicates: By feeding the estimate of average and standard deviation to another event, we can detect outlier VMs that receive more than $k\times$ standard deviation above the average.

40G and beyond: Although Trumpet needs only one core to process 14.8Mpps small packets on 10G links, or full line rate 650 byte packets on 40G links, at 100G and/or with smaller packet sizes, multiple

cores might be necessary. To avoid inter-core synchronization, TEM runs independent TPMs on each core and treats them as independent entities. Some synchronization overhead is encountered at the TEM, but that is small as it is incurred only when one of triggers is satisfied. Assuming all packets from a flow are usually handled by the same core, this ensures TPMs can keep the state of a flow correctly. Our network-wide use-case in Section 3.7.1 demonstrates this capability. It is also possible to design, at each host with multiple TPMs, a *local-TEM* which performs similar aggregation at the host itself with minimal synchronization overhead: this can reduce polling overhead and speed up event detection at the TEM.

DoS resiliency: TEM allows operators to set a DoS threshold such that flows whose size are below that threshold are not processed. This reduces matching overhead, allowing Trumpet to degrade gracefully under DoS attacks. TEM calculates the threshold based on a model that profiles offline, using the set of defined triggers (Equation 3.1), the TPM processing costs in the end-host system configuration. Trumpet processing costs include: a) packet processing and checking the filter table, T_P b) matching, T_M c) updating the flow table, T_U and d) sweeping, T_S. The goal is to find the maximum threshold value that keeps total Trumpet processing time, T, below 1s in each second. We can find $T - T_P$ by profiling the free time of the CPU in an experiment that only forwards the traffic with smallest packets that do not pass DoS threshold. Matching overhead per flow (*Match(#patterns)*) is a linear function of # filter patterns with coefficients that can be calculated offline. Similarly, we compute maximum time to update per flow statistics (*Update*) and the time to update a trigger based on a flow (*Sweep*) offline.[6] The factor 1.5 in Equation 3.4 is because the sweep can run in multiple steps while new flows may arrive. As a result, triggers may keep flow entries for both flows that came last epoch and current epoch. Therefore, the sweep may process 50% more entries per trigger on average. We evaluate the accuracy of our model in Section 3.7.2.

[6]The model assumes that the attacker is not aware of the installed triggers, thus triggers only need to process "good" flows. The model can be extended to the case where the attacker knows the installed triggers; we have left this to future work.

$$T = T_P + T_M + T_U + T_S \tag{3.1}$$

$$T_M = Match(\#patterns) \times (\frac{rate_{dos}}{threshold} + \frac{rate_{good}}{\text{avg pkt per flow}}) \tag{3.2}$$

$$T_U = Update \times rate_{good} \tag{3.3}$$

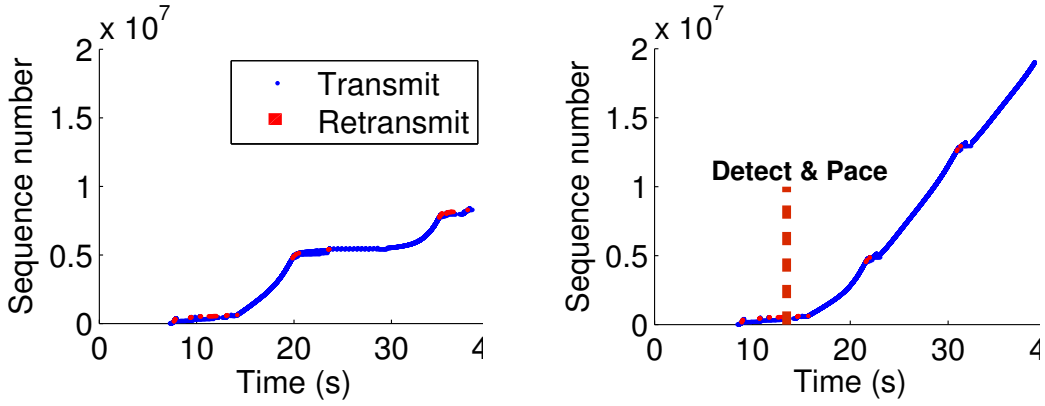
$$T_S = 1.5 \times Sweep \times \text{max trigger per flow} \times \frac{rate_{good}}{\text{avg pkt per flow}} \tag{3.4}$$

TEM Scalability and TPM performance: As we show in Section 3.7, TEM can process nearly 16M trigger satisfaction reports per second per core. If necessary, we can scale TEM even more by sharding events on different servers or by reducing the frequency of polling if TEM divides the threshold unequally based on the history of trigger satisfaction at each host [34]. To avoid delaying packets at end-hosts, TPM uses non-blocking sockets. To reduce poll response latency, TPM batches socket reads and writes and poll processing. TEM can, if necessary (left to future work), reduce TPM overhead by slicing event filters to minimize the number of trigger patterns for matching, and by time multiplexing triggers on hosts by (un)installing them over time.

3.7 Evaluation

In this section, we evaluate Trumpet's expressivity and performance using an implementation of Trumpet's TPM and TEM (10,000 lines of C code).[7] Our implementation uses DPDK 2.2 [44] to bypass the network stack. Our experiments in this section are conducted on 3 Xeon E5-2650 v3 2.30GHz with two 10-core CPUs 25MB L3 and 256KB L2 cache. Our machine has an Intel 82599 10G NIC.

[7] Available at https://github.com/USC-NSL/Trumpet



(a) *Original* **(b)** *With Trumpet+Pacing*

Figure 3.9: *Losses caused by bursts*

3.7.1 Expressivity

Trumpet is expressive enough to track and control fine time-scale flow dynamics, correlate or drill-down on interfering flows, or detect service-scale anomalies. To demonstrate this, we implemented in Trumpet the three use-cases discussed in Section 3.2 whose event definition is presented in Section 3.3.

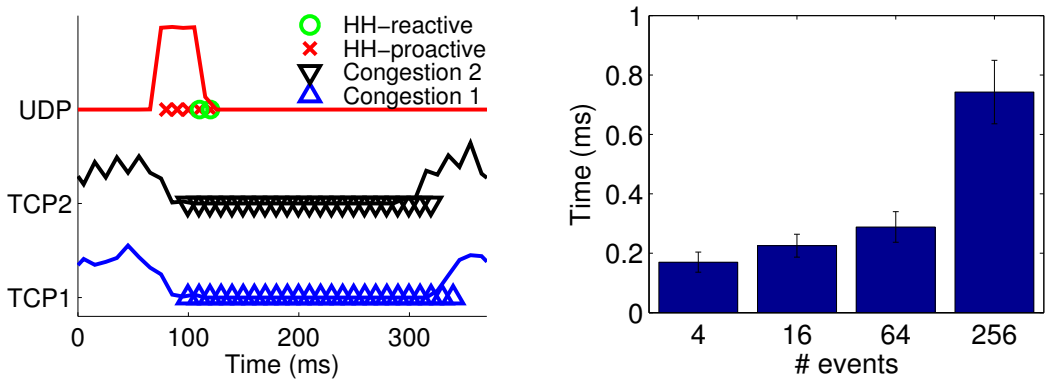
Identifying losses caused by traffic bursts: Trumpet can detect losses caused by traffic bursts and, in a tight control loop, activate pacing to smooth bursts; this can ameliorate the overhead of pacing [4, 56]. Such an approach can be used in data centers with bottlenecks in the wide-area: bursts can be preferentially paced to reduce loss. As an aside, VM stacks, which may often run different transport protocols, can use Trumpet instead of implementing burst (and other kinds of event) detection within their protocol stacks. In our experiment, a user connects to a server in the cloud through an edge router with shallow queues. The connection between the Internet user and the edge router is 10Mbps and incurs 100ms delay. The link from the server to the edge router is 1 Gbps. Figure 3.9a shows that there are packet losses every time there is a traffic burst. Trumpet quickly detects the burst and installs a trigger that informs the controller whenever there are more than 8 lost packets within a burst. Figure 3.9b shows that the trigger is matched after 3 seconds from the starting point of the experiment when the bursts become large enough. Based on the triggers, the controller quickly enables TCP pacing to eliminate the losses and achieves much higher throughput (from 1.94Mbps to 5.3Mbps).

Identifying the root cause of congestion: Despite advances in transport protocols, transient packet losses can be triggered during sudden onset of flows. In data centers, these losses can slow distributed computations and affect job completion times. To diagnose this transient congestion, it is necessary to identify competing flows that might be root causes for the congestion. Trumpet can be used for this task in two ways. In the *reactive* drill-down approach, the operator defines a TCP congestion detection event and, when it is triggered for many flows, programmatically installs another event to find heavy hitters at the common bottleneck of those flows. This may miss short flows because of the delay in detecting congestion. In the *proactive* approach, the operator installs both events on all servers in advance and correlates their occurrence.

To demonstrate both approaches, we use three senders and one receiver with a bottleneck bandwidth of 100Mbps. Using iperf, two senders send TCP traffic for 20s, and after 15s, the third sender sends a short UDP flow (512KB in 40ms) and creates congestion. We define a congestion detection event to report if, at the end of an epoch, any 5-tuple flow cannot receive acks for more than 50% of outstanding data at the beginning of the epoch (Table 3.2). We define the heavy hitter detection event to report if the volume of any 5-tuple flow is larger than 12KB in every 10ms (10% of link bandwidth).

Figure 3.10a shows the normalized throughput of the three connections measured at the receiver as lines and the detected events at the controller as dots. The congestion event could correctly detect throughput loss in TCP connections three epochs after the start of the UDP traffic. The delay is partly because it takes time for the UDP traffic to affect TCP connection throughput. After detecting congestion, TEM installs the `HH_Reactive` event to detect heavy hitters with 200μs delay (order of ping delay). `HH_Reactive` can correctly detect the UDP connection in the fourth epoch after its initiation. Figure 3.10a shows that the proactive approach (`HH_proactive`) could also correctly detect the UDP flow over its lifetime.

Network-wide event to detect transient service-scale anomaly: This use-case shows how Trumpet can detect if the total volume of traffic of a service exceeds a given threshold within a 10ms window. In our experiment, two end-hosts each monitor two 10G ports on two cores. The controller treats each core as a separate end-host (this demonstrates how Trumpet can handle higher NIC speeds). The RTT between



(a) *Congestion diagnosis* **(b)** *Network-wide scenario*

Figure 3.10: *Network-wide and congestion usecases*

the controller and end-hosts is 170 μs. We generate different number of triggers that are satisfied every measurement epoch. When the predicate of a trigger is satisfied at an end-host, the end-host sends a satisfaction message to the controller and the controller polls other servers and collects replies from them.

Figure 3.10b shows the delay between receiving the first satisfaction and the last poll reply at the controller for different numbers of events. Even with 256 satisfied events, the average delay is only about 0.75ms. The average delay for processing satisfactions and generating polls at the controller is 60ns, showing that Trumpet controller can potentially scale to more than 16M messages from TPMs per second per core.

3.7.2 Performance

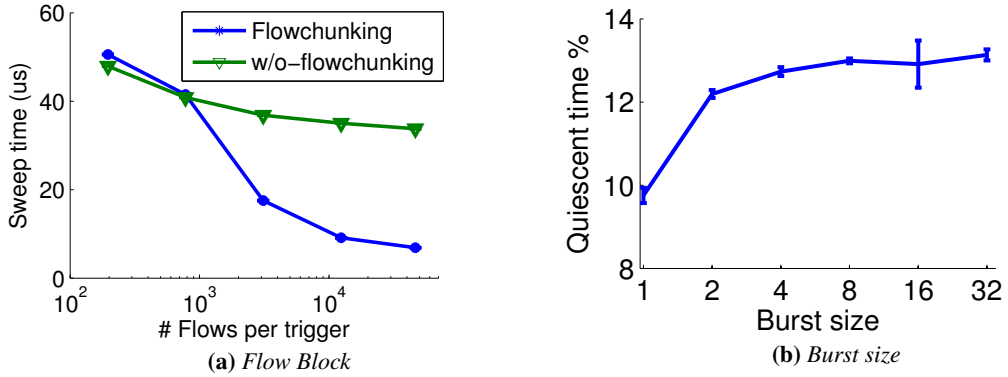
Methodology: In our experiments, our test machine receives traffic from a sender. The traffic generation pattern we test mirrors published flow arrival statistics for data centers [127]: exponential flow inter-arrival time *at each host* with 1ms average with 300 active flows. The TPM monitors packets and forwards them based on their destination IP. It has 4k triggers and each packet matches 8 triggers. Packets may match multiple triggers when a flow is monitored using multiple queries with different predicates (variable, aggregate function, threshold), time intervals or flow granularities. Triggers track one of 3 statistics: packet count, traffic volume or the number of lost packets. In particular, tracking packet losses requires tracking retransmissions across multiple epochs, which can consume memory and additional CPU cycles. However,

the triggers are designed such that the 8 triggers matched by each packet cover the 3 types of statistics. Also, every trigger is evaluated at the granularity of 10ms. Unless otherwise specified, the default burst size is 1 packet (*i.e.,* every subsequent packet changes its flow tuples), which foils lookup caching. These settings were chosen to saturate the packet processing core on our server. We also explore the parameter space (number of triggers, flow arrival rates, time interval, *etc.*) to characterize the *feasibility region* — the set of parameters for which Trumpet can monitor events without losing a packet or missing a sweep.

We use several metrics to evaluate our schemes, including the fraction of time the CPU was *quiescent* (*i.e.,* not in either of the two phases; this quantity is obtained by aggregating the time between two queue polls that returned no traffic), the time for the sweep phase, and whether a particular design point incurred loss or not. We ran each experiment for 5 times with different flow arrival patterns for 50 seconds. The variance across runs is very small, and we present the average.

Baseline experiment: We ran our synthetic traffic at the maximum packet rate of 14.8Mpps (64B per packet) on a 10G link for the trigger configuration discussed above and found that (no associated figure) TPM (a) *never loses a packet*, (b) is able to *correctly compute* every statistic, and (c) is able to *complete sweeps* within an epoch so triggers are correctly evaluated. We also *got the same results on a 40G (4x10G) NIC* with 650 byte packets at full line rate. For a 40G NIC, we expected to be able to support 256 byte packets (since for a 10G NIC we can support 64 byte packets at line rate). However, in our evaluation setting, TPM has to poll four ports, and this polling introduces overhead, requiring a larger packet size for Trumpet to be feasible at 40G.

The time granularity in this experiment is 10ms, and we earlier showed that other simpler strategies (Section 3.5.1) incur 10-20% packet loss in this regime. This experiment validates our design and suggests that it may be possible to monitor a variety of events precisely and at fine time-scales by leveraging the computing capabilities of end-hosts in data centers. More important, in this baseline experiment, the *CPU is never quiescent*. Finally, we have designed TPM for the worst-case: servers in data centers are unlikely to see sustained 14.88Mpps rate (we quantify below how our system performs at lower rates).



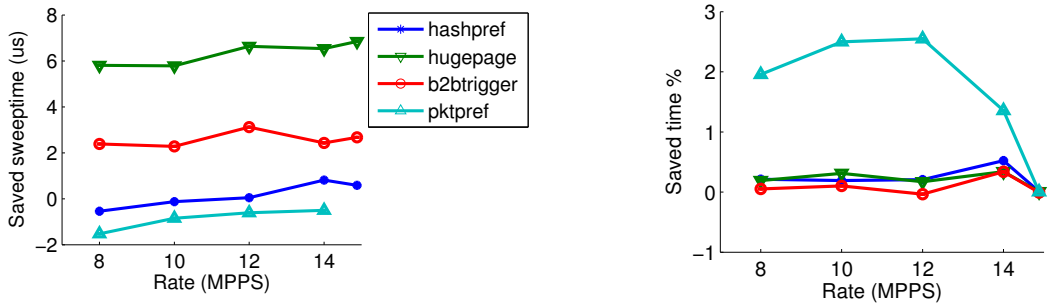
(a) *Flow Block* **(b)** *Burst size*

Figure 3.11: *Optimizations saving*

Match-and-scatter optimizations: We described several optimizations for the match-and-scatter phase in Section 3.5.4. Here we quantify the benefits of each optimization (Figure 3.12). Packet prefetching saves about 2% of CPU time over different packet rates, a significant improvement because we save about 200μs in a 10ms interval. This is more than enough for two sweeps (each sweep takes $< 100\mu s$). More important, recall that at full packet rate, the CPU is never quiescent. Thus, any small increase in CPU time will cause the monitoring system to lose packets, which is completely unacceptable. Indeed, when we turn off packet prefetching, at 14.88Mpps, we experience 4.5% packet loss, and TPM cannot finish the sweep of all the triggers.

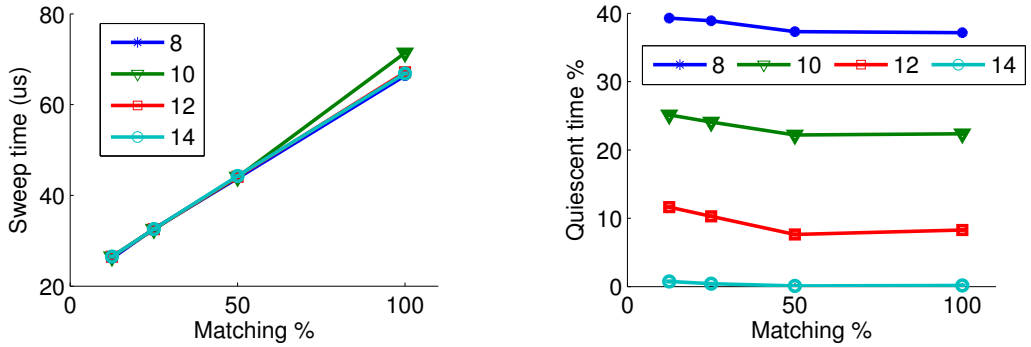
Similarly, although other optimizations contribute small benefits, each of these benefits is critical: without these, TPM would either lose packets or not be able to finish sweeps in time (which can lead to missed events). Using huge pages for storing the flow table and triggers saved 6μs of sweep time. Moreover, huge pages and prefetching flow table entries saves 0.2% of CPU time. To understand the effectiveness of caching the result of the last flow table lookup for a burst of packets, Figure 3.11b shows the impact on quiescent time with different burst sizes for 12Mpps: with the growth of the burst size from 1 to 8, the CPU quiescent time increases from 10% to 13%.

Gather-test-and-report phase optimizations: For a similar reason, even small benefits in this phase can be critical to the viability of TPM. Storing trigger entries contiguously in the trigger repository reduces the sweep time by 2.5μs (Figure 3.12) because it enables hardware prefetching. To evaluate the benefit of



(a) *Sweep time* **(b)** *Quiescent time*

Figure 3.12: *Optimizations saving of flow tables and trigger tables*



(a) *Sweep time* **(b)** *Quiescent time*

Figure 3.13: *Proportional resource usage on % flows matched (legend shows rate in Mpps)*

using chunked lists, we keep the same total number of active flows as 300, but change the number of flows per trigger from 200 to 45k (recall that, in our traffic, 1000 flows arrive at each host every second). Figure 3.11a shows that this can save up to 27μs (80%) in sweep time when there are many flows per trigger, and its overhead is small when there are few flows per trigger.

Resource-proportional design: Trumpet's careful partitioning of functionality results in resource-usage that scales well both with traffic rate, and with the level of *monitored* traffic. This efficient design is the key reason we are able to increase the expressivity of Trumpet's language, and track fairly complex per-packet properties like loss rates and RTTs.

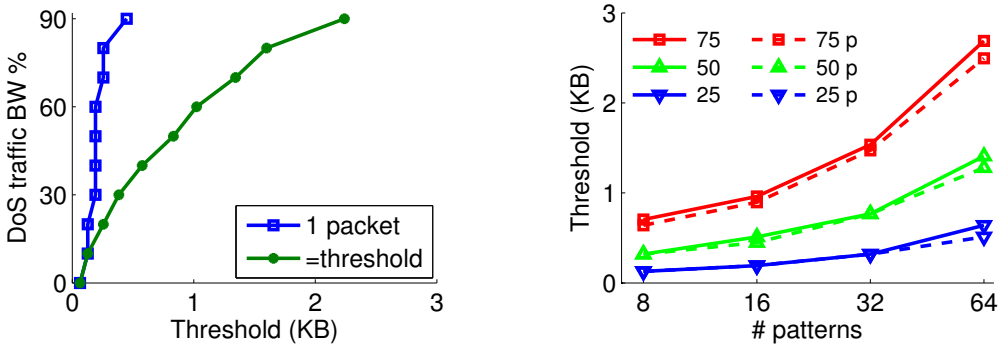
Figure 3.13b shows that CPU quiescent time decreases steadily as a function of the packet rate: at 8Mpps, the system is quiescent 40% of the time, while at 14Mpps it is always fully utilized. In this experiment, the quiescent time is independent of the fraction of matching traffic because the dominant matching cost is incurred for every new flow, regardless of whether the flow matches triggers filters or not.

However, the gather phase scales well with the level of *monitored* traffic: as Figure 3.13a shows, the sweep time is proportional only to the number of flows that match a trigger. Thus if trigger filters match fewer flows, the sweep will take shorter, and the TPM can support smaller time intervals.

DoS-resilience: We perform an experiment in which the TPM is subjected to the full line rate. Legitimate flows send 10 packets of maximum size (1.5KB). We consider two attack models: a) Syn attack where the attacker sends only one packet per flow to maximize packet processing overhead b) Threshold attack where the attacker knows the threshold and sends flows with size equal to the threshold to maximize matching overhead. Also all flows come in a burst of one (no subsequent packets are from the same flow). Our experiment quantifies the percentage of DoS traffic that can be tolerated at each DoS threshold. Thus, it uses two independent knobs: the fraction of DoS traffic in the full packet rate, and the DoS threshold. Each point in Figure 3.14a plots the DoS fraction and the threshold at which the system is functional: even a small increase in one of these can cause the system to fail (either lose packets or not finish sweeps).

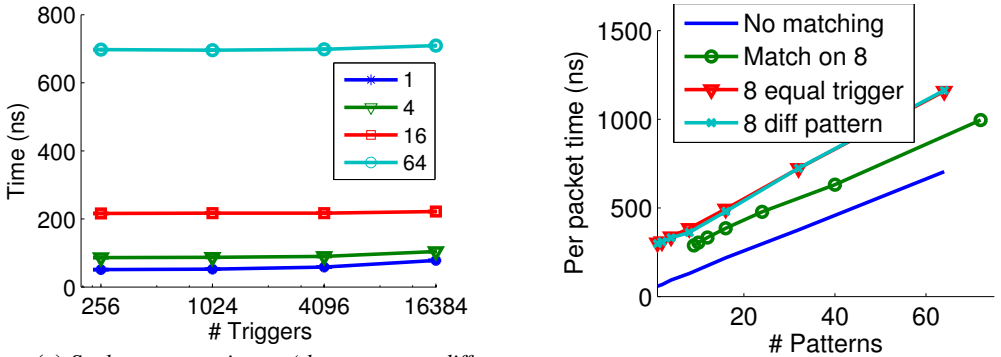
As Figure 3.14a shows, for any threshold more than 440B (on the right of the one packet line), the TPM can sustain a SYN attack. This form of attack is easy for the TPM to handle, since if the traffic is below the threshold, then matching is not incurred. The threshold is higher than 1 packet because there are false positives in the fast counter array implementation of the filter table [6]. At lower thresholds, smaller fractions of attack traffic can be sustained. For the threshold attack, a threshold of 384B (832B) ensures immunity to more than 30% (50%) DoS traffic; this level of bandwidth usage by DoS traffic means that 90% (96%) of packets are from the attacker. The DoS threshold can be decreased even further, to 128 bytes, by checking against a few filters just before matching triggers, to see if a flow would likely match *any* trigger (this is cheaper than checking which trigger a flow matches). At this threshold, a very small fraction of Web server flows in a large content provider would go unmonitored [127].

As discussed in Section 3.6, the DoS threshold can be calculated by a profiling-based model. Figure 3.14b shows that the predicted threshold is close to the experimental threshold over different number of trigger patterns (series show DoS traffic % and suffix “p” means prediction).



(a) *DoS BW per threshold* **(b)** *DoS threshold prediction*

Figure 3.14: *DoS resiliency*



(a) *Scale to many triggers (the curves are different #patterns)* **(b)** *Impact of #patterns*

Figure 3.15: *Performance of trigger matching*

Performance of matching triggers: A key bottleneck in Trumpet is the cost of matching a packet to a trigger filter. Even with aggressive caching of lookups, because our event descriptions involve a multi-dimensional filter, this cost can be prohibitive. Figure 3.15a depicts the impact of matching complexity on Trumpet. Recall that we use the tuple search algorithm to match across multiple dimensions.

The first important result is that this cost is independent of the number of triggers. This is key to our scaling performance: we are able to support nearly 4K concurrent triggers precisely because the tuple search algorithm has this nice scaling property. Where this algorithm scales less well is in the direction of the number of "patterns". Recall that a pattern is a type of filter specification: a filter expressed on only `srcIP` or `dstIP` is a different pattern than one expressed on the conjunction of those two fields. Our matching cost increases with the number of patterns. It may be possible for the TEM to analyze trigger

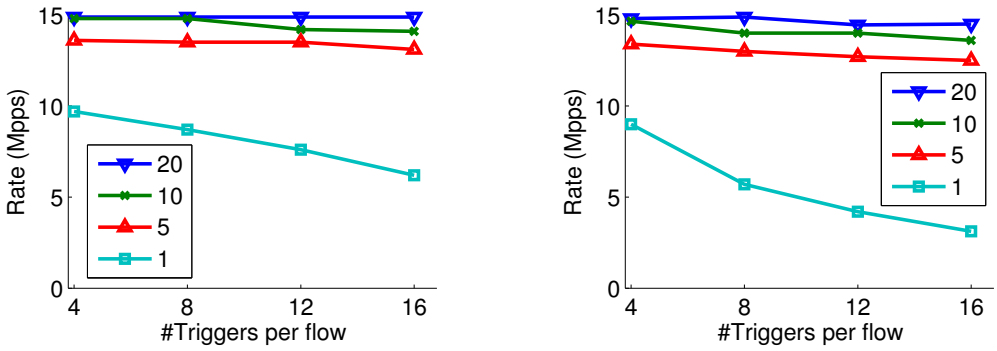
descriptions and avoid installing triggers with too many patterns at a TPM, a form of admission control[8]. To further reduce the impact of the total number of patterns, we can adopt trie indexes [138] to reduce the number of hash table lookups. We have left this to future work.

Finally, in Figure 3.15b we test how matching cost depends on increasingly complex matching scenarios. We consider four such scenarios: *no matching* (no flows match any triggers), *same 8 triggers* (all the flows match the same 8 triggers, the other triggers are not matched), *diff 8 triggers* (each flow matches 8 different triggers, but these triggers have the same filter), and *8 trigger patterns* (each flow matches 8 different trigger patterns). We observe that the per packet processing time increases from *no matching* to *same 8 triggers* and to *diff 8 triggers*. However, the processing time does not further grow with *8 trigger patterns* because our performance does not depend on whether a flow matches different trigger pattern or not, but only depends on the number of patterns.

TPM's feasibility region: We ran experiments with different traffic properties (packet rate, number of active flows, flow arrival rate) and TPM parameters (number of triggers, number of triggers per flow, time interval) to explore the *feasibility region* for the TPM on our server. We call a set of parameters feasible if, for those parameters, TPM does not drop packets and completes all sweeps. The feasibility region is the maximal boundary of feasible parameter sets. We run each parameter set 5 times and show the feasibility region in Figure 3.16 for 300 and 600 active flows per time interval (*e.g.,* 10ms). The settings have different number of triggers per flow (x axis) and different time intervals (series). As the number of triggers per flow increases, the number of triggers also increases accordingly. For example, the setting with 8 triggers per flow has 4k triggers, and the setting with 16 triggers per flow has 8k triggers.

As we decrease the time interval, there will be fewer cycles in an epoch to use for sweeps, and there is a higher chance that the TPM cannot finish the sweep. For example in Figure 3.16a, if the time interval is 5ms and there are 8 triggers per flow, the TPM cannot finish all sweeps for 14.88Mpps. However, reducing the rate to 13.5Mpps (73B packets in 10G), gives more free cycles between packets for sweeping, which

[8]More generally, TEM might perform other forms of admission control, such as rejecting events whose filters span a large part of the address space.



(a) *300 active flows* **(b)** *600 active flows*

Figure 3.16: *Feasibility region over the TPM parameters*

makes the setting feasible. Moreover, as we increase the triggers per flow, the gather phase must process more flow entries, thus can handle lower rates. For example, if the time interval is 10ms and there are 16 triggers per flow, the packet rate must be below 14.1Mpps. Increasing the number of active flows to 600 in Figure 3.16b also increases the number of flow entries to be processed in the gather phase and reduces the feasible packet rate. We have also tried with different flow arrival rates and noticed that the rate does not affect the feasible boundary until it changes the number of active flows in an interval significantly. Cloud operators can characterize the feasibility region for their servers, and then parameterize TPM based on their workload.

TPM and packet mirroring: We have verified the mirroring capability by configuring the NIC to mirror every packet to a reserved queue for TPM while other cores read from the other queues. The TPM could handle 7.25Mpps without any packet loss or unfinished sweeps. This rate is almost half of the 10G maximum line rate as each packet is fetched twice from the NIC.

3.8 Conclusions

In this Chapter, we discuss the design of a new capability in data-center network: active fine time-scale and precise event monitoring. Our event description language is expressive enough to permit novel events that are particularly relevant in data centers. Our algorithms and systems optimizations ensure a design that

can process every packet at line-rate, is DoS-resilient, scales to future network technologies, and permits programmed tight-loop control and drill-down.

In future work, Trumpet can benefit from NIC capabilities such as rule matching to detect events in links with higher packet rates with less CPU overhead. Moreover, in collaboration with monitoring at other vantage points (*e.g.,* switches), Trumpet may achieve the most cost-effective and comprehensive solution for network telemetry and root cause analysis.

Chapter 4

vCRIB: Scalable Rule Management for Data Centers

Cloud operators increasingly need more and more fine-grained rules to better control individual network flows for various traffic management policies. In this chapter, we explore automated rule management in the context of a system called vCRIB (a virtual Cloud Rule Information Base), which provides the abstraction of a centralized rule repository. The challenge in our approach is the design of algorithms that automatically off-load rule processing to overcome resource constraints on hypervisors and/or switches, while minimizing redirection traffic overhead and responding to system dynamics. vCRIB contains novel algorithms for finding feasible rule placements and adapting traffic overhead induced by rule placement in the face of traffic changes and VM migration. We demonstrate that vCRIB can find feasible rule placements with less than 10% traffic overhead even in cases where the traffic-optimal rule placement may be infeasible with respect to hypervisor CPU or memory constraints.

4.1 Introduction

To improve network utilization, application performance, fairness and cloud security among tenants in multi-tenant data centers, recent research has proposed many novel traffic management policies [5, 134, 113, 68]. These policies require *fine-grained* per-VM, per-VM-pair, or per-flow rules. Given the scale of today's data centers, the total number of rules within a data center can be hundreds of thousands or even millions (Section 4.2). Given the expected scale in the number of rules, rule processing in future data

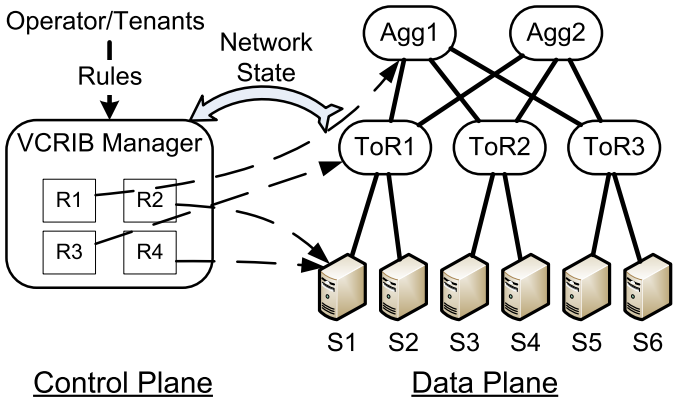


Figure 4.1: *Virtualized Cloud Rule Information Base (vCRIB)*

centers can hit CPU or memory resource constraints at servers (resulting in fewer resources for revenue-generating tenant applications) and rule memory constraints at the cheap, energy-hungry switches.

In this chapter, we argue that future data centers will require *automated rule management* in order to ensure rule placement that respects resource constraints, minimizes traffic overhead, and automatically adapts to dynamics. We describe the design and implementation of a virtual Cloud Rule Information Base (vCRIB), which provides the *abstraction* of a centralized rule repository, and automatically manages rule placement without operator or tenant intervention (Figure 4.1). vCRIB manages rules for different policies in an integrated fashion even in the presence of system dynamics such as traffic changes or VM migration, and is able to manage a variety of data center configurations in which rule processing may be constrained either to switches or servers or may be permitted on both types of devices, and where both CPU and memory constraints may co-exist.

vCRIB's rule placement algorithms achieve resource-feasible, low-overhead rule placement by offloading rule processing to nearby devices, thus trading off some traffic overhead to achieve resource feasibility. This trade-off is managed through a combination of three novel features (Section 4.3).

- Rule offloading is complicated by dependencies between rules caused by overlaps in the rule hyperspace. vCRIB uses per-source rule partitioning with replication, where the partitions encapsulate the dependencies, and replicating rules across partitions avoids rule inflation caused by splitting rules.

- vCRIB uses a *resource-aware placement* algorithm that offloads partitions to other devices in order to find a feasible placement of partitions, while also trying to co-locate partitions which share rules in order to optimize rule memory usage. This algorithm can deal with data center configurations in which some devices are constrained by memory and others by CPU.
- vCRIB also uses a *traffic-aware refinement algorithm* that can, either online, or in batch mode, refine partition placements to reduce traffic overhead while still preserving feasibility. This algorithm avoids local minima by defining novel benefit functions that perturb partitions allowing quicker convergence to feasible low overhead placement.

We evaluate (Section 4.4) vCRIB through large-scale simulations, as well as experiments on a prototype built on Open vSwitch [115] and POX [1]. Our results demonstrate that vCRIB is able to find feasible placements with a few percent traffic overhead, even for a particularly adversarial setting in which the current practice needs more memory than the memory capacity of all the servers combined. In this case, vCRIB is able to find a feasible placement, without relying on switch memory, albeit with about 20% traffic overhead; with modest amounts of switch memory, this overhead drops dramatically to less than 3%. Finally, vCRIB correctly handles heterogeneous resource constraints, imposes minimal additional traffic on core links, and converges within 5 seconds after VM migration or traffic changes.

4.2 Motivation and Challenges

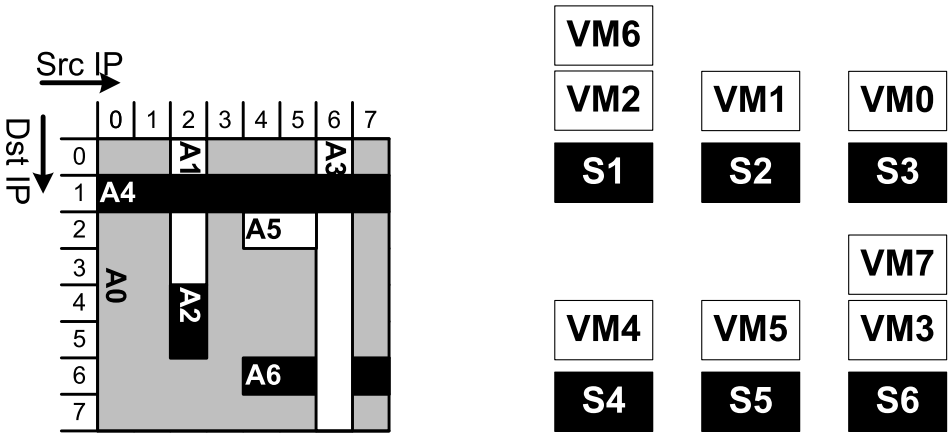
Today, tenants in data centers operated by Amazon [124] or whose servers run software from VMware place their rules at the servers that source traffic. However, multiple tenants at a server may install too many rules at the same server causing unpredictable failures [2]. Rules consume resources at servers, which may otherwise be used for revenue-generating applications, while leaving many switch resources unused.

Motivated by this, we propose to automatically manage rules by offloading rule processing to other devices in the data center. The following paragraphs highlight the main design challenges in scalable automated rule management for data centers.

The need for many fine-grained rules. In this chapter, we consider the class of data centers that provide computing as a service by allowing tenants to rent virtual machines (VMs). In this setting, tenants and data center operators need fine-grained control on VMs and flows to achieve different management *policies*. *Access control policies* either block unwanted traffic, or allocate resources to a group of traffic (*e.g.,* rate limiting [134], fair sharing [122]). For example, to ensure each tenant gets a fair share of the bandwidth, Seawall [134] installs rules that match the source VM address and performs rate limiting on the corresponding flows. *Measurement policies* collect statistics of traffic at different places. For example, to enable customized routing for traffic engineering [5, 16] or energy efficiency [68], an operator may need to get traffic statistics using rules that match each flow (*e.g.,* defined by five tuples) and count its number of bytes or packets. *Routing policies* customize the routing for some types of traffic. For example, Hedera [5] performs specific traffic engineering for large flows, while VLAN-based traffic management solutions [113] use different VLANs to route packets. Most of these policies, expressed in high level languages [70, 147], can be translated into virtual rules at switches[1].

A simple policy can result in a large number of fine-grained rules, especially when operators wish to control individual virtual machines and flows. For example, bandwidth allocation policies require one rule per VM pair [122] or per VM [122], and access control policies might require one rule per VM pair [121]. Data center traffic measurement studies have shown that 11% of server pairs in the same rack and 0.5% of inter-rack server pairs exchange traffic [84], so in a data center with $100K$ servers and 20 VMs per server, there can, be $1G$ to $20G$ rules in total ($200K$ per server) for access control or fair bandwidth allocation. Furthermore, state-of-the-art solutions for traffic engineering in data centers [5, 16, 68] are most effective when *per-flow* statistics are available. In today's data centers, switches routinely handle between 1K to 10K active flows within a one-second interval [17]. Assume a rack with 20 servers and if each server is the source of 50 to 500 active flows, then, for a data center with $100K$ servers, we can have up to $50M$ active flows, and need one measurement rule per-flow.

[1] Translating high-level policies to fine-grained rules is beyond the scope of our work.



(a) *Wild card rules in a flow space* **(b)** *VM assignment*

Figure 4.2: *Sample ruleset (black is accept, white is deny) and VM assignment (VM number is its IP)*

In addition, in a data center where multiple concurrent policies might co-exist, rules may have dependencies between them, so may require carefully designed offloading. For example, a rate-limiting rule at a source VM A can overlap with the access control rule that blocks traffic to destination VM B, because the packets from A to B match both rules. These rules cannot be offloaded to different devices.

Resource constraints. In modern data centers, rules can be processed either at servers (hypervisors) or programmable network switches (*e.g.,* OpenFlow switches). Our focus in this chapter is on flow-based rules that match packets on one or more header fields (*e.g.,* IP addresses, MAC addresses, ports, VLAN tags) and perform various actions on the matching packets (*e.g.,* drop, rate limit, count). Figure 4.2a shows a flow-space with source and destination IP dimensions (in practice, the flow space has 5 dimensions or more covering other packet header fields). We show seven flow-based rules in the space; for example, $A1$ represents a rule that blocks traffic from source IP 2 (VM2) to destination IP 0-3 (VM 0-3).

While software-based hypervisors at servers can support complex rules and actions (*e.g.,* dynamically calculating rates of each flow [134]), they may require committing an entire core or a substantial fraction of a core at each server in the data center. Operators would prefer to allocate as much CPU/memory as possible to client VMs to maximize their revenue; *e.g.,* RackSpace operators prefer not to dedicate even a portion of a server core for rule processing [125]. Some hypervisors offload rule processing to the NIC, which can only handle limited number of rules due to memory constraints. As a result, the number of

rules the hypervisor can support is limited by the available CPU/memory budget for rule processing at the server.

We evaluate the numbers of rules and wildcard entries that can be supported by Open vSwitch, for different values of flow arrival rates and CPU budgets in Figure 4.3. With 50% of a core dedicated for rule processing and a flow arrival rate of 1K flows per second, the hypervisor can only support about 2K rules when there are 600 wildcard entries. This limit can easily be reached for some of the policies described above, so that manual placement of rules at sources can result in *infeasible* rule placement.

To achieve feasible placement, it may be necessary to offload rules from source hypervisors to other devices and redirect traffic to these devices. For instance, suppose VM2, and VM6 are located on S1 (Figure 4.2b). If the hypervisor at S1 does not have enough resources to process the deny rule $A3$ in Figure 4.2a, we can install the rule at ToR1, introducing more traffic overhead. Indeed, some commercial products already support offloading rule processing from hypervisors to ToRs [146]. Similarly, if we were to install a measurement rule that counts traffic between S1 and S2 at Aggr1, it would cause the traffic between S1 and S2 to traverse through Aggr1 and then back. The central challenge is to design a collection of algorithms that manages this tradeoff — keeps the traffic overhead induced by rule offloading low, while respecting the resource constraint.

Offloading these rules to programmable switches, which leverage custom silicon to provide more scalable rule processing than hypervisors, is also subject to resource constraints. Handling the rules using expensive power-hungry TCAMs limits the switch capacity to a few thousand rules [42], and even if this number increases in the future its power and silicon usage limits its applicability. For example, the HP ProCurve 5406zl switch hardware can support about 1500 OpenFlow wildcard rules using TCAMs, and up to 64K Ethernet forwarding entries [42].

Heterogeneity and dynamics. Rule management is further complicated by two other factors. Due to the different design tradeoffs between switches and hypervisors, in the future different data centers may choose to support either programmable switches, hypervisors, or even, especially in data centers with large rule bases, a combination of the two. Moreover, existing data centers may replace some existing devices

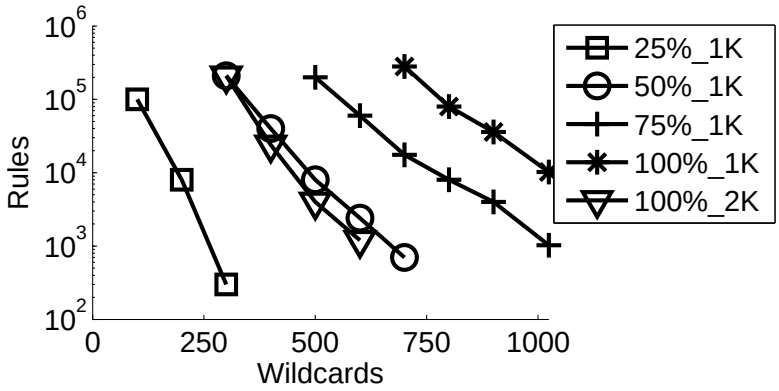


Figure 4.3: *Performance of Open vSwitch (The two numbers in the legend mean CPU usage of one core in percent and number of new flows per second.)*

with new models, resulting in device heterogeneity. Finding feasible placements with low traffic overhead in a large data center with different types of devices and qualitatively different constraints is a significant challenge. For example, in the topology of Figure 4.1, if rules were constrained by an operator to be only on servers, we would need to automatically determine whether to place a measurement rule for tenant traffic between S1 and S2 at one of those servers, but if the operator allowed rule placement at any device, we could choose between S1, ToR1, or S2; in either case, the tenant need not know the rule placement technology.

Today's data centers are highly dynamic environments with policy changes, VM migrations, and traffic changes. For example, if VM2 moves from $S1$ to $S3$, the rules $A0$, $A1$, $A2$ and $A4$ should me moved to $S3$ if there are enough resources at $S3$'s hypervisor. (This decision is complicated by the fact that $A4$ overlaps with $A3$.) When traffic changes, rules may need to be re-placed in order to satisfy resource constraints or reduce traffic overhead.

4.3 vCRIB Automated Rule Management

To address these challenges, we propose the design of a system called vCRIB (virtual Cloud Rule Information Base) (Figure 4.1). vCRIB provides the abstraction of a centralized repository of rules for the cloud. Tenants and operators simply install rules in this repository. Then vCRIB uses network state information including network topology and the traffic information to *proactively* place rules in hypervisors and/or

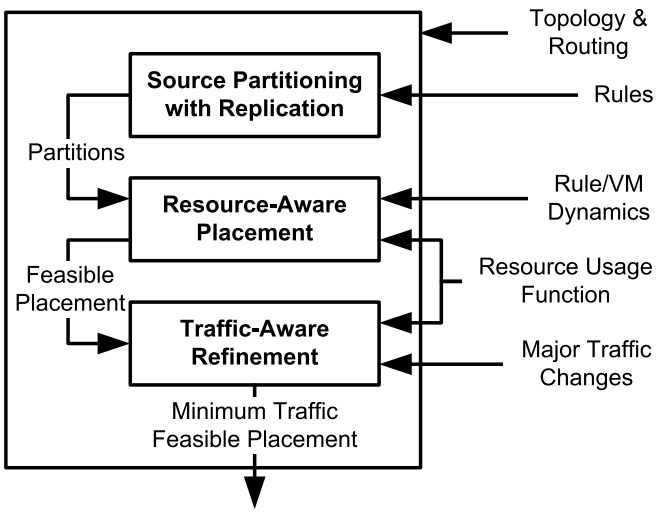


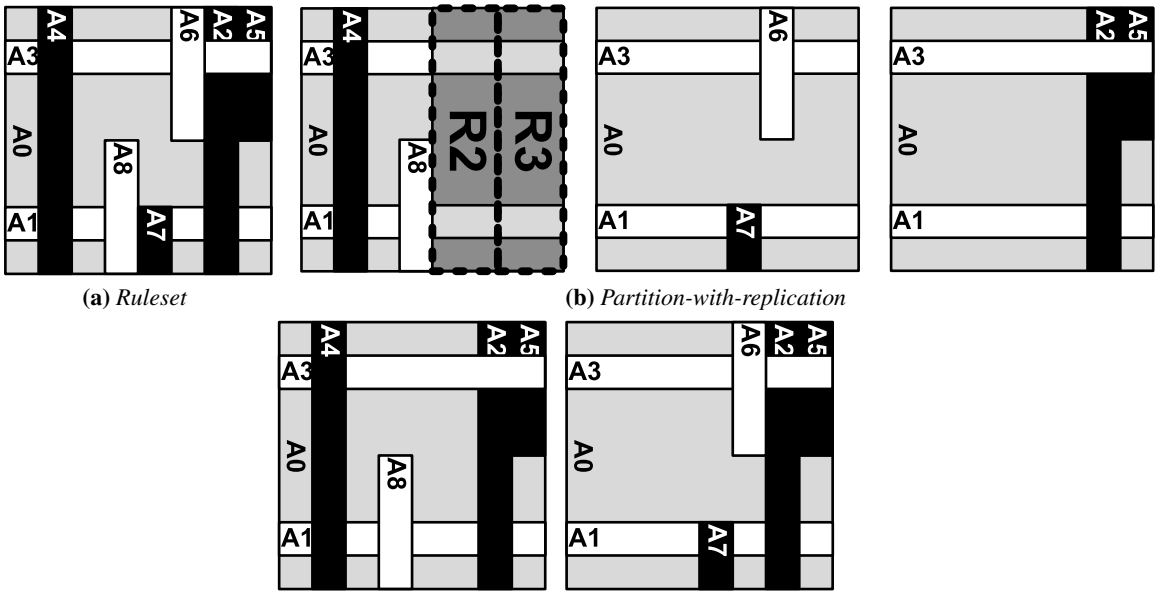
Figure 4.4: *vCRIB controller architecture*

switches in a way that respects resource constraints and minimizes the redirection traffic. Proactive rule placement incurs less controller overhead and lower data-path delays than a *purely reactive* approach, but needs sophisticated solutions to optimize placement and to quickly adapt to cloud dynamics (*e.g.,* traffic changes and VM migrations), which is the subject of this chapter. A hybrid approach, where some rules can be inserted reactively, is left to future work.

Challenges / Designs	Overlapping rules	Resource constraints	Traffic overhead	Heterogeneity	Dynamics
Partitioning with replication	■	■			
Per-source partitions		■	■		
Similarity		■			
Resource usage functions				■	
Resource-aware placement		■			■
Traffic-aware refinement			■		■

Table 4.1: *Design choices and challenges mapping*

vCRIB makes several carefully chosen design decisions (Figure 4.4) that help address the diverse challenges discussed in Section 4.2 (Table 4.1). It partitions the rule space to break dependencies between rules, where each partition contains rules that can be co-located with each other; thus, a partition is the unit



(a) *Ruleset* **(b)** *Partition-with-replication*

(c) *P1 & P3 on a device* **(d)** *P2 & P3 on a device*

Figure 4.5: *Illustration of partition-with-replications (black is accept, white is deny)*

of offloading decisions. Rules that span multiple partitions are *replicated*, rather than split; this reduces rule inflation. vCRIB uses *per-source* partitions: within each partition, all rules have the same VM as the source so only a single rule is required to redirect traffic when that partition is offloaded. When there is *similarity* between co-located partitions (*i.e.,* when partitions share rules), vCRIB is careful not to double resource usage (CPU/memory) for these rules, thereby scaling rule processing better. To accommodate device heterogeneity, vCRIB defines *resource usage functions* that deal with different constraints (CPU, memory *etc.*) in a uniform way. Finally, vCRIB splits the task of finding "good" partition off-loading opportunities into two steps: a novel bin-packing heuristic for *resource-aware partition placement* identifies feasible partition placements that respect resource constraints, and leverage similarity; and a fast *online traffic-aware refinement* algorithm which migrates partitions between devices to explore only feasible solutions while reducing traffic overhead. The split enables vCRIB to quickly adapt to small-scale dynamics (small traffic changes, or migration of a few VMs) without the need to recompute a feasible solution in some cases. These design decisions are discussed below in greater detail.

4.3.1 Rule Partitioning with Replication

The basic idea in vCRIB is to offload the rule processing from source hypervisors and allow more *flexible* and *efficient* placement of rules at both hypervisors and switches, while respecting resource constraints at devices and reducing the traffic overhead of offloading. Different types of rules may be best placed at different places. For instance, placing access control rules in the hypervisor (or at least at the ToR switches) can avoid injecting unwanted traffic into the network. In contrast, operations on the aggregates of traffic (*e.g.,* measuring the traffic traversing the same link) can be easily performed at switches inside the network. Similarly, operations on inbound traffic from the Internet (*e.g.,* load balancing) should be performed at the core/aggregate routers. Rate control is a task that can require cooperation between the hypervisors and the switches. Hypervisors can achieve end-to-end rate control by throttling individual flows or VMs [134], but in-network rate control can directly avoid buffer overflow at switches. Such flexibility can be used to manage resource constraints by moving rules to other devices.

However, rules cannot be moved unilaterally because there can be dependencies among them. Rules can overlap with each other especially when they are derived from different policies. For example, with respect to Figure 4.2, a flow from $VM6$ on server $S1$ to $VM1$ on server $S2$ matches both the rule $A3$ that blocks the source $VM1$ and the rule $A4$ that accepts traffic to destination $VM1$. When rules overlap, operators specify priorities so only the rule with the highest priority takes effect. For example, operators can set $A4$ to have higher priority. Overlapping rules make automated rule management more challenging because they constrain rule placement. For example, if we install $A3$ on S1 but $A4$ on ToR1, the traffic from $VM6$ to $VM1$, which should be accepted, matches $A3$ first and gets blocked.

One way to handle overlapping rules is to divide the flow space into multiple partitions and split the rule that intersects multiple partitions into multiple independent rules, *partition-with-splitting* [152]. Aggressive rule splitting can create many small partitions making it flexible to place the partitions at different switches [111], but can increase the number of rules, resulting in inflation. To minimize splitting,

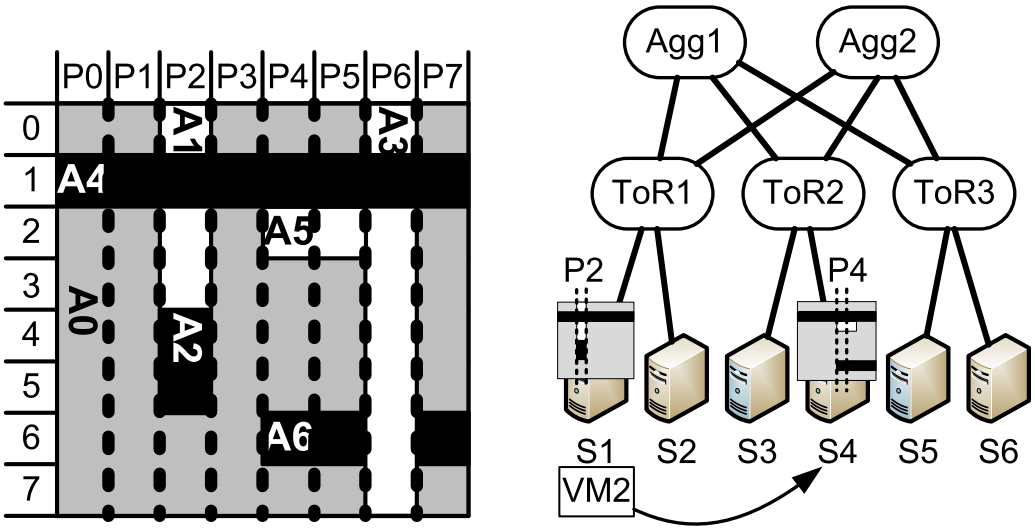
one can define a few large partitions, but these may reduce placement flexibility, since some partitions may not "fit" on some of the devices.

To achieve the flexibility of small partitions while limiting the effect of rule inflation, we propose a *partition-with-replication* approach that replicates the rules across multiple partitions instead of splitting them. Thus, in our approach, each partition contains the original rules that are covered partially or completely by that partition; these rules are not modified (*e.g.,* by splitting). For example, considering the rule set in Figure 4.5a, we can form the three partitions shown in Figure 4.5b. We include both $A1$ and $A3$ in $P1$, the left one, in their original shape. The problem is that there are other rules (*e.g.,* $A2$, $A7$) that overlap with $A1$ and $A3$, so if a packet matches $A1$ at the device where $P1$ is installed, it may take the wrong action – $A1$'s action instead of $A7$'s or $A2$'s action. To address this problem, we leverage redirection rules $R2$ or $R3$ at the source of the packet to completely cover the flow space of $P2$ or $P3$, respectively. In this way, any packets that are outside $P1$'s scope will match the redirection rules and get directed to the current host of the right partition where the packet can match the right rule. Notice that the other alternatives described above also require the same number of redirection rules, but we leverage high priority of the redirection rules to avoid incorrect matches.

Partition-with-replication allows vCRIB to flexibly manage partitions without rule inflation. For example, in Figure 4.5c, we can place partitions $P1$ and $P3$ on one device; the same as in an approach that uses small partitions with rule splitting. The difference is that since $P1$ and $P3$ both have rules $A1$, $A3$ and $A0$, we only need to store 7 rules using partition-with-replication instead of 10 rules using small partitions. On the other hand, we can prove that the total number of rules using partition-with-replication is the same as placing one large partition per device with rule splitting (proof omitted for brevity).

vCRIB generates *per-source* partitions by cutting the flow space based on the source field according to the source IP addresses of each virtual machine. For example, Figure 4.6a presents eight per-source partitions $P0, \cdots, P7$ in the flow space separated by the dotted black lines.

Per-source partitions contain rules for traffic sourced by a single VM. Per-source partitions make the placement and refinement steps simpler. vCRIB only needs one redirection rule installed at the source



(a) *Per-source partitions* **(b)** *partition assignment*

Figure 4.6: *Rule partition example*

hypervisor to direct the traffic to the place where the partition is stored. Unlike per-source partitions, a partition that spans multiple source may need to be replicated; vCRIB does not need to replicate partitions. Partitions are ordered in the source dimension, making it easy to identify similar partitions to place on the same device.

4.3.2 Partition Assignment and Resource Usage

The central challenge in vCRIB design is the assignment of partitions to devices. In general, we can formulate this as an optimization problem, whose goal is to minimize the total traffic overhead subject to the resource constraints at each device.[2]

Suppose there are P partitions where the ith partition has $|P_i|$ rules, and N devices each with the memory size S_j ($j = 1..N$). An indicator variable $M_{i,j}$ denotes that partition P_i is placed on device j. To calculate the traffic overhead for placing partition P_i on device j, for each flow f matching P_i, we calculate the traffic (T) of directing f to j versus placing the partition at $s(f)$ (the source of f): $T^f_{i,j} - T^f_{i,s(f)}$.

The goal of the optimization is to minimize the total traffic overhead $\sum_{i=1..P,j=1..N} \sum_{f\ match\ P_i} M_{i,j}(T^f_{i,j} - T^f_{i,s(f)})$. The resource constraint is that the total number of rules in the partitions at any device should be

[2]One may formulate other optimization problems such as minimizing the resource usage given the traffic usage budget. A similar greedy heuristic can also be devised for these settings.

less than the device memory capacity. (in this formulation, we consider only memory constraints, and later generalize this to CPU constraints). For the approaches that use small partitions with splitting, or one large partition, this is $\sum_i M_{i,j} * |P_i| \leq S_j$ while for our partition-with-replication approach, this constraint is $\left|\bigcup_{i,M_{i,j}=1} P_i\right| \leq S_j$. This problem, even for *partition-with-splitting*, is equivalent to the *generalized assignment problem*, which is NP-hard and even APX-hard to approximate [25]. Although there are some pseudo-polynomial approximation algorithms, they are inefficient for objects with similarity. For example, the algorithm in [32] performs the Knapsack dynamic programming (DP) algorithm for each device. The complexity of this approach is large for our problem not only because of its pseudo-polynomial complexity but also because of similarity between objects: In each step, we need to check all solutions in the previous row of DP table as we do not know adding the new partition to which previous solution will be feasible.

We propose a two-step heuristic algorithm to solve this problem. First, we perform *resource-aware placement* of partitions, a step which only considers resource constraints; next, we perform *traffic-aware refinement*, a step in which partitions reassigned from one device to another to reduce traffic overhead. An alternative approach might have mapped partitions to devices first to minimize traffic overhead (*e.g.,* placing all the partitions at the source), and then refined the assignments to fit resource constraints. With this approach, however, we cannot guarantee that we can find a feasible solution in the second stage. Similar two-step approaches have also been used in the resource-aware placement of VMs across servers [81]. However, placing partitions is more difficult than placing VMs because it is important to co-locate partitions which share rules, and placing partitions at different devices incurs different resource usage.

Before discussing these algorithms, we describe how vCRIB models resource usage in hypervisors and switches in a uniform way. As discussed in Section 4.2, CPU and memory constraints at hypervisors and switches can impact rule placement decisions. We model resource constraints using a function $\mathscr{F}(P,d)$; specifically, $\mathscr{F}(P,d)$ is the percentage of the resource consumed by placing partition P on a device d. $\mathscr{F}$ determines how many rules a device can store, based on the rule patterns (*i.e.,* exact match, prefix-based matching, and match based on wildcard ranges) and the resource constraints (*i.e.,* CPU, memory). For example, for a *hardware OpenFlow switch* d with $s_{TCAM}(d)$ TCAM entries and $s_{SRAM}(d)$ SRAM entries,

the resource consumption $\mathscr{F}(P,d) = r_e(P)/s_{SRAM}(d) + r_w(P)/s_{TCAM}(d)$, where r_e and r_w are the numbers of exact matching rules and wildcard rules in P respectively.

The resource function for *Open vSwitch* is more complicated and depends upon the number of rules $r(P)$ in the partition P, the number of wildcard patterns $w(P)$ in P, and the rate $k(d)$ of new flow arriving at switch d. Figure 4.3 shows the number of rules an Open vSwitch can support for different number of wild card patterns.[3] The number of rules it can support reduces exponentially with the increase of the number of wild card patterns (the y-axis in Figure 4.3 is in log-scale), because Open vSwitch creates a hash table for each wild card pattern and goes through these tables linearly. For a fixed number of wild card patterns and the number of rules, to double the number of new flows that Open vSwitch can support, we must double the CPU allocation.

We capture the CPU resource demand of Open vSwitch as a function of the number of new flows per second matching the rules in partition and the number of rules and wild card patterns handled by it. Using non-linear least squares regression, we achieved a good fit for Open vSwitch performance in Figure 4.3 with the function $\mathscr{F}(P,d) = \alpha(d) \times k(d) \times w(P) \times \log\left(\frac{\beta(d)r(P)}{w(P)}\right)$, where $\alpha = 1.3 \times 10^{-5}$, $\beta = 232$, with $R^2 = 0.95$.[4]

4.3.3 Resource-aware Placement

Resource-aware partition placement where partitions do not have rules in common can be formulated as a bin-packing problem that minimizes the total number of devices to fit all the partitions. This bin-packing problem is NP-hard, but there exist approximation algorithms for it [83]. However, resource-aware partition placement for vCRIB is more challenging since partitions may have rules in common and it is important to co-locate partitions with shared rules in order to save resources.

[3] The IP prefixes with different lengths 10.2.0.0/24 and 10.2.0.0/16 are two wildcard patterns. The number of wildcard patterns can be large when the rules are defined on multiple tuples. For example, the source and destination pairs can have at most 33*33 wildcard patterns.

[4] R^2 is a measure of *goodness of fit* with a value of 1 denoting a perfect fit.

$\mathscr{P}$= set of not placed partitions
while $|\mathscr{P}| > 0$ **do**
  Select a partition P_i randomly
  Place P_i on an empty device M_k.
  repeat
    Select $P_j \in \mathscr{P}$ with maximum similarity to P_i
  until Placing P_j on M_k Fails
end while

Figure 4.7: *First Fit Decreasing Similarity algorithm*

We use a heuristic algorithm for bin-packing similar partitions called *First Fit Decreasing Similarity* (FFDS) (Figure 4.7) which extends the traditional FFD algorithm [135] for bin packing to consider *similarity* between partitions. One way to define similarity between two partitions is as the number of rules they share. For example, the similarity between $P4$ and $P5$ is $|P4 \cap P5| = |P4| + |P5| - |P4 \cup P5| = 4$. However, different devices may have different resource constraints (one may be constrained by CPU, and another by memory). A more general definition of similarity between partitions P_i and P_k on device d is based on the resource consumption function $\mathscr{F}$: our similarity function $\mathscr{F}(P_i,d) + \mathscr{F}(P_k,d) - \mathscr{F}(P_i \cup P_k,d)$ compares the network resource usage of co-locating those partitions.

Given this similarity definition, FFDS first picks a partition P_i randomly and stores it in a new device.[5] Next, we pick partitions similar to P_i until the device cannot fit more. Finally, we repeat the first step till we go through all the partitions.

When we are dealing with per-source partitions in the memory model case, we do not need to compute the similarity of partitions to find the most similar partition to another: Suppose that we have a sorted list of per-source partitions based on the source IP in ascending order and P_i is the ith partition in this list. As rules are continuous boxes in the flow space if a rule is in partition P_i and P_k it must be present in P_j if $i < j < k$.[6] So according to the definition of similarity of two partitions ($|P_1 \cap P_2|$), P_j and P_i similarity is

[5] As a greedy algorithm, one would expect to pick large partitions first. However, since we have different resource functions for different devices, it is hard to pick the large partitions based on different metrics. Fortunately, in theory, picking partitions randomly or greedily do not affect the approximation bound of the algorithm. As an optimization, instead of picking a new device, we can pick the device whose existing rules are most similar to the new partition.

[6] Note that although the part of the rule in P_j may be covered completely by other rules, in partitioning with replication method we do not remove it.

larger than P_k and P_i. Applying the conclusion for k on the other way shows that P_j is more similar to P_k comparing to P_i:

$$|P_i \cap P_j| \geq |P_i \cap P_k| \quad if\ |i-j| \leq |i-k| \tag{4.1}$$

So closer partitions to P_i are always more similar and the most similar ones are the neighbors in the sorted list: P_{i-1} and P_{i+1}. Using this property, we can improve the duration of resource-aware placement step from 2 minutes to a seconds. For example, $P4$ has 4 common rules with $P5$ but 3 common rules with $P7$ in Figure 4.6a. Therefore, when we place $P4$ on an empty device, we only need to check neighbor partitions $P5$ and $P3$.

To illustrate the algorithm, suppose each server in the topology of Figure 4.1 has a capacity of four rules to place the partitions and switches have no capacity. Considering the ruleset in Figure 4.2a, we first pick a random partition $P4$ and place it on an empty device. Then, we check $P3$ and $P5$ and pick $P5$ as it has more similar rules (4 vs 2). Between $P3$ and $P6$, $P6$ is the most similar but the device has no additional capacity for $A3$, so we stop. In the next round, we place $P2$ on an empty device and bring $P1$, $P0$ and $P3$ but stop at $P6$ again. The last device will contain $P6$ and $P7$.

Our FFDS algorithm is inspired by the tree-based placement algorithm proposed in [135], which minimizes the number of servers to place VMs by putting VMs with more common memory pages together. There are three key differences: (1) since we use per-source partitions, it is easier to find the most similar partitions than memory pages; (2) instead of placing sub-trees of VMs in the same device, we place a set of similar partitions in the same device. These similar partitions are not bounded by the boundaries of a sub-tree; and (3) we are able to achieve a tighter approximation bound (2, instead of 3).

To prove the approximation bound, we first go through the procedure presented in [135] for creating a tree to capture similarity of partitions and assigning sub-trees to machines. Then we prove the binary tree guarantees 2-approximation of the *Fractional Packing* lower bound instead of 3-approximation for k-d tree. Fractional Packing approach relaxes the integrality of partitions and assumes that we can break them to arbitrary small pieces. However, the Fractional Packing solution can be worse than the optimal integral

partition assignment if the tree cannot capture 100% similarities of partitions. In the second step, we prove that for the rules with prefix wildcards, the created tree captures 100% of similarity of partitions. Thus the Fractional Packing solution is the lower bound for optimal solution of placing integral partitions. At last, we prove that FFDS algorithm is as good as the binary tree in theory and show that it is better in many cases.

Create the tree Suppose that we have devices with rule capacity M and we want to create a tree of nodes where each node v is associated with a set of rules R_v, a capacity cap_v, an exclusive size w_v, a size $size_v$ and a count c_v.

Creating the tree has three phases: creating the tree-structure bottom-up, updating the capacities of nodes top-down, and updating the count values for nodes bottom-up. Capacities, sizes and counts are 0 by default. To create the tree, we put partitions as the leaves of the tree. Then we create a tree structure out of them recursively by picking the most similar nodes, v_i and v_j, and put them under a new node as the parent, v_p, with $R_{v_p} = R_{v_i} \cap R_{v_j}$ and set $w_{v_i} = |R_{v_i} - R_{v_p}|$ and $w_{v_j} = |R_{v_j} - R_{v_p}|$. In the second phase, we set the capacity of the root node to M. Going top-down on the tree from v_p to its child node v_i, we set $cap_{v_i} = cap_{v_p} - w_{v_p}$. In the last phase, going up in the tree to the parent v_p of v_i and v_j, we set $c_{v_p} = \frac{size_{v_i} + size_{v_j}}{cap_{v_p} - w_{v_p}}$ and $size_{v_p} = size_{v_i} + size_{v_j} + c_{v_p} \times w_{v_p}$. After this procedure, w_v will show the number of exclusive rules saved because of partitions under v and are common among the partitions, cap_v will show the capacity remained on a machine for rules saved exclusively because of partitions under v, $size_v$ will show the total rule spaces needed to save those exclusive rules and most importantly, c_v shows the number of devices required for saving partitions under node v in *Fractional Packing* which is: 1) put each leaf node v on a device with capacity cap_v 2) Recursively, consolidate the devices of children (v_i, v_j) of node v_p into devices with capacity $cap_{v_p} - w_{v_p}$ and add w_{v_p} exclusive rules to them. After each consolidation all c_{v_p} devices (may be except one) will be full. The consolidation relies on the fact that we can break partitions to arbitrary sizes.

Placement: To place the partitions on the tree, we follow Figure 4.8 which is the revised version of the Greedy algorithm presented in [135] for the binary tree. In each round we find partitions under two most similar sub-trees that cannot be placed on a device and place them on two devices.

while tree has a node with count > 1 **do**
  Find a node, v with count $c_v = 2$ where count of its children is 1.
  Place partitions in sub-trees of each children on one device
  Remove v from tree and update *size* and c of its ancestors
end while
Place partitions under root node on one device.

Figure 4.8: *Greedy Packing on tree*

2-approximation bound: The Greedy Packing algorithm (G) uses at most two times devices used by Fractional Packing algorithm (F).

Proof. We use induction on the structure of tree after each round to prove the bound

$$G(T) \leq 2F(T) \tag{4.2}$$

The base case is correct as if $c_{root} = 1$, we use only one device. Suppose T' is created by removing v from the tree T in the nth round of Figure 4.8. We add 2 devices each round so

$$G(T) = G(T') + 2 \tag{4.3}$$

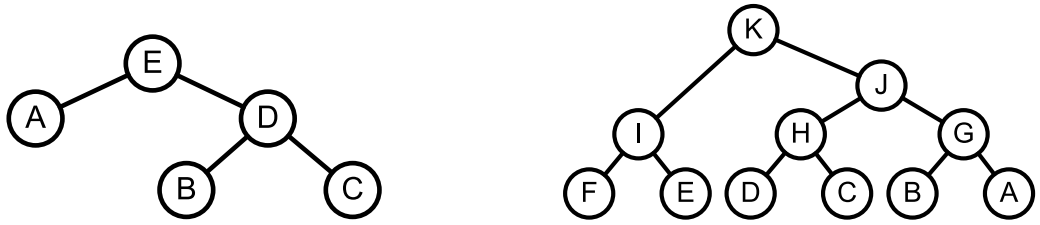
Also the Fractional packing algorithm creates at least one full device for v which will be untouched until the end of the algorithm.

$$F(T') \leq F(T) - 1 \tag{4.4}$$

Putting Equations 4.2, 4.3 and 4.4 together we get:

$$G(T) = G(T') + 2 \leq 2F(T') + 2 \leq 2F(T)$$

□



(a) *Tree captures 100% similarity* **(b)** *A Greedy Packing tree example*

Figure 4.9: *The tree for Greedy Packing algorithm.*

Fractional Packing is a true lower bound: We just proved that we are in 2-approximation bound of the Fractional Packing method. However, the Fractional Packing solution can be larger than optimum if the tree cannot capture the 100% of similarity between partitions. For example, in Figure 4.9a nodes B and C where the most similar. The tree loses similarity of nodes in case A has some similar rules with only B or C. This similarity is not captured by node D and then the node E. So the count value of E can be larger than the optimal solution. However, we prove that for per-source partitions and prefix wildcard rules this situation does not happen. Therefore, Fractional Packing algorithm is the lower bound for the optimal solution. As we use per-source partitions, closer partitions are always the most similar ones (Equation 4.1), so each node has one or more partitions continuously picked from source IP dimension. Note that partitions in A cannot be between B and C in source IP dimension as we know B and C are the most similar and similar nodes are the most close ones. Without loss of generality, assume partitions of node C have larger source IP than of node B. Assume that there is a common rule, R_1, between A and B. If partitions in A have larger source IP than C, node C must also have the rule R_1 as rules are continuous boxes. On the other hand, if partitions in A have smaller source IP than B, it violates the prefix wildcards condition: As B and C are the most similar nodes, they must have at least one common rule, R_2. So R_1 and R_2 share a part of source IP dimension while neither covers another and this cannot happen if rules follow prefix wildcards on the source IP dimension.

FFDS vs. Greedy Packing: In the last step, we show that FFDS algorithm is as good as the Greedy Packing algorithm on the tree and it even works better in many cases. Consider a similarity tree, that the Greedy Packing algorithm put one of its sub-trees on a device. If FFDS starts from one of the partitions in that sub-tree, it will first go through the partitions in that sub-tree because they are most similar. Then, it may add some partitions from adjacent sub-trees. So it deals with an equal or smaller tree in every round. For example, suppose Greedy Packing created the similarity tree in Figure 4.8, for partitions with size 60 where each pair (AB, CD, EF) have 10 common rules. The goal is to place partitions on devices with 170 rule capacities. Greedy Packing algorithm will select node J as its count is two but the count of its children are one. So partitions A and B will be on one device and C and D on another. In the last step it will place F and E on the third device. However, FFDS may start from the partition B[7]. Next, it will put A as it is the most similar to B. FFDS is not bounded by the sub-trees boundaries, so it checks partition C in the next step. Putting partition C on the first device will be successful as it still has 60 free spaces. Following the same approach in the next round, FFDS places partitions D, E and F on the second device. So FFDS used two devices while Greedy Packing needs three devices.

In this section, we proposed FFDS algorithm to minimize the number of switches required to save a set of rule partitions when switches have limited rule capacity. We also proved that FFDS is within 2-approximation bound of the Fractional Packing method (with any partitioning scheme and resource model). Then we proved that, for source-partitioning scheme on switches with memory resource usage model, FFDS algorithm is in 2-approximation bound of optimal placement and proposed a fast way to implement it for that setting. In Section 4.4.4, we show that FFDS is very close to the optimal solution for switches with different memory capacities. The evaluation of FFDS for different ways of partitioning [88, 85] is left to future work.

[7] Starting from other partitions result the same conclusion.

4.3.4 Traffic-aware Refinement

The resource-aware placement places partitions without heed to traffic overhead since a partition may be placed in a device other than the source, but the resulting assignment is *feasible* in the sense that it respects resource constraints. We now describe an algorithm that *refines* this initial placement to reduce traffic overhead, while still maintaining feasibility. Having thus separated placement and refinement, we can run the (usually) fast refinement after small-scale dynamics (some kinds of traffic changes, VM migration, or rule changes) that do not violate resource feasibility. Because each per-source partition matches traffic from exactly one source, the refinement algorithm only stores each partition *once* in the entire network but tries to migrate it closer to its source.

Given per-source partitions, an *overhead-greedy* heuristic would repeatedly pick the partition with the largest traffic overhead, and place it on the device which has enough resources to store the partition and the lowest traffic overhead. However, this algorithm cannot handle dynamics, such as traffic changes or VM migration. This is because in the steady state many partitions are already in their best locations, making it hard to rearrange other partitions to reduce their traffic overhead. For example, in Figure 4.6a, assume the traffic for each rule (excluding A0) is proportional to the area it covers and generated from servers in topology of Figure 4.6b. Suppose each server has a capacity of 5 rules and we put $P4$ on $S4$ which is the source of $VM4$, so it imposes no traffic overhead. Now if $VM2$ migrates from $S1$ to $S4$, we cannot save both $P2$ and $P4$ on $S4$ as it will need space for 6 rules, so one of them must reside on ToR2. As $P2$ has 3 units deny traffic overhead on $A1$ plus 2 units of accept traffic overhead from local flows of $S4$, we need to bring $P4$ out of its sweet spot and put $P2$ instead. However, the overhead-greedy algorithm cannot move $P4$ as it is already in its best location.

To get around this problem, it is important to choose a potential refinement step that not only considers the benefit of moving the selected partition, but also considers the other partitions that might take its place in future refinement steps. We do this by calculating the *benefit* of moving a partition P_i from its current device $d(P_i)$ to a new device j, $M(P_i, j)$. The benefit comes from two parts: (1) The reduction

in traffic (the first term of Equation 4.5); (2) The potential benefit of moving other partitions to $d(P_i)$ using the freed resources from P_i, excluding the lost benefit of moving these partitions to j because P_i takes the resources at j (the second term of Equation 4.5). We define the potential benefit of moving other partitions to a device j as the maximum benefits of moving a partition P_k from a device d to j, *i.e.,* $Q_j = max_{k,d}(T(P_k,d) - T(P_k,j))$. We speed up the calculation of Q_j by only considering the current device of P_k and the best device $b(P_k)$ for P_k with the least traffic overhead. (We omit the reasons for brevity.) In summary, the benefit function is defined as:

$$M(P_i, j) = (T(P_i, d(P_i)) - T(P_i, j)) + (Q_{d(P_i)} - Q_j) \tag{4.5}$$

Update $b(P_i)$ and $Q(d)$
while not timeout **do**
  Update the benefit of moving every P_i to its best feasible target device $M(P_i, b(P_i))$
  Select P_i with the largest benefit $M(P_i, b(P_i))$
  Select the target device j for P_i that maximizes the benefit $M(P_i, j)$
  Update best feasible target devices for partitions and Q's
end while
return the best solution found

Figure 4.10: *Benefit-Greedy algorithm*

Our traffic-aware refinement algorithm is *benefit-greedy*, as described in Figure 4.10. The algorithm is given a time budget (a "timeout") to run; in practice, we have found time budgets of a few seconds to be sufficient to generate low traffic-overhead refinements. At each step, it first picks that partition P_i that would benefit the most by moving to its best feasible device $b(P_i)$, and then picks the most beneficial and feasible device j to move P_i to.[8]

We now illustrate the benefit-greedy algorithm (Figure 4.10) using our running example in Figure 4.6b. The best feasible target device for both $P2$ and $P4$ are $ToR2$. $P2$ maximizes Q_{S4} with value 5 because its deny traffic is 3 and has 1 unit of accept traffic to $VM4$ on $S4$. Also we assume that Q_j is zero for all other devices. In the first step, the benefit of migrating $P2$ to $ToR2$ is larger than moving $P4$ to $ToR2$, while the benefits of all the other migration steps are negative. After moving $P2$ to $ToR2$ the only beneficial step is

[8] By feasible device, we mean the device has enough resources to store the partition according to the function $\mathscr{F}$.

moving $P4$ out of $S4$. After moving $P4$ to $ToR2$, migrating $P2$ to $S4$ become feasible, so Q_{S4} will become 0 and as a result the benefit of this migration step will be 5. So the last step is moving $P2$ to $S4$.

A couple of optimizations to the benefit-greedy algorithm are possible. The algorithm may move a partition out of its "sweet spot" location, but it is not guaranteed to move another partition into that slot. To restore the best position for the partitions, we run the overhead-greedy algorithm at the end of each run of benefit-greedy while respecting the timeout. Instead of predicting if saving a partition will be feasible after bringing out another one, we change the Equation 4.5 to Equation 4.6 by estimating the average number of partitions, α, we need to bring out from their sweet spot to save another. Large α signals that too many partitions must sacrifice for an infeasible partition to move to a better place. So Equation 4.6 will damp the effect of infeasible migrations.

$$M(P_i, j) = (T(P_i, d(P_i)) - T(P_i, j)) + \frac{1}{\alpha}(Q_{d(P_i)} - Q_j) \tag{4.6}$$

An alternative to using a greedy approach would have been to devise a randomized algorithm for perturbing partitions. For example, a Markov approximation method is used in [81] for VM placement. In this approach, checking feasibility of a partition movement to create the links in the Markov chain turns out to be computationally expensive. Moreover, a randomized iterative refinement takes much longer to converge after a traffic change or a VM migration.

4.4 Evaluation

We first use simulations on a large fat-tree topology with many fine-grained rules to study vCRIB's ability to minimize traffic overhead given resource constraints. Next, we explore how the online benefit-greedy algorithm handles rule re-placement as a result of VM migrations. Our simulations are run on a machine with quad-core 3.4 GHz CPU and 16 GB Memory. Finally, we deploy our prototype in a small testbed to understand the overhead at the controller, and end-to-end delay between detecting traffic changes and re-installing the rules.

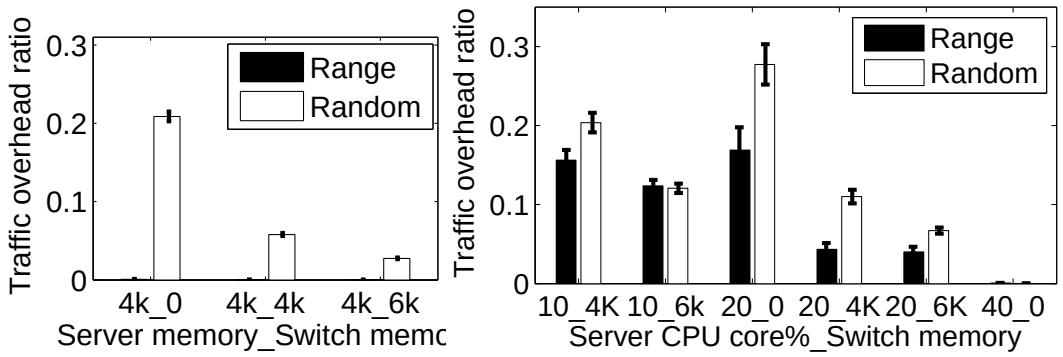
4.4.1 Simulation Setup

Topology: Our simulations use a three-level fat-tree topology with degree 16, containing 1024 servers in 128 racks connected by 320 switches. Since current hypervisor implementations can support multiple concurrent VMs [128], we use 20 VMs per machine. We consider two models of resource constraints at the servers: memory constraints (*e.g.,* when rules are offloaded to a NIC), and CPU constraints (*e.g.,* in Open vSwitch). For switches, we only consider memory constraints.

Rules: Since we do not have access to realistic data center rule bases, we use ClassBench [141] to create 200K synthetic rules each having 5 fields. ClassBench has been shown to generates rules representative of real-world access control.

VM IP address assignment: The IP address assigned to a VM determines the number of rules the VM matches. A random address assignment that is oblivious to the rules generated in the previous set may cause most of the traffic to match the default rule. The problem of selecting VM IPs such that the maximum number of rules are covered is equal to finding the Densest subhypergraph which is NP-Hard and hard to approximate [64]. Instead, we use a heuristic – we first segment the IP range with the boundaries of rules on the source and destination IP dimensions and pick random IP addresses from randomly chosen ranges. We test two arrangements: *Random* allocation which assigns these IPs randomly to servers and *Range* allocation which assigns a block of IPs to each server so the IP addresses of VMs on a server are in the same range.

Flow generation: Following prior work, we use a staggered traffic distribution (ToRP=0.5, PodP=0.3, CoreP=0.2) [5]. We assume that each machine has an average of 1K flows that are uniformly distributed among hosted VMs; this represents larger traffic than has been reported [17], and allows us to stress vCRIB. For each server, we select the source IP of a flow randomly from the VMs hosted on that machine and select the destination IP from one of the target machines matching the traffic distribution specified above. The protocol and port fields of flows also affect the distribution of used rules. The source port



(a) *Memory budget at servers* **(b)** *CPU budget at servers*

Figure 4.11: *Traffic overhead and resource constraints tradeoffs*

is wildcarded for ClassBench rules so we pick that randomly. We pick the destination port based on the protocol fields and the port distributions for different protocols (This helps us cover more rules and do not dwell on different port values for ICMP protocol.). Flow sizes are selected from a Pareto distribution [17]. Since CPU processing is impacted by newly arriving flows, we marked a subset of these flows as new flows in order to exercise the CPU resource constraint [17]. We run each experiment multiple times with different random seeds to get a stable mean and standard deviation.

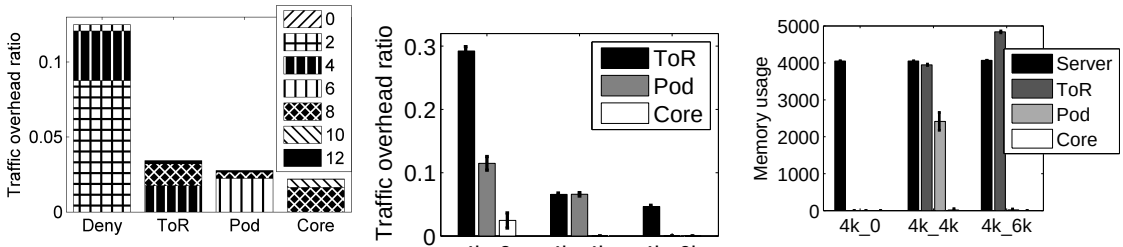
4.4.2 Resource Usage and Traffic Trade-off

The goal of vCRIB rule placement is to minimize the traffic overhead given the resource constraints. To calibrate vCRIB's performance, we compare it against *SourcePlacement*, which stores the rules at the source hypervisor. Our metric for the efficacy of vCRIB's performance is the ratio of traffic as a result of vCRIB's rule placement to the traffic incurred as a result of SourcePlacement (regardless of whether SourcePlacement is feasible or not). When *all* the servers have enough capacity to process rules (*i.e.,* SourcePlacement is feasible), it incurs lowest traffic overhead; in these cases, vCRIB automatically picks the same rule placement as SourcePlacement, so here we only evaluate cases that SourcePlacement is infeasible. We begin with memory resource model at servers because of its simpler similarity model and later compare it with CPU-constrained servers.

vCRIB uses similarity to find feasible solutions when SourcePlacement is infeasible. With *Range* IP allocation, partitions in the Source IP dimension which are similar to each other are saved on one server, so the average load on machines is smaller for SourcePlacement. However, there may still be a few overloaded machines that result in an infeasible SourcePlacement. With *Random* IP allocation, the partitions on a server have low similarity and as a result the average load of machines is larger and there are many overloaded ones. Having the maximum load of machines above 5K in all runs for both Range and Random cases, we set a capacity of 4K for servers and 0 for switches ("4K_0" setting) to make SourcePlacement infeasible. vCRIB could successfully fit all the rules in the servers by leveraging the similarities of partitions and balancing the rules. The power of leveraging similarity is evident when we observe that in the Random case *the average number of rules per machine (4.2K) for SourcePlacement exceeds the server capacity*, yet vCRIB finds a feasible placement by saving similar partitions on the same machine. Moreover, vCRIB finds a feasible solution when we add switch capacity and uses this capacity to optimize traffic (see below), yet SourcePlacement is unable to offload the load.

vCRIB finds a placement with low traffic overhead. Figure 4.11a shows the traffic ratio between vCRIB and SourcePlacement for the Range and Random cases with error bars representing standard deviation for 10 runs. For the Range IP assignment, vCRIB minimizes the traffic overhead under 0.1%. The worst-case traffic overhead for vCRIB is 21% when vCRIB cannot leverage rule processing in switches to place rules and the VM IP address allocation is random, an adversarial setting for vCRIB. The reason is that in the Random case the arrangement of the traffic sources is oblivious to the similarity of partitions. So any feasible placement depending on similarity puts partitions far from their sources and incurs traffic overhead. When it is possible to process rules on switches, *vCRIB's traffic overhead decreases dramatically* (6% (3%) for 4K (6K) rule capacity in internal switches); in these cases, to meet resource constraints, vCRIB places partitions on ToR switches on the path of traffic, incurring minimal overhead. As an aside, these results illustrate the potential for using vCRIB's algorithms for *provisioning*: a data center operator might decide when, and how much, to add switch rule processing resources by exploring the trade-off between traffic and resource usage.

vCRIB can also optimize placement given CPU constraints. We now consider the case where servers may be constrained by CPU allocated for rule processing (Figure 4.11b). We vary the CPU budget allocated to rule processing (10%, 20%, 40%) in combination with zero, 4K or 6K memory at switches. For example in case "40_0" (*i.e.,* each server has 40% CPU budget, but there is no capacity at switches), SourcePlacement results in an infeasible solution, since the highest CPU usage is 56% for range IP allocation and 42% for random IP allocation. In contrast, vCRIB can find feasible solutions in all the cases except "10_0" case. When we have only 10% CPU budget at servers, vCRIB needs some memory space at the switches (*e.g.,* 4K rules) to find a feasible solution. With a 20% CPU budget, vCRIB can find a feasible solution even without any switch capacity ("20_0"). With higher CPU budgets, or with additional switch memory, vCRIB's traffic overhead becomes negligible. Thus, vCRIB can effectively manage heterogeneous resource constraints and find low traffic-overhead placement in these settings. Unlike with memory constraints, Range IP assignment with CPU constraints does not have a lower average load on servers for SourcePlacement, nor does it have a feasible solution with lower traffic overhead, since with the CPU resource usage function closer partitions in the source IP dimension are no longer the most similar.



(a) *Traffic overhead for different rules*

(b) *Traffic overhead on different links*

(c) *Memory usage on different devices*

Figure 4.12: *Spatial distribution of traffic and resource usage*

vCRIB can handle switch only cases. There are scenarios where the operator cannot save the rules at hypervisors. For example, the rules need a special hardware that is only available at switches, or the operator does not want to install hypervisors at servers as it has only one VM per machine. vCRIB works in this case as good as the previous cases owing to the fact that it can adapt the placement of rules based on the available resources in the data center. For example, if it is not possible to place all rules for VMs

of a rack on ToR switch, vCRIB saves the rules on other ToR switches or other internal switches and puts only the forwarding rules on the servers. The Figure 4.13 shows the traffic overhead vCRIB introduces in Range and Random IP assignment cases where putting the rules on ToR switches needs 7951 (5054) and 6974 (5564) rules in maximum (mean) respectively. vCRIB could find a feasible solution for 0K_4K case, while 39% more traffic comparing to saving rules on ToR switches may be unacceptable. However, vCRIB shows that to handle the current ruleset there is no need to upgrade switches to $8K$ rules capacity but only $6K$ rules capacity is enough thanks to vCRIB which finds the optimized placement with only 0.1% traffic overhead for 0K_6K.

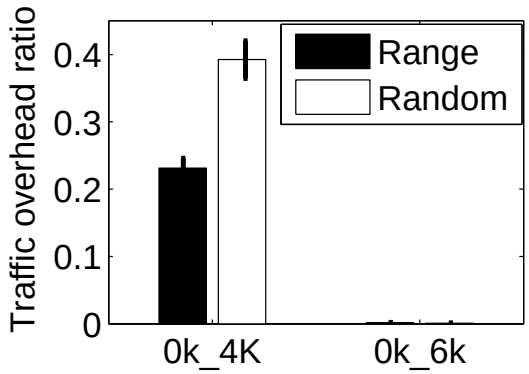


Figure 4.13: *Traffic overhead ratio for switch only cases*

vCRIB's integrated rule management is beneficial. We explore the benefits of vCRIB's ability to jointly manage tenant rules. Suppose we have two tenants in the cloud. Each tenant has 10 of the 20 VMs on each server, its own IP ranges (half of the IP space), defines its own rules on the traffic among its own VMs. Each tenant also generates 500 out of 1K flows of each server. Assume server memory constraint is 4K and switches have no rule processing capacity. We compare two settings: (1) *Disjoint:* Each tenant has half of memory (2K) to run vCRIB independently (2) *Joint:* A single vCRIB controller jointly manage rules for both tenants. Although the disjoint setting already leverages vCRIB to fully leverage its allocated resources and there is no overlapping across the two sets of rules, the disjoint setting has 0.5% more traffic than the Joint setting. This is because the joint setting can balance the rules across tenants at different

servers. For example, if all the flows from one VM of Tenant A should be accepted, the joint setting can save these accept rules at the destination server leaving more space for the rules of Tenant B.

4.4.3 Resource Usage and Traffic Spatial Distribution

We now study how resource usage and traffic overhead are spatially distributed across a data center for the Random case.

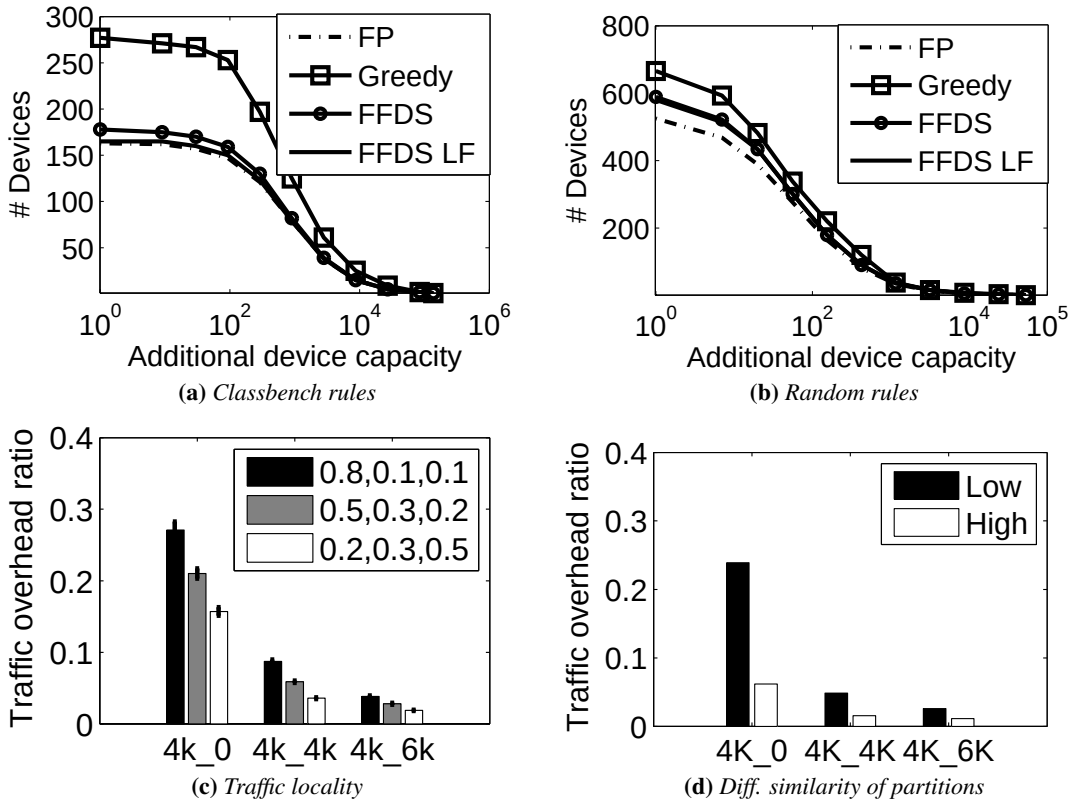
vCRIB is effective in leveraging on-path and nearby devices. Figure 4.12a shows the case where servers have a capacity of 4K and switches have none. We classify the rules into deny rules, accept rules whose traffic stays within the rack (labelled as "ToR"), within the Pod ("Pod"), or goes through the core routers ("Core"). In general, vCRIB may redirect traffic to other locations away from the original paths, causing traffic overhead. We thus classify the traffic overhead based on the hops the traffic incurs, and then normalize the overhead based on the traffic volume in the SourcePlacement approach. Adding the percentage of traffic that is handled in the same rack of the source for deny traffic (8.8%) and source or destination for accept traffic (1.8% ToR, 2.2% POD, and 1.6% Core), shows that out of 21% traffic overhead, about 14.4% is handled in nearby servers.

Most traffic overhead vCRIB introduces is within the rack. Figure 4.12b classifies the locations of the extra traffic vCRIB introduces. vCRIB does not require additional bandwidth resources at the core links; this is advantageous, since core links can limit bisection bandwidth. In part, this can be explained by the fact that only 20% of our traffic traverses core links. However, it can also be explained by the fact that vCRIB places partitions only on ToRs or servers close to the source or destination. For example, in the "4K_0" case, there is 29% traffic overhead in the rack, 11% in the Pod and 2% in the core routers, and based on Figure 4.12c all partitions are saved on servers. However, if we add 4K capacity to internal switches, vCRIB will offload some partitions to switches close to the traffic path to lower the traffic overhead. In this case, for accept rules, the ToR switch is on the path of traffic and does not increase traffic overhead. Next, 4k_6k case, in Figure 4.12c, shows that if we increase the capacity of ToR switches, we can compact all rules saved on internal switches on them and decrease the traffic overhead more. The interesting point

is that 4k_6k uses less resources (computed by summing the bars) comparing to 4k_4k as it can compact more partitions by leveraging their similarity. Note that the servers are always full as they are the best place for saving partitions.

4.4.4 Parameter Sensitivity Analysis

The IP assignment method, traffic locality and rules in partitions can affect vCRIB performance in finding a feasible solution with low traffic. Our previous evaluations have explored *uniform* IP assignment for two extreme cases Range and Random above. In this section, we evaluate the performance of vCRIB for skewed distribution of the number of IPs/VMS per machine, different machine capacities, different traffic locality patterns, and partitions with different sizes and similarities.



(a) *Classbench rules* **(b)** *Random rules*

(c) *Traffic locality* **(d)** *Diff. similarity of partitions*

Figure 4.14: *Parameter sensitivity analysis for traffic patterns, machine sizes and similarity of partitions*

vCRIB is effective for highly skewed rule distribution across servers. Since data center VM placement is dictated by considerations other than rule processing, it may be possible for SourcePlacement to result

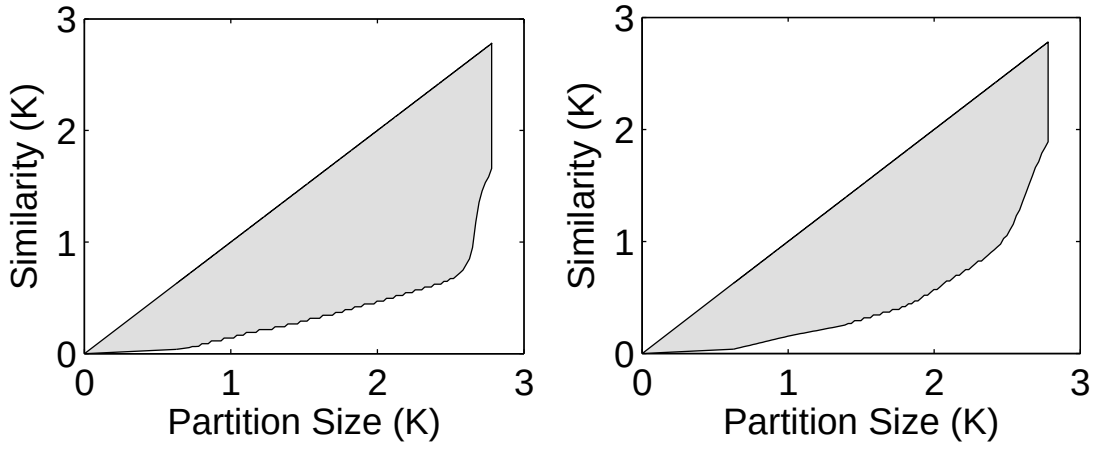
in a *skewed* distribution of rules on servers: *i.e.,* some servers have a large number of rules for example because the VMs placed on them source many flows. To mimic skewed rule distributions, we vary the distribution of #VMs at different servers using a Pareto distribution (with mean 20 and $k = 3$). Compared with the uniform distribution of #VMs, those servers with more VMs have more rules accordingly, and need vCRIB to balance the load. In these cases, vCRIB is able to balance rules across servers, and thus incurs low traffic overhead. vCRIB uses this non-uniformity and puts partitions on under-loaded machines near overloaded ones.

vCRIB is effective for different device capacities. We compare the performance of FFDS algorithm with Greedy Packing algorithm (Figure 4.8) by evaluating the number of used devices when we increase the capacity of devices from the size of the largest partition to the size of all rules in Figure 4.14. Besides evaluating for ClassBench rules (Figure 4.14a), we tested the algorithms for Random rules too (Figure 4.14b). The ranges for these rules are selected based on uniform distribution while making sure that they are prefix wildcard. Firstly, the diagrams show that the performance of Greedy Packing algorithm is within a factor of 2 of the bound for Fractional Packing algorithm. Secondly, it confirms that FFDS works better than Greedy Packing algorithm. Lastly, it shows that selecting the largest partition first (FFDS LF) slightly improves the performance of FFDS.

vCRIB has lower traffic overhead for less local traffic patterns. We explored the effect of traffic locality on vCRIB traffic overhead in Figure 4.14c where the numbers for each series represent the probability of connections inside a rack (ToRP), inside a Pod (PodP) and through the core switches (CorP) respectively for each source server. The result shows that having more non-local flows gives vCRIB more choices to offload rule processing and reach feasible solutions with lower traffic overhead.

vCRIB uses similarity to accommodate larger partitions. We have explored two properties of the rules in partitions by changing the ruleset. In Figure 4.15, we define a two dimensional space: one dimension measures the average similarity between partitions and the other the average size of partitions. Intuitively, the size of partitions is a measure of the difficulty in finding a feasible solution and similarity is the property

of a ruleset that vCRIB exploits to find solutions. To generate this figure, we start from an infeasible setting for SourcePlacement with a maximum of 5.7K rules for "4k_0" setting and then change the ruleset without changing the load on the maximum loaded server. We then explore the two dimensions as follows. Starting from the ClassBench ruleset and Range IP assignment, we split rules into half in the source IP dimension to decrease similarity without changing partition sizes. To increase similarity, we extend a rule in source IP dimension and remove rules in the extended area to maintain the same partition size. Adding or removing rules matching only one VM (micro rules), also help us change average partitions size without changing the similarity. Unfortunately, removing just micro rules is not enough to explore the entire range of partition sizes, so we also remove rules randomly.



(a) *Feasibility region* **(b)** *10% traffic overhead*

Figure 4.15: *vCRIB working region and ruleset properties*

Figure 4.15a presents the *feasibility region* for vCRIB regardless of traffic overhead. Since average similarity cannot be more than the average partition size, the interesting part of the space is below the 45°. Note that vCRIB is able to cover a large part of the space. Moreover, the shape of the feasibility region shows that for a fixed average partition size, vCRIB works better for partitions with larger similarity. This means that to handle larger partitions, vCRIB needs more similarity between partitions; however, this relation is not linear since vCRIB may not be able to utilize the available similarity given limits on server capacity. When considering only solutions with less than 10% traffic overhead, vCRIB's feasibility region

(Figure 4.15b) is only slightly smaller. This figure demonstrates vCRIB's utility: for a small additional traffic overhead, vCRIB can find many additional operating points in a data center that, in many cases, might have otherwise been infeasible.

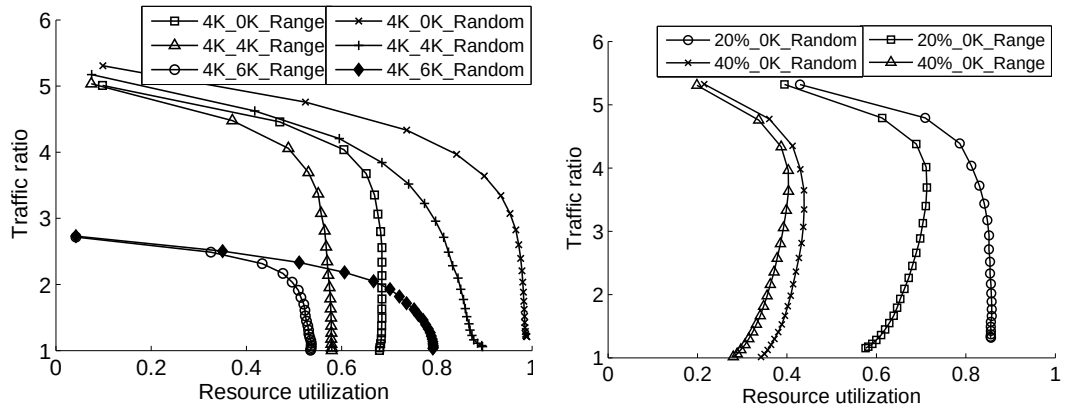
We also tried a different method for exploring the space, by tuning the IP selection method on a fixed ruleset, and obtained qualitatively similar results. To control the similarity of partitions, we adapted our IP address allocation method to tunably control the similarity of partitions assigned to a server in VM placement. Instead of selecting IPs uniformly from the ranges found by rule boundaries, we define a Normal distribution on the source IP dimension. As a result, because of the distribution mass in the middle of the IP range, IP addresses will be closer to each other and more similar. Figure 4.14d shows that the traffic overhead of vCRIB is much lower with high similarity across partitions.

4.4.5 Traffic vs. Resource Usage Trend

We draw the trend of traffic ratio and resource utilization of the data center during the *batch* traffic-refinement for memory model and CPU model case in Figure 4.16b. Each point represents 1000 partition migration steps and the curve starts from the output of resource-aware placement with high traffic and goes to an optimized low traffic placement (top-down). The resource-aware placement algorithm compacts the partitions into a few devices, so there may be many under-loaded devices in the early steps of the traffic-refinement. Therefore, migrating the partitions unpacks the solution of resource-aware placement in the early steps. That is why in Figure 4.16a, each line goes to the right direction fast. However, as the resources around the sources are being used up, migration steps just swap partitions, so the resource usage will be almost the same but the traffic overhead will decrease. Besides, when we add capacity to internal switches, the resource utilization decreases. Part of this decrease is because the rules are compacted into the larger ToR switches and Figure 4.12c confirms that.

The CPU model diagram in Figure 4.16b also shows the same relation between Random vs Range IP assignment schemes and low vs high resources availability. However, the interesting point is that the curves come back to left at the end which means the resource usage decreases even when we are optimizing the

traffic. We believe that this is because of the resource usage required for handling new flows. If we do not save a partition on the source server, *two* machines need to handle all of its new flows: source and current host. However, as vCRIB moves partitions to their source servers (or even destination of accepted flows), the resource usage to handle the partition decreases. In the first few thousand migration steps, the resource usage increases as moving partitions to empty sources increases the number of wild card and rules and its effect is larger than handling flows on sources. Then the reduction of new flows overhead overcomes the effect of wild card and rules. Because similar rules and wildcards will not add any overhead to the devices, but there is no similarity relation among partitions based on new flows.



(a) *Memory model* **(b)** *CPU model*

Figure 4.16: *Traffic ratio vs resource utilization ratio during batch traffic-refinement*

4.4.6 Reaction to Cloud Dynamics

Figure 4.17 compares benefit-greedy (with timeout 10 seconds) with overhead-greedy and a randomized algorithm[9] after a single VM migration for the 4K_0 case. Each point in Figure 4.17 shows a step in which one partition is moved, and the horizontal axis is time in log scale. At time A, we migrate a VM from its current server S_{old} to a new one S_{new}, but S_{new} does not have any space for the partition of the VM, P. As a result, P remains on S_{old} and the traffic overhead increases by $40MBps$. Both benefit-greedy and overhead-greedy move the partition P for the migrated VM to a server in the rack containing S_{new} at time B and reduce traffic by 20Mbps. At time B, benefit-greedy brings out two partitions from their current host

[9]Markov Approximation [81] with target switch selection probability $\propto$ exp(traffic reduction of migration step)

S_{new} to free up the memory for P while imposing a little traffic overhead. At time C, benefit-greedy moves P to S_{new} and reduces traffic further by $15Mbps$. The entire process takes only 5 seconds. In contrast, the randomized algorithm takes 100 seconds to find the right partitions and thus is not useful with these dynamics.

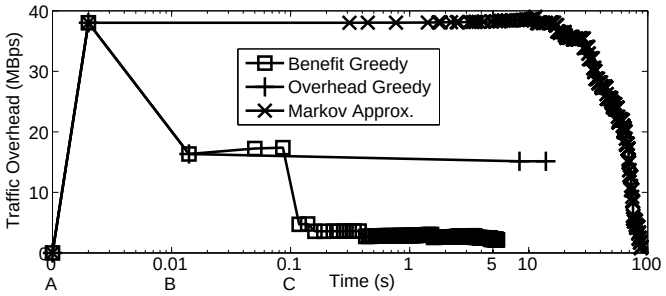


Figure 4.17: *Traffic refinement for one VM migration*

We then run multiple VM migrations to study the average behavior of benefit-greedy with 5 and 10 seconds timeout. In each 20 seconds interval, we randomly pick a VM and move it to another random server. Our simulations last for 30 minutes. The trend of data center traffic in Figure 4.18 shows that benefit-greedy maintains traffic levels, while overhead-greedy is unable to do so. Over time, benefit-greedy (both configurations) reduces the average traffic overhead around 34 MBps, while overhead-greedy algorithm increases the overhead by 117.3 MBps. Besides, this difference increases as the interval between two VM migration increases.

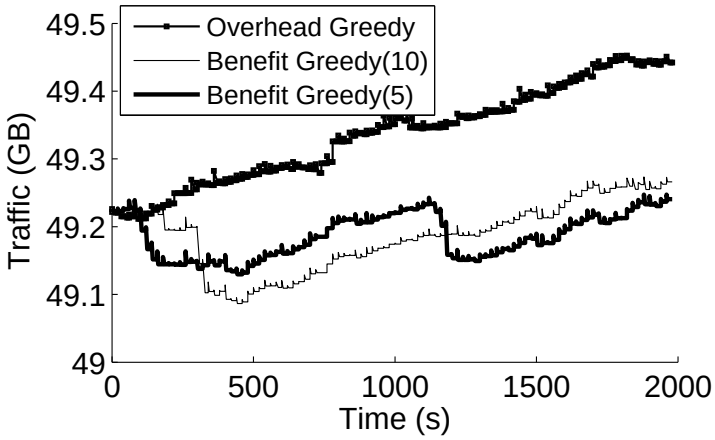


Figure 4.18: *The trend of traffic during multiple VM migrations*

4.4.7 Prototype Evaluation

We built vCRIB prototype using Open vSwitch [115] as servers and switches, and POX [1] as the platform for vCRIB controller for micro-benchmarking.

Overhead of collecting traffic information: In our prototype, we send traffic information collected from each server's Open vSwitch kernel module to the controller. Each piece of information requires 13 Bytes for 5 tuples[10] and 2 Bytes for the traffic change volume.

Since we only need to detect traffic changes at the rule-level, we can more aggressively filter the traffic information than traditional traffic engineering solutions [16]. The vCRIB controller sets a threshold $\delta(F)$ for traffic changes of a set of flows F and sends the threshold to the servers. The servers then only report traffic changes above $\delta(F)$. We set the threshold δ for two different granularities of flow sets F. A larger set F makes vCRIB less sensitive to individual flow changes and leads to less reporting overhead but incurs less accuracy. (1) We set F as the volume each *rule* for each *destination server* in each *per-source partition*. (2) We assume all the rules in a partition have accept actions (as the worst case for traffic). Thus, the vCRIB controller sets the threshold that affects the size of traffic to each *destination server* for each *per-source partition* (summing up all the rules). If there are 20 flow changes above the threshold, we need to send 260B/s per server, which means 20Mbps for 10K servers in the data center. For VM migrations and rule insertion/deletion, the vCRIB controller can be notified directly by the data center management system.

Controller overhead: We measure the delay of processing 200K ClassBench rules. Initially, the vCRIB controller partitions these rules, runs the resource-aware placement algorithm and the traffic-aware refinement to derive an initial placement; this takes up to *five* minutes. However, these recomputations are triggered only when a placement becomes infeasible; this can happen after a long sequence of rule changes or VM add/remove.

[10] Some rules may have more packet header fields and thus require more bytes. In this cases, we can compress these information using fingerprints to reduce the overhead.

The traffic overhead of rule installation and removal depends on the number of refinement steps and the number of rules per partition. The size of OpenFlow command for a rule entry is 100 Bytes, so if a partition has 1K rules, the overhead of removing it from one device and installing at another device is 200KB. For each VM migration, which needs an average of 11 partitions, the bandwidth overhead of moving the rules is 11×200KB=2.2MB.

Reaction to cloud dynamics: We evaluate the latency of handling traffic changes by deploying our prototype in a topology with five switches and six servers as shown in Figure 4.1. We deploy a vCRIB controller that connects with all the devices with an RTT of 20 ms. We set the capacity of each server/switch as large enough to store at most one partition. We then inject a traffic change pattern that causes vCRIB to swap two partitions and add a redirection rule at a VM. It takes vCRIB *30ms* to detect the traffic changes, and move the rules to the new locations. Note that updating rules at hardware switches may take longer than software switches [75]. Therefore, we need to prioritize the partition migrations and update the rules incrementally similar to what is done for DREAM in Chapter 2.

4.5 Conclusions

vCRIB, is a system for automatically managing the fine-grained rules for various management policies in data centers. It jointly optimizes resource usage at both switches and hypervisors while minimizing traffic overhead and quickly adapts to cloud dynamics such as traffic changes and VM migrations. We have validated its design using simulations for large ClassBench rulesets and evaluation on a vCRIB prototype built on Open vSwitch. Our results show that vCRIB can find feasible placements in most cases with very low additional traffic overhead, and its algorithms react quickly to dynamics.

Chapter 5

Literature Review

This dissertation covers topics in data center network management from measurement to control. In this chapter, we present the related work on different techniques for network measurement, resource allocation for measurement tasks and rule management systems.

5.1 Software-defined Measurement at Switches

Prior work has explored different measurement primitives [46, 130], but, unlike DREAM and SCREAM, assumes *offline* analysis of collected measurement data, and thus cannot dynamically change their measurement resource usage when traffic changes or more measurement tasks come.

Previous work on software-defined measurement [150, 112, 82] and programmable measurement [41, 153] has shown the benefits of allowing operators or cloud tenants to customize the measurement for their traffic with different measurement primitives. Amazon CloudWatch [10] also provides simple customized measurement interface for tenants. Like these, DREAM and SCREAM allow measurement tasks to specify the flows and traffic characteristics to measure, but, beyond prior work, provide dynamic resource allocation solutions to enable more and finer-grained measurement tasks.

TCAM-based measurement: Previous TCAM-based algorithms for specific measurement tasks either only work on a single switch [112, 89, 82] or do not adjust counters for bounded resources at switches [155, 89]. We designed a generic divide-and-merge measurement framework for multiple switches with resource

constraints. We also proposed heuristics for estimating the accuracy of TCAM-based measurement algorithms by exploiting relationships between counters already collected.

Sketch-based measurement: There have been many works on leveraging sketches for individual measurement tasks. Some works propose sketch-based measurement solutions on a single switch for HH [36], HHH [39], and SSD [35]. UnivMon [103] proposes to use a universal sketch inside switches that can estimate a broad range of measurement tasks. Other works [3, 63] provide algorithms to run sketches on multiple switches with a fixed sketch size on each switch. Instead, SCREAM provides efficient resource allocation solutions for *multiple* measurement tasks on *multiple switches* with different sketch sizes at these switches. Previous work has proved the theoretical bounds for the *worst case* resource usage for only hash-based measurements [38, 37]. We also proposed heuristics for estimating the accuracy of sketch-based measurement algorithms by exploiting relationships between sketch counters.

5.2 Resource Allocation for Measurement Tasks

CSAMP [131] uses consistent sampling to distribute flow measurement on multiple switches for a single measurement task and aims at maximizing the flow coverage. Volley [107] uses a sampling-based approach to monitor state changes in the network, with the goal of minimizing the number of sampling operations. Payless [31] decides the measurement frequency for concurrent measurement tasks to minimize the controller bandwidth usage, but does not provide any guarantee on accuracy or bound on switch resources. OpenSketch [150] provides a generic data plane that can support many types of sketches with commodity switch components. It leverages the *worst-case* accuracy bounds of sketches to allocate resources on a *single* switch for measurement tasks at task instantiation. In contrast, DREAM focuses on flow-based rules in TCAM. DREAM dynamically allocates network-wide resources to multiple measurement tasks to achieve their given accuracy bound. SCREAM dynamically allocates sketch resources on multiple switches by leveraging the instantaneous accuracy estimation of tasks, and thus can support more tasks with higher accuracy than OpenSketch.

5.3 Timely and Precise Network Measurement

At end-hosts, existing monitoring tools often focus on a specific type of information, require access to VMs or use too much of a resource. For example, SNAP [151] and HONE [140] monitor TCP-level statistics for performance diagnosis. PerfSight [149] instruments hypervisor and VM middleboxes to diagnose packet losses and resource contention among VMs in 100ms timescale. RINC [55] infers internal TCP state of VMs from packets. Pingmesh [61] leverages active probes to diagnose connectivity problems. In contrast, Trumpet shows it is possible to inspect every packet at end-hosts at line speed for a wide set of use-cases. Trumpet runs in the hypervisor and assumes no access to the VM's networking stack. Trumpet can possibly be extended to non-virtualized systems (e.g., containers, by interposing on the network stack), but we have left this to future work. n2disk [114] allows capturing all packets to disk in 10G links using multiple cores. Trumpet aggregates packet-level information on the fly and requires no storage. Packet monitoring functions can be offloaded to programmable NICs [65, 14]. However, these devices currently have limited resources, are hard to program [99], and have limited programmability compared to CPUs. Trumpet can leverage NIC offloading if the hypervisor cannot see packets (*e.g.,* SR-IOV and RDMA). As NICs evolve, it may be possible to save CPU by offloading some parts of Trumpet processing, such as matching, to NICs.

Recent research has proposed a variety of solutions towards more active forms of monitoring, mostly by leveraging switch functionality. Planck [126] mirrors sampled packets at switches. NetSight [66] and EverFlow [156] use switches inside the network to send packet headers to a controller. NetSight incurs bandwidth overhead and CPU overhead by processing packet postcards twice (at hosts/switches and controller). EverFlow [156] tries to address the bandwidth overhead by letting the operator select packets for mirroring. OpenSketch [150], FlowRadar [101] and UnivMon [103] use sketches inside switches to support many per-flow queries with low bandwidth overhead, but require changes in switches. Relative to these systems, Trumpet sits at a different point in the design space: it can monitor every packet by using compute resources in servers, and incurs less bandwidth overhead by sending trigger satisfaction reports.

Ultimately, we see these classes of solutions co-existing in a data center: Trumpet can give quick and early warning of impending problems and has more visibility into host-level artifacts (*e.g.,* NIC drops), and other switch-based monitoring systems can be used for deep drill-down when packet or header inspection inside the network is required.

At the controller, there are many systems that can potentially integrate with Trumpet. Kinetic [90] allows controlling networks based on events that can be generated using Trumpet. Felix [26] generates matching filters for end-host measurement from high-level user queries and routing configuration.

Gigascope [41] introduces a stream database of traffic mirrored from the network, and supports a variety of traffic queries on the database. It also introduces a two stage approach of low-level queries and high-level queries to improve streaming efficiency [40]. Instead of collecting traffic into a central database, Trumpet distributes the monitoring and stream processing at all the end-hosts and aggregates the trigger information to capture network-wide events. Trumpet also introduces a new trigger repository that captures events in 10 ms intervals at line speed.

5.4 Network Rule Management

Policies and rules in the cloud: Recent proposals for new policies often propose customized systems to manage rules on either hypervisors [115, 23, 134, 121]) or switches [18, 5, 122]. vCRIB proposes an abstraction of a centralized rule repository for all the policies, frees these systems from the complexity inherent in the rule management, and handles heterogeneous resource constraints at devices while minimizing the traffic overhead.

Rule management in software-defined networks (SDNs): Recent work on SDNs provides rule repository abstractions and some rule management capabilities [22, 91, 152, 23]. vCRIB focuses on data centers, which are more dynamic, more sensitive to traffic overhead, and face heterogeneous resource constraints.

Distributed firewall: Distributed firewalls [15, 79], often used in enterprises, leverage a centralized manager to deploy security policies on edge machines. vCRIB manages more fine-grained rules on flows and

VMs for various policies including firewalls in the cloud. Rather than placing these rules at the edge, vCRIB places these rules taking into account the rule processing constraints, while minimizing traffic overhead.

Rule partition and placement solutions: The problem of partitioning and placing multi-dimensional data at different locations also appears in other contexts. Unlike traditional partitioning algorithms [143, 137, 62, 110, 93] which divide rules into partitions using a top-down approach, vCRIB uses *per-source partitions* to place the partitions close to the source with low traffic overhead. Compared with DIFANE [152], which *randomly* places *a single* partition of rules at each switch, vCRIB takes the *partitions-with-replication* approach to flexibly place *multiple* per-source partitions at one device. In preliminary work [111], we proposed an *offline* placement solution which works *only* for the TCAM resource model. The paper has a top-down heuristic partition-with-split algorithm which cannot limit the overhead of redirection rules and is not optimized for CPU-based resource model. Besides, having partitions with traffic from multiple sources requires complicated partition replication to minimize traffic overhead. In contrast, vCRIB uses fast per-source partition-with-replication algorithm which reduces TCAM-usage by leveraging similarity of partitions and restricts the resource usage of redirection by using limited number of equal shaped redirection rules. Our preliminary work used an unscalable DFS branch-and-bound approach to find a feasible solution and optimized the traffic in one step. vCRIB scales better using a two-phase solution where the first phase has an approximation bound in finding a feasible solution and the second can be run separately when the placement is still feasible.

Later work [85, 86, 88] tried to solve the rule placement problem with different approaches. N. Kang *et al.* [85] proposed a system to support the abstraction of a "one big switch" to place rules on hardware switches on the shortest path of flows. The proposed solution 1) allocates the portion of each switch to the rules necessary for handling the traffic from each source-destination path using linear programming; 2) for the policies of each path on each switch with an allocated capacity, tries to pack a partition of rules; 3) Step 2 may fail to cover all rules of a path because breaking a ruleset of overlapping rules may need more

space than their total. If it fails, we have to re-run the optimization in step 1 using higher headroom for the failed paths. Palette [86] also places the rules on hardware switches on the shortest path of flows and makes sure that every flow goes at least once through the whole policy. CacheFlow [88] offloads rules with the lowest traffic from hardware switches to software switches. CacheFlow can also update the placement when the ruleset updates. In comparison, vCRIB can place the rules on and off the shortest path of flows on both hardware and software switches and minimizes the detouring traffic overhead. vCRIB updates the placement per traffic changes.

Chapter 6

Conclusions

In this dissertation, we have explored how to provide *timely, accurate and scalable network management for data centers* through high-level abstractions at a centralized controller. To support the abstractions, we developed four systems that leverage algorithms at the controller to program switches and end-hosts. These algorithms can quickly fine tune the switches and end-hosts to keep high accuracy, drill down fast, and leverage device optimizations and network knowledge to scale. Here, we summarize the contribution of the systems from three different aspects: exploring different measurement primitives, leveraging measurement data to improve network control, and exploring end-host and switch capabilities.

6.1 Measurement Primitives

DREAM, SCREAM and Trumpet explore three different measurement primitives for Software-defined Measurement in data centers. Here, we summarize the advantages and disadvantages of each primitive and the contribution of each work.

Flow counting: The flow-based switches (*e.g.,* OpenFlow switches) allow operators to flexibly specify the flows to monitor based on different packet fields (*e.g.,* source and/or destination IP addresses), and *count* the number of bytes or packets for these flows. These switches are already available from various vendors (*e.g.,* HP, NEC) and used by cloud companies (*e.g.,* Google).

Flow-based switches rely on the controller to perform analysis on these counters. Based on the analysis of the counters, the controller then periodically adjusts the measurement rules at the switches, for example, to find source IP addresses that send the most traffic. Due to the delay of configuring and fetching the counters at switches, the controller can only achieve accurate analysis for measurements at large time-scales (*e.g.,* seconds). In addition, since the controller adjusts the rules based on the traffic history, this approach only works for traffic that changes at time-scales larger than the periodicity at which the controller adapts measurements. Finally, these flow-based counters are maintained in a power-hungry TCAM, and we can only use a limited number of TCAM entries for measurement. However, the accuracy of a measurement task is a function of its allocated memory on each switch, and this function changes with traffic properties. This forces operators to either provision for the worst case and lose scalability, or get measurements with low accuracy.

DREAM focuses on accuracy and scalability in network monitoring. It provides the abstraction of flow detection with minimum accuracy bound. Given the accuracy bound, it iteratively allocates TCAM entries to support many task instances of different types and parameters with an accuracy above the bound on switches with limited TCAM.

Hash-based counters: Hash-based counters implement *sketches*, summaries of streaming data for approximately answering a specific set of queries. They can be easily implemented in SRAM memory which is cheaper and more power-efficient than TCAM. Sketches are more expressive than flow counters because they can support more measurement tasks. Moreover, they can capture the right set of flow properties in the data plane without any iterative reconfiguration from the controller.

However, although these hash-based counters are feasible to implement with existing commodity switch components, they are still not implemented in today's switches. Also, since they still rely on sending *all* the counters to the controller for analysis, they can only support measurement analysis at relatively large time granularities. Finally, similar to flow-based counters, the accuracy of hash-based measurement tasks depends on their allocated memory and the properties of traffic seen on each switch.

SCREAM extends DREAM to switches that support hash-tables for monitoring. SCREAM has a new implementation of tasks and uses probabilistic methods to estimate accuracy, but it could reuse the iterative allocation algorithm in DREAM.

Programming: As many network events happen in sub-second time-scale (*e.g.,* burst losses and congestion), it is important to detect per packet and fine time-scale events. However, it is not scalable to send counters every 10ms from thousands of switches to a controller. Instead given the intent of the operator, we need to program the network to aggregate and filter measurement data before sending to the controller. A programmable primitive can look into per packet variables such as sequence numbers or ECN marks instead of just their size to answer much more queries than flow counters and hash-based counters.

Switches have limited capabilities (memory, CPU, reporting bandwidth and programmability) for detecting per packet or fine time-scale events. Instead, end-hosts are programmable and can check per packet conditions in millisecond time-scale. End-hosts, in turn, only see end-to-end properties and can only infer how switches inside the network behave (*e.g.,* drop/delay packets). We should also be cautious in using their CPU for network management as cloud users pay to run their workload on their CPU.

We developed Trumpet with an expressive event description language that permits novel events that are particularly relevant in data centers. Trumpet installs corresponding triggers at end-hosts where our algorithms and systems optimizations ensure a design that can process every packet at line-rate and report in 10ms without requiring extra CPU cores on a software switch.

6.2 Leveraging Monitoring for Network Control

Managing the network involves a loop of monitoring and control. The control logic looks into the measurement data (*e.g.,* flow sizes) and makes new decisions (*e.g.,* sending flows on different paths to control the link congestion). Leveraging the timely, accurate and scalable network monitoring systems (*e.g.,* DREAM, SCREAM and Trumpet), we can also make new control systems with these properties.

Our fourth system, vCRIB, leverages the monitoring capabilities to improve the network control scalability. vCRIB provides the abstraction of a scalable centralized rule repository for operators. Operators

just define their rules without worrying about which switch must host the rule. Then vCRIB compresses the rules and places them on switches or end-hosts with enough resources. The placement may impose bandwidth overhead because invalid flows that should have been dropped at the source may be dropped inside the network after one or two hops, and valid flows may be handled on a switch off the shortest path. Therefore, vCRIB relies on a monitoring system to provide the traffic volume for each rule, so that it can refine the rule placement to minimizing the traffic overhead.

In the future, we imagine that more control systems can leverage the monitoring data to facilitate the operators job. For example, network operators change network configuration frequently because of device faults/upgrades, new applications and variable traffic patterns. However, there are simply too many places that can induce error: the translation of policies to switch configurations; interaction of different configurations (sometimes at different switches); saving the configurations at network switches; and hardware faults at switches. Such errors may only affect a subset of packets but still damage the performance of applications. Unfortunately, they are not detectable by traditional measurement tools (e.g., SNMP) or current static configuration checkers. The only way to make sure network control is accurate is through validation. Future systems can automatically translate control policies and network-wide invariants to the right packet-level measurement tasks and validate them.

6.3 End-host vs. Switch Based SDN

SDN can use both hardware switches inside the network and software switches at end-hosts. Hardware switches support a high packet rate and have visibility inside the network, but they have limited resources and programmability. Software switches provide programmability and flexibility for applying complex network policies on traffic, but must use the precious CPU.

In this thesis, we explored the monitoring capabilities of hardware and software switches. On hardware switches, DREAM and SCREAM defined an abstraction for operators to get accurate measurements without worrying about hardware resource limitations. On end-hosts, we developed a monitoring system, Trumpet, that is tuned based on the computer architecture features for fast packet processing. In vCRIB,

we proposed a general algorithm to place control rules on both hardware and software switches. The algorithm uses a resource usage function to handle the differences between the two options. In vCRIB, we proposed the function for a de facto software switch called Open vSwitch.

Ultimately, we see hardware and software switch solutions co-existing in a data center: For monitoring, Trumpet can give quick and early warning of impending problems and has more visibility into host-level artifacts (*e.g.,* NIC drops), and other switch-based monitoring systems such as DREAM can be used for deep drill-down when packet or header inspection inside the network is required. For control, vCRIB already uses both hardware and software switches in an equal role. Other solutions also used hardware switches as a fast cache for rules that handle more traffic [88, 53].

References

[1] http://www.noxrepo.org/pox/about-pox.

[2] http://www.praxicom.com/2008/04/the-amazon-ec2.html.

[3] P. K. Agarwal, G. Cormode, Z. Huang, J. Phillips, Z. Wei, and K. Yi. "Mergeable Summaries". In: *PODS*. 2012.

[4] A. Aggarwal, S. Savage, and T. Anderson. "Understanding the Performance of TCP Pacing". In: *INFOCOM*. Vol. 3. 2000.

[5] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. "Hedera: Dynamic Flow Scheduling for Data Center Networks". In: *NSDI*. 2010.

[6] O. Alipourfard, M. Moshref, and M. Yu. "Re-evaluating Measurement Algorithms in Software". In: *HotNets*. 2015.

[7] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. "Data Center TCP (DCTCP)". In: *SIGCOMM*. 2011.

[8] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. "pFabric: Minimal Near-optimal Datacenter Transport". In: *SIGCOMM*. 2013.

[9] M. Allman, W. M. Eddy, and S. Ostermann. "Estimating Loss Rates with TCP". In: *SIGMETRICS Performance Evaluation Review* 31.3 (2003), pp. 12–24.

[10] *Amazon CloudWatch*. http://aws.amazon.com/cloudwatch/.

[11] *Amazon Web Services' Growth Unrelenting*. http://news.netcraft.com/archives/2013/05/20/amazon-web-services-growth-unrelenting.html.

[12] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska. "End-to-End Performance Isolation Through Virtual Datacenters". In: *OSDI*. 2014.

[13] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. "Workload Analysis of a Large-scale Key-value Store". In: *SIGMETRICS*. 2012.

[14] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea. "Enabling End-host Network Functions". In: *SIGCOMM*. 2015.

[15] S. M. Bellovin. "Distributed Firewalls". In: *;login:* (Nov. 1999), pp. 39–47.

[16] T. Benson, A. Anand, A. Akella, and M. Zhang. "MicroTE: Fine Grained Traffic Engineering for Data Centers". In: *CoNEXT*. 2011.

[17] T. Benson, A. Akella, and D. A. Maltz. “Network Traffic Characteristics of Data Centers in the Wild”. In: *IMC*. 2010.

[18] *Big Switch Networks*. http://www.bigswitch.com/.

[19] T. Bu, J. Cao, A. Chen, and P. P. C. Lee. “Sequential Hashing: A Flexible Approach for Unveiling Significant Patterns in High Speed Networks”. In: *Computer Networks* 54.18 (2010), pp. 3309–3326.

[20] *CAIDA Anonymized Internet Traces 2012*. http://www.caida.org/data/passive/passive_2012_dataset.xml.

[21] J. Cao, Y. Jin, A. Chen, T. Bu, and Z.-L. Zhang. “Identifying High Cardinality Internet Hosts”. In: *INFOCOM*. 2009.

[22] M. Casado, M. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker. “Rethinking Enterprise Network Control”. In: *Transactions on Networking* 17.4 (2009).

[23] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. “Virtualizing the Network Forwarding Plane”. In: *PRESTO*. 2010.

[24] M. Charikar, K. Chen, and M. Farach-Colton. “Finding Frequent Items in Data Streams”. In: *Automata, Languages and Programming*. Springer, 2002, pp. 693–703.

[25] C. Chekuri and S. Khanna. “A PTAS for the Multiple Knapsack Problem”. In: *SODA*. 2001.

[26] H. Chen, N. Foster, J. Silverman, M. Whittaker, B. Zhang, and R. Zhang. “Felix: Implementing Traffic Measurement on End Hosts Using Program Analysis”. In: *SOSR*. 2016.

[27] K. Chen, A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang, X. Wen, and Y. Chen. “OSA: An Optical Switching Architecture for Data Center Networks With Unprecedented Flexibility”. In: *NSDI*. 2012.

[28] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. “Understanding TCP Incast Throughput Collapse in Datacenter Networks”. In: *WREN*. 2009.

[29] H. Chernoff. “A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations”. In: *The Annals of Mathematical Statistics* 23.4 (1952).

[30] M. Chowdhury and I. Stoica. “Efficient Coflow Scheduling Without Prior Knowledge”. In: *SIGCOMM*. 2015.

[31] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba. “PayLess: A Low Cost Network Monitoring Framework for Software Defined Networks”. In: *IEEE/IFIP Network Operations and Management Symposium*. 2014.

[32] R. Cohen, L. Katzir, and D. Raz. “An Efficient Approximation for the Generalized Assignment Problem”. In: *Information Processing Letter* 100.4 (2006).

[33] G. Cormode. “Sketch Techniques for Approximate Query Processing”. In: *Synposes for Approximate Query Processing: Samples, Histograms, Wavelets and Sketches, Foundations and Trends in Databases*. Ed. by G. Cormode, M. Garofalakis, P. Haas, and C. Jermaine. NOW publishers, 2011.

[34] G. Cormode, R. Keralapura, and J. Ramimirtham. “Communication-Efficient Distributed Monitoring of Thresholded Counts”. In: *SIGMOD*. 2006.

[35] G Cormode and S Muthukrishnan. “Space Efficient Mining of Multigraph Streams”. In: *PODS*. 2005.

[36] G. Cormode and M. Hadjieleftheriou. “Finding Frequent Items in Data Streams”. In: *VLDB*. 2008.

[37] G. Cormode and S. Muthukrishnan. “An Improved Data Stream Summary: The Count-Min Sketch and its Applications”. In: *Journal of Algorithms* 55.1 (2005).

[38] G. Cormode and S Muthukrishnan. “Summarizing and Mining Skewed Data Streams”. In: *SIAM International Conference on Data Mining*. 2005.

[39] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. “Finding Hierarchical Heavy Hitters in Data Streams”. In: *VLDB*. 2003.

[40] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. “Holistic UDAFs at Streaming Speeds”. In: *SIGMOD*. 2004.

[41] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. “Gigascope: a Stream Database for Network Applications”. In: *SIGMOD*. 2003.

[42] A. Curtis, J. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. “DevoFlow: Scaling Flow Management for High-Performance Networks”. In: *SIGCOMM*. 2011.

[43] M. Dobrescu, K. Argyraki, G. Iannaccone, M. Manesh, and S. Ratnasamy. “Controlling Parallelism in a Multicore Software Router”. In: *PRESTO*. 2010.

[44] *DPDK*. http://dpdk.org.

[45] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. “Maglev: A Fast and Reliable Software Network Load Balancer”. In: *NSDI*. 2016.

[46] C. Estan and G. Varghese. “New Directions in Traffic Measurement and Accounting”. In: *SIGCOMM Computer Communication Review* 32.4 (2002), pp. 323–336.

[47] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. “Deriving Traffic Demands for Operational IP Networks: Methodology and Experience”. In: *Transactions on Networking* 9.3 (2001).

[48] D. Firestone. “SmartNIC: FPGA Innovation in OCS Servers for Microsoft Azure”. In: *OCP U.S. Summit*. 2016.

[49] P. Flajolet and G. N. Martin. “Probabilistic Counting Algorithms for Data Base Applications”. In: *Journal of computer and system sciences* 31.2 (1985), pp. 182–209.

[50] *Floodlight controller*. http://www.projectfloodlight.org/floodlight/.

[51] É. Fusy, G Olivier, and F. Meunier. “Hyperloglog: The Analysis of a Near-optimal Cardinality Estimation Algorithm”. In: *Analysis of Algorithms(AofA)*. 2007.

[52] M. Gabel, A. Schuster, and D. Keren. “Communication-Efficient Distributed Variance Monitoring and Outlier Detection for Multivariate Time Series”. In: *IPDPS*. 2014.

[53] R. Gandhi, H. Liu, Y. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. “Duet: Cloud Scale Load Balancing with Hardware and Software”. In: *SIGCOMM*. 2014.

[54] R. Gandhi, Y. C. Hu, C.-k. Koh, H. H. Liu, and M. Zhang. “Rubik: Unlocking the Power of Locality and End-Point Flexibility in Cloud Scale Load Balancing”. In: *ATC*. 2015.

[55] M. Ghasemi, T. Benson, and J. Rexford. *RINC: Real-Time Inference-based Network Diagnosis in the Cloud*. Tech. rep. Technical Report TR-975-14, Princeton University, 2015.

[56] M. Ghobadi and Y. Ganjali. “TCP Pacing in Data Center Networks”. In: *High-Performance Interconnects (HOTI)*. 2013.

[57] P. Gill, N. Jain, and N. Nagappan. “Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications”. In: *SIGCOMM Computer Communication Review* 41.4 (2011).

[58] *Google Compute Engine Incident 15041*. https://status.cloud.google.com/incident/compute/15041. 2015.

[59] *Google Compute Engine Incident 16007*. https://status.cloud.google.com/incident/compute/16007. 2016.

[60] *Google Compute Engine Service Level Agreement (SLA)*. https://cloud.google.com/compute/sla. 2016.

[61] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. “Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis”. In: *SIGCOMM*. 2015.

[62] P. Gupta and N. McKeown. “Packet Classification using Hierarchical Intelligent Cuttings”. In: *Hot Interconnects VII*. 1999.

[63] M. Hadjieleftheriou, J. W. Byers, and J. Kollios. *Robust Sketching and Aggregation of Distributed Data Streams*. Tech. rep. Boston University Computer Science Department, 2005.

[64] M. Hajiaghayi, K. Jain, K. Konwar, L. Lau, I. Mandoiu, A. Russell, A. Shvartsman, and V. Vazirani. “The minimum k-colored subgraph problem in haplotyping and DNA primer selection”. In: *IWBRA*. 2006.

[65] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. *SoftNIC: A Software NIC to Augment Hardware*. Tech. rep. UCB/EECS-2015-155. http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html. EECS Department, University of California, Berkeley, 2015.

[66] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. “I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks”. In: *NSDI*. 2014.

[67] P. Hardin. *Amazon.com Announces Fourth Quarter Sales up 22% to $35.7 Billion*. http://phx.corporate-ir.net/phoenix.zhtml?c=97664&p=irol-newsArticle&ID=2133281. Jan. 2016.

[68] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Bannerjee, and N. McKeown. “ElasticTree: Saving Energy in Data Center Networks”. In: *NSDI*. 2010.

[69] S. Heule, M. Nunkesser, and A. Hall. “HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm”. In: *International Conference on Extending Database Technology*. 2013.

[70] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. “Practical Declarative Network Management”. In: *WREN*. 2009.

[71] T. Hoff. *Latency Is Everywhere And It Costs You Sales - How To Crush It*. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it. 2009.

[72] Y.-J. Hong and M. Thottethodi. “Understanding and Mitigating the Impact of Load Imbalance in the Memory Caching Tier”. In: *SOCC*. 2013.

[73] *HP 5120 EI Switch Series: QuickSpecs*. http://h18000.www1.hp.com/products/quickspecs/13850_na/13850_na.PDF.

[74] L. Hu, K. Schwan, A. Gulati, J. Zhang, and C. Wang. “Net-cohort: Detecting and Managing VM Ensembles in Virtualized Data Centers”. In: *ICAC*. 2012.

[75] D. Y. Huang, K. Yocum, and A. C. Snoeren. “High-fidelity Switch Models for Software-defined Network Emulation”. In: *HotSDN*. 2013.

[76] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse. “Characterizing Load Imbalance in Real-World Networked Caches”. In: *HotNets*. 2014.

[77] “IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems”. In: *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)* (2008), pp. 1–269.

[78] *Intel Data Direct I/O Technology*. http://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html.

[79] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. “Implementing a Distributed Firewall”. In: *CCS*. 2000.

[80] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hlzle, S. Stuart, and A. Vahdat. “B4: Experience with a globally-deployed software defined WAN”. In: *SIGCOMM*. 2013.

[81] J. Jiang, T. Lan, S. Ha, M. Chen, and M. Chiang. “Joint VM Placement and Routing for Data Center Traffic Engineering”. In: *INFOCOM*. 2012.

[82] L. Jose, M. Yu, and J. Rexford. “Online Measurement of Large Traffic Aggregates on Commodity Switches”. In: *Hot-ICE*. 2011.

[83] E. G. C. Jr., M. R. Carey, and D. S. Johnson. “Approximation Algorithms for NP-hard Problems”. In: Boston, MA, USA: PWS Publishing Co., 1997. Chap. Approximation Algorithms for Bin Packing: A Survey.

[84] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. “The Nature of Datacenter Traffic: Measurements and Analysis”. In: *IMC*. 2009.

[85] N. Kang, Z. Liu, J. Rexford, and D. Walker. “Optimizing the “One Big Switch” Abstraction in Software-Defined Networks”. In: *CoNEXT*. 2013.

[86] Y. Kanizo, D. Hay, and I. Keslassy. “Palette: Distributing Tables in Software-Defined Networks”. In: *INFOCOM*. 2013.

[87] R. Kapoor, A. C. Snoeren, G. M. Voelker, and G. Porter. “Bullet Trains: A Study of Nic Burst Behavior at Microsecond Timescales”. In: *CoNEXT*. 2013.

[88] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. “Cacheflow: Dependency-aware Rule-caching for Software-defined Networks”. In: *SOSR*. 2016.

[89] F. Khan, N. Hosein, C.-N. Chuah, and S. Ghiasi. “Streaming Solutions for Fine-Grained Network Traffic Measurements and Analysis”. In: *ANCS*. 2011.

[90] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. “Kinetic: Verifiable Dynamic Network Control”. In: *NSDI*. 2015.

[91] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. “Onix: A Distributed Control Platform for Large-scale Production Networks”. In: *OSDI*. 2010.

[92] F. Korn, S Muthukrishnan, and Y. Wu. “Modeling Skew in Data Streams”. In: *SIGMOD*. 2006.

[93] V. Kriakov, A. Delis, and G. Kollios. “Management of Highly Dynamic Multidimensional Data in a Cluster of Workstations”. In: *Advances in Database Technology-EDBT* (2004).

[94] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. “Sketch-based Change Detection: Methods, Evaluation, and Applications”. In: *IMC*. 2003.

[95] A. Kumar, M. Sung, J. J. Xu, and J. Wang. “Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Size Distribution”. In: *SIGMETRICS*. 2004.

[96] A. Kumar, S. Jain, U. Naik, N. K. Anand Raghuraman, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat. “BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing”. In: *SIGCOMM*. 2015.

[97] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang. “Data Streaming Algorithms for Estimating Entropy of Network Traffic”. In: *SIGMETRICS/Performance*. 2006.

[98] G. M. Lee, H. Liu, Y. Yoon, and Y. Zhang. “Improving Sketch Reconstruction Accuracy Using Linear Least Squares Method”. In: *IMC*. 2005.

[99] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, and P. Cheng. “ClickNP: Highly Flexible and High-performance Network Processing with Reconfigurable Hardware”. In: *SIGCOMM*. 2016.

[100] P. Li and C.-H. Zhang. “A New Algorithm for Compressed Counting with Applications in Shannon Entropy Estimation in Dynamic Data”. In: *COLT*. 2011.

[101] Y. Li, R. Miao, C. Kim, and M. Yu. “FlowRadar: A Better NetFlow for Data Centers”. In: *NSDI*. 2016.

[102] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. “MICA: A Holistic Approach to Fast In-memory Key-value Storage”. In: *NSDI*. 2014.

[103] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. “One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon”. In: *SIGCOMM*. 2016.

[104] H. Lotfi. *Sensors and Data Analytics in Large Data Center Networks*. https://ripe72.ripe.net/presentations/57-RIPE72-Google.pdf. 2016.

[105] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani. “Counter braids: A Novel Counter Architecture for Per-flow Measurement”. In: *SIGMETRICS Performance Evaluation Review* 36.1 (2008), pp. 121–132.

[106] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. “OpenFlow: Enabling Innovation in Campus Networks”. In: *SIGCOMM Computer Communication Review* 38.2 (2008).

[107] S. Meng, A. K. Iyengar, I. M. Rouvellou, and L. Liu. “Volley: Violation Likelihood Based State Monitoring for Dataceners”. In: *ICDCS* (2013).

[108] R. Miao, R. Potharaju, M. Yu, and N. Jain. “The Dark Menace: Characterizing Network-based Attacks in the Cloud”. In: *IMC*. 2015.

[109] M. Mitzenmacher, T. Steinke, and J. Thaler. “Hierarchical Heavy Hitters with the Space Saving Algorithm”. In: *arXiv:1102.5540* (2011).

[110] A. Mondal, M. Kitsuregawa, B. C. Ooi, and K. L. Tan. “R-tree-based Data Migration and Self-Tuning Strategies in Shared-Nothing Spatial Databases”. In: *GIS*. 2001.

[111] M. Moshref, M. Yu, A. Sharma, and R. Govindan. “vCRIB: Virtualized Rule Management in the Cloud”. In: *HotCloud*. 2012.

[112] M. Moshref, M. Yu, and R. Govindan. “Resource/Accuracy Tradeoffs in Software-Defined Measurement”. In: *HotSDN*. 2013.

[113] J. Mudigonda, P. Yalagandula, J. Mogul, and B. Stiekes. “NetLord: A Scalable Multi-Tenant Network Architecture for Virtualized Datacenters”. In: *SIGCOMM*. 2011.

[114] *n2disk: A Multi-Gigabit Network Traffic Recorder with Indexing Capabilities*. http://www.ntop.org/products/traffic-recording-replay/n2disk/.

[115] *Open vSwitch*. http://openvswitch.org/.

[116] N. Parlante. *Linked List Basics*. http://cslibrary.stanford.edu/103/LinkedListBasics.pdf. 2001.

[117] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. “Ananta: Cloud Scale Load Balancing”. In: *SIGCOMM*. 2013.

[118] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. “Fastpass: A Centralized “Zero-queue” Datacenter Network”. In: *SIGCOMM*. 2014.

[119] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. “The Design and Implementation of Open vSwitch”. In: *NSDI*. 2015.

[120] *Pica8 P-3290 switch*. http://www.pica8.com/documents/pica8-datasheet-48x1gbe-p3290-p3295.pdf.

[121] L. Popa, M. Yu, S. Y. Ko, I. Stoica, and S. Ratnasamy. “CloudPolice: Taking Access Control out of the Network”. In: *HotNets*. 2010.

[122] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. “FairCloud: Sharing The Network In Cloud Computing”. In: *SIGCOMM*. 2012.

[123] R. Potharaju and N. Jain. “Demystifying the Dark Side of the Middle: A Field Study of Middlebox Failures in Datacenters”. In: *IMC*. 2013.

[124] *Private conversation with Amazon*.

[125] *Private conversation with RackSpace operators*.

[126] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. “Planck: Millisecond-scale Monitoring and Control for Commodity Networks”. In: *SIGCOMM*. 2014.

[127] A. Roy, H. Zeng, J. Bagga, G. M. Porter, and A. C. Snoeren. “Inside the Social Network’s (Datacenter) Network”. In: *SIGCOMM*. 2015.

[128] S. Rupley. *Eyeing the Cloud, VMware Looks to Double Down On Virtualization Efficiency*. http://gigaom.com/2010/01/27/eyeing-the-cloud-vmware-looks-to-double-down-on-virtualization-efficiency. 2010.

[129] R. Schweller, A. Gupta, E. Parsons, and Y. Chen. “Reversible Sketches for Efficient and Accurate Change Detection over Network Data Streams”. In: *IMC*. 2004.

[130] V. Sekar, M. K. Reiter, and H. Zhang. “Revisiting the Case for a Minimalist Approach for Network Flow Monitoring”. In: *IMC*. 2010.

[131] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen. “CSAMP: A System for Network-Wide Flow Monitoring”. In: *NSDI*. 2008.

[132] V. Sekar, N. G. Duffield, O. Spatscheck, J. E. van der Merwe, and H. Zhang. “LADS: Large-scale Automated DDoS Detection System.” In: *ATC*. 2006.

[133] I. Sharfman, A. Schuster, and D. Keren. “A Geometric Approach to Monitoring Threshold Functions over Distributed Data Streams”. In: *Transaction on Database Systems* 32.4 (Nov. 2007).

[134] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. “Sharing the Datacenter Networks”. In: *NSDI*. 2011.

[135] M. Sindelar, R. K. Sitaram, and P. Shenoy. “Sharing-Aware Algorithms for Virtual Machine Colocation”. In: *SPAA*. 2011.

[136] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hoelzle,

S. Stuart, and A. Vahdat. “Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Googles Datacenter Network”. In: *SIGCOMM*. 2015.

[137] S. Singh, F. Baboescu, G. Varghese, and J. Wang. “Packet Classification Using Multidimensional Cutting”. In: *SIGCOMM*. 2003.

[138] V. Srinivasan, S. Suri, and G. Varghese. “Packet Classification Using Tuple Space Search”. In: *SIGCOMM*. 1999.

[139] *Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region*. http://aws.amazon.com/message/65648. 2011.

[140] P. Sun, M. Yu, M. J. Freedman, J. Rexford, and D. Walker. “HONE: Joint Host-Network Traffic Management in Software-Defined Networks”. In: *Journal of Network and Systems Management* 23.2 (2015), pp. 374–399.

[141] D. E Taylor and J. S Turner. “ClassBench: A Packet Classification Benchmark”. In: *Transactions on Networking* 15.3 (2007).

[142] A. Vahdat. *Enter the Andromeda zone - Google Cloud Platform’s Latest Networking Stack*. http://goo.gl/smN6W0.

[143] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. “Efficuts: Optimizing Packet Classification for Memory and Throughput”. In: *SIGCOMM*. 2010.

[144] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., 2001.

[145] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum. “New Streaming Algorithms for Fast Detection of Superspreaders”. In: *NDSS*. 2005.

[146] *Virtual networking technologies at the server-network edge*. http://h20000.www2.hp.com/bc/docs/support/SupportManual/c02044591/c02044591.pdf.

[147] A. Voellmy, H. Kim, and N. Feamster. “Procera: A Language for High-Level Reactive Network Control”. In: *HotSDN*. 2010.

[148] M. Wang, B. Li, and Z. Li. “sFlow: Towards Resource-efficient and Agile Service Federation in Service Overlay Networks”. In: *International Conference on Distributed Computing Systems*. 2004.

[149] W. Wu, K. He, and A. Akella. “PerfSight: Performance Diagnosis for Software Dataplanes”. In: *IMC*. 2015.

[150] M. Yu, L. Jose, and R. Miao. “Software Defined Traffic Measurement with OpenSketch”. In: *NSDI*. 2013.

[151] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. “Profiling Network Performance for Multi-tier Data Center Applications”. In: *NSDI*. 2011.

[152] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. “Scalable Flow-Based Networking with DIFANE”. In: *SIGCOMM*. 2010.

[153] L. Yuan, C.-N. Chuah, and P. Mohapatra. “ProgME: Towards Programmable Network MEasurement”. In: *Transactions on Networking* 19.1 (2011).

[154] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund. “Online Identification of Hierarchical Heavy Hitters: Algorithms, Evaluation, and Applications”. In: *IMC*. 2004.

[155] Y. Zhang. “An Adaptive Flow Counting Method for Anomaly Detection in SDN”. In: *CoNEXT*. 2013.

[156] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, H. Zheng, and B. Zhao. “Packet-Level Telemetry in Large Datacenter Networks”. In: *SIGCOMM*. 2015.