Masoud Moshref Javadi

www.masoudmoshref.com
moshrefj@usc.edu
2139104140

941 W. 37th Place, SAL 224
Los Angeles, CA, 90089

Education

University of Southern California, Los Angeles
Ph.D. in Computer Engineering — Expected May 2016
Advisor: Ramesh Govindan & Minlan Yu
Thesis: Scalable, timely and accurate network management systems for datacenters

Sharif University of Technology, Tehran, Iran
MSc in Information Technology Engineering — Jan 2010
Advisor: Hamid R. Rabiee
Thesis: LayeredCast: A hybrid peer-to-peer architecture for real-time layered video streaming over Internet

Sharif University of Technology, Tehran, Iran
BSc in Information Technology Engineering — Sep 2007
Thesis: MobiSim: Design and implementation of a mobility model simulator in mobile ad-hoc networks

Research Interests

I develop networked systems based on Software-defined Networking (SDN) paradigm to achieve scalable, timely and accurate network management. I am interested in measuring traffic and controlling switches and middleboxes inside a network and at end-hosts.

Awards and Accomplishments

Google US/Canada Ph.D. fellowship in Computer Networking	2015
Student poster contest winner in NANOG on the Road - Los Angeles	2014
USC Provost Ph.D. fellowship recipient	2010
1st GPA among IT Engineering students & 2nd among CE 80+ (HW, SW, AI, IT) students in MSc	2010
1st GPA among IT Engineering students & 4th among 110+ CE (HW, SW, IT) students in BSc	2007

Publications

Software Defined Networking

1. *M. Moshref,* M. Yu, R. Govindan, A. Vahdat, SCREAM: Sketch Resource Allocation for Software-defined Measurement, **CoNEXT**, 2015
2. O. Alipourfard, *M. Moshref,* M. Yu, Re-evaluating Measurement Algorithms in Software, **HotNets**, 2015
3. *M. Moshref,* M. Yu, R. Govindan, A. Vahdat, DREAM: Dynamic Resource Allocation for Software-defined Measurement, **SIGCOMM**, 2014
4. *M. Moshref,* A. Bhargava, A. Gupta, M. Yu, R. Govindan, Flow-level State Transition as a New Switch Primitive for SDN, **HotSDN**, 2014
5. *M. Moshref,* M. Yu, A. Sharma, R. Govindan, Scalable Rule Management for Data Centers, **NSDI**, 2013
6. *M. Moshref,* M. Yu, R. Govindan, Resource/Accuracy Tradeoffs in Software-Defined Measurement, **HotSDN**, 2013
7. *M. Moshref,* M. Yu, A. Sharma, R. Govindan, vCRIB: Virtualized Rule Management in the Cloud, **HotCloud**, 2012

P2P Video Streaming

8. *M. Moshref*, R. Motamedi, H. Rabiee, M. Khansari, LayeredCast - A Hybrid Peer-to-Peer Live Layered Video Streaming Protocol, **International Symposium on Telecommunication (IST)**, 2010
9. *M. Moshref*, H. Rabiee, S. Nari, Challenges and Solutions in Peer-to-peer Live Video Streaming, Tech. rep. Computer Engineering, Sharif University of Technology, 2009

Mobile Ad-hoc Networks

10. M. Mousavi, H. Rabiee, *M. Moshref*, A. Dabirmoghaddam, Mobility Pattern Recognition in Mobile Ad-Hoc Networks, ACM **International Conference on Mobile Technology, Applications and Systems**, 2007
11. M. Mousavi, H. Rabiee, *M. Moshref*, A. Dabirmoghaddam, Model Based Adaptive Mobility Prediction in Mobile Ad-Hoc Networks, IEEE **WiCOM**, 2007
12. M. Mousavi, H. Rabiee, *M. Moshref*, A. Dabirmoghaddam, Mobility Aware Distributed Topology Control in Mobile Ad-hoc Networks with Model Based Adaptive Mobility Prediction, IEEE **WiMob**, 2007
13. M. Mousavi, H. Rabiee, *M. Moshref*, A. Dabirmoghaddam, MobiSim: A Framework for Simulation of Mobility Models in Mobile Ad-Hoc Networks, IEEE **WiMob**, 2007

Others

14. *M. Moshref*, A. Sharma, H. Madhyastha, L. Golubchik, R. Govindan, MRM: Delivering Predictability and Service Differentiation in Shared Compute Clusters, Tech. rep. Computer Science, USC, 2013
15. A. Gharakhani, *M. Moshref*, Evaluating Iran's Progress in ICT Sector Using e-Readiness Index, A System Dynamics Approach, **International System Dynamics Conference**, 2007

Work in Progress

16. *M. Moshref*, M. Yu, R. Govindan, A. Vahdat, Trumpet: Timely and Precise Triggers in Data Centers, Submitted to SIGCOMM, 2016
17. *M. Moshref*, A. Sharma, H. Madhyastha, L. Golubchik, R. Govindan, Paradise: Enabling Differentiated Service with Predictable Completion Time in Shared Compute Clusters, In preparation
18. S. Zhu, J. Bi, C. Sun, H. Chen, Z. Zheng, H. Hu, M. Yu, *M. Moshref*, C. Wu, C. Zhang, HiPPA: a High-Performance and Programmable Architecture for Network Function Virtualization, In preparation

Refereed Posters

19. *M. Moshref*, A. Bhargava, A. Gupta, M. Yu, R. Govindan, Flow-level State Transition as a New Switch Primitive for SDN, **SIGCOMM**, 2014
20. *M. Moshref*, M. Yu, R. Govindan, A. Vahdat, DREAM: Dynamic Resource Allocation for Software-defined Measurement, **NANOG on the road**, 2014
21. *M. Moshref*, M. Yu, R. Govindan, Software Defined Measurement for Data Centers, **NSDI**, 2013
22. *M. Moshref*, A. Sharma, H. Madhyastha, L. Golubchik, R. Govindan, MRM: Delivering Predictability and Service Differentiation in Shared Compute Clusters, **SoCC**, 2013

Teaching and Advising Experience

Teaching Assistant

University of Southern California: Introduction to Operating Systems (CSCI350) 2014
Sharif University of Technology: Object Oriented System Design (2 semesters), Cryptography Theory, Multimedia Systems, Web Programming, Information Technology Project Management, Computer Workshop (3 semesters) 2005-2009

Guest Lectures

Computer Communications (CSCI551), University of Southern California 2015
Software Defined Networking (CSCI694b), University of Southern California 2014

Mentoring

Omid Alipourfard (USC, CS Ph.D.): Optimizing Network Measurement in Software Switches	2015
Harsh Patel (USC, CS MSc): Virtualizing Rate-limiters in SDN	2014
Aditya Kamath (Viterbi-India program, Undergraduate): Implementing Sketches in Software Switches	2014
Adhip Gupta (USC, CS MSc): Flow-level State Transition as a New Switch Primitive for SDN	2014
Apoorv Bhargava (USC, CS MSc): Flow-level State Transition as a New Switch Primitive for SDN	2014

Professional Experience

Research Assistant, University of Southern California (Networked Systems Laboratory), Los Angeles — Spring 2011 - Present

- *Resource allocation for accurate network measurement:* I explored the tradeoff space of resource usage versus accuracy for different network measurement primitives [6]. I quantified these tradeoffs in the context of hierarchical heavy hitter detection for two primitives: flow-counters (TCAMs) and hash-based counters (sketches). In addition, I proposed an SDN controller framework to allocate network resources for concurrent measurement tasks while guaranteeing their accuracy for flow-counters (DREAM [3]) and sketches (SCREAM [1]). I designed a fast and stable allocation control loop and algorithms based on probabilistic bounds for estimating instantaneous measurement accuracy as a feedback for the control loop. I released a prototype of DREAM for OpenFlow switches and Floodlight framework.
- *Timely accurate measurement on software switches:* I quantified the performance and accuracy effects of different memory-saving algorithms for network measurement on software switches based on a Click + DPDK implementation [2]. Then I developed a timely and accurate event detection for datacenters, Trumpet [16]: I designed the network-wide event definition language and the controller system to support the events using triggers at end-hosts. I developed a fast packet processing framework at end-hosts on top of DPDK to run expressive measurement codes for triggers.
- *Scalable rule management:* I proposed vCRIB [5, 7] as a scalable way to place networking rules dynamically in a datacenter having devices with limited rule capacity. I proposed an approximation algorithm with a proved bound plus an online refinement algorithm to solve a novel version of bin-packing optimization problem for rule placement.
- *State machines for programming switches:* I proposed a switch programming primitive to support state machines inside switches. I led two graduate students in developing FAST [4], the controller framework and the switch architecture (using components already available in commodity switches) to support the new primitive.
- *Differentiated service with predictable completion time for Map-Reduce:* I proposed MRM [14, 22], a management system for Map-Reduce framework. MRM provides service differentiation to delay-sensitive jobs along with predictable finish times for all jobs, delay-sensitive as well as delay-tolerant, in enterprise data analytics clusters shared by multiple users. I used Gaussian Processes to predict tasks duration. My implementation on a 40-node Hadoop cluster and simulations show MRM superior performance vs. priority queues and FCFS queues.

Graduate Student Researcher, Sharif University of Technology (Digital Media Laboratory), Tehran, Iran — Fall 2006 - Winter 2010

- Developed MobiSim [13], a mobility trace generator, evaluator, and analyzer framework. I Maintained the code on SourceForge.com and it had over 5k downloads and 60 citations. I also used MobiSim to improve distance prediction in Ad-hoc networks [11], propose new power-aware topology management protocol [12], and test a new mobility pattern recognition algorithm [10].
- Proposed LayeredCast [8], a hybrid live layered video streaming protocol on peer-to-peer networks. LayeredCast pushes the basic layer of video frames in a tree topology to guarantee smooth video play and pulls the enhanced layers in a mesh topology to improve video quality wherever extra bandwidth is available. I evaluated LayeredCast in an innovative simulation framework on top of OMNET++.

J2EE Developer, System Group (Sepehre Mehr), Tehran, Iran Summer 2007
Sepehre Mehr was a web-based educational software development startup company bought by System Group, one of the largest software companies in Iran.

- Developed the exam module in their online education system using J2EE framework (Hibernate and Jboss)
- Performed the analyze phase of a faculty assessment system in the medical branch during one week on-site interviews with administrative staffs in Shiraz University

Flash Application Programmer, Simin Negar, Tehran, Iran Winter 2007
Simin Negar is a startup company established by former AICTC employees. I got a contract to develop an object oriented Flash quiz generator application with 7 templates which connects to a CMS using an XML interface. The templates were used in exams for almost 20 courses.

CMS Supporter, AICTC, Tehran, Iran 2004 - 2006
AICTC is a startup company related to Sharif University of Technology providing consulting services and developing and localizing open-source portals.

- Produced e-learning contents as the standard model for out-sourced companies that create SCORM-based online university courses
- Developed interfaces for uPortal channels using HTML, XSD and CSS
- Took the initiative to learn HyperContent CMS which uses an XML based form generator and XSD based templates to generate online courses
- Set up AICTC CMS group, training the members and presenting on-site workshops for customer's IT staffs

Presentations

- DREAM: Dynamic Resource Allocation for Software-defined Measurement, **SIGCOMM & Samsung Electronics**, 2014
- Flow-level State Transition as a New Switch Primitive for SDN, **HotSDN**, 2014
- Resource Virtualization for Software Defined Networks: **NEC Labs & RSRG group in Caltech & Center for Networked Systems in UCSD & CS department in Princeton University & CS201 Seminar course in UCLA**, 2014
- Scalable Rule Management for Data Centers, **NSDI & Cognizant Technologies**, 2013
- vCRIB: Virtualized Rule Management in the Cloud, **HotCloud**, 2012

Academic Service

Journal Review: Transaction on Networking, Transaction on Communication, Transactions on Dependable and Secure Computing, Transactions on Network and Service Management, Transactions on Parallel and Distributed Systems, Communications Letters, Wireless Networks (WINE), Computer Communication Review

External Reviewer: PAM'15, Performance'15, ANCS'14, HotCloud'14, HotSDN'14, CoNEXT'14

Co-chair for NSDI Shadow PC (2013, 2014, 2015)
We selected a representative set of NSDI submitted papers to review. We set-up HotCRP similar to the original conference and hold PC meetings to practice the decision making process. At the end, we shared our reviews with authors and compared our decisions and reviews with actual PC's for a subset of papers.

References

Ramesh Govindan
Professor
Department of Computer Science
University of Southern California
ramesh@usc.edu

Minlan Yu
Assistant Professor
Department of Computer Science
University of Southern California
minlanyu@usc.edu

Amin Vahdat
Google Fellow
Google
vahdat@google.com

Harsha V. Madhyastha
Assistant Professor
Computer Science and Engineering Division
University of Michigan
harshavm@umich.edu

Abhishek B. Sharma
Data Scientist
Fidelis Cybersecurity
absharma@gmail.com

Masoud Moshref Javadi - Research Statement

I am a systems researcher in the field of computer networking. I propose efficient **algorithms** (approximation, greedy, distributed, scheduling), define **abstractions** (hide complexities, are general, expose tradeoffs) and develop **systems** (fast, distributed, at switches and end-hosts) to solve real-world networking problems. In particular, I am interested in designing and implementing systems for operators to manage networks more efficiently. My research is motivated by the fact that we need to achieve high levels of performance and availability in networks with huge scale, but our solutions for managing networks are primitive. Thus, I build **scalable, timely and accurate network management systems**.

New networks especially datacenter networks have tight requirements in scale, timeliness and accuracy. Datacenter networks have huge scale: They connect hundreds of thousands of servers inside a datacenter using thousands of network switches that must work in coordination. The bandwidth demand on these networks is in the millions of Gbps and is doubling every 12 months. Thousands of users concurrently use the network for applications with diverse requirements. Networks must work accurately: An inaccurate network control can cause congestion and packet losses that damage the performance of prevalent short connections. A challenge is that a device failure is a common event because such networks are usually built upon commodity devices. Networks must react fast: The end-to-end latency requirement is in microseconds. Traffic patterns among servers are unpredictable and change rapidly in tens of milliseconds.

However, the management tools for operators are too limited. Network operators are involved in every aspect of datacenters from design and deployment to operation, 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. Even after detecting the events, the control system that reacts to the events must scale to thousands of switches and hundreds of application requirements. Slow control systems cannot react in milliseconds to be effective for the variable network traffic patterns and to hide the effect of failures and resource congestions from user applications. Moreover, current inaccurate control systems let errors and human mistakes go through and cause inaccessible services for hours even in big companies like Google and Time Warner cable.

Dissertation Work

Software-Defined Networking (SDN), a new trend in networking, distinguishes two layers in networks: the control plane running at a logically centralized server 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 the applications running on the controller. The 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.

Network management involves both measurement and control since controlling a network is not possible without observing the events in the network. In my dissertation, I built both accurate and timely measurement systems and timely and scalable network control systems. I worked on systems that run on hardware switches and end-hosts. For hardware switches, I defined general abstractions for operators that free them from the complexities imposed by hardware resource limitations. For end-hosts, I studied the effect of computer architecture on packet processing and used the result to improve measurement and control systems.

Accurate, yet resource efficient, measurement: 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 datacenters 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 tradeoffs. I qualitatively and quantitatively explored the tradeoff space of resource usage versus accuracy for three different primitives [6]: (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 can express many more measurement task types with higher accuracy and use cheap SRAM 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 tradeoffs. Focusing on flow counters and hash-based counters, I 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.

I developed DREAM [3] for flow counters and SCREAM [2] for hash-based counters to provide operators with the abstraction of guaranteed measurement accuracy that hides resource limits from operators. The 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 an estimated accuracy feedback from each task that enables iterative allocations. I 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 analyse their output without knowing the ground-truth. (c) A scalable resource allocation algorithm that converges fast and is stable.

I built a prototype of DREAM on OpenFlow switches with three network-wide measurement task types (heavy hitter, hierarchical heavy hitter and change detection), and I showed that DREAM can support 30% more concurrent tasks with up to 80% more accurate measurements than fixed allocation. For SCREAM, I 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.

Scalable, timely and accurate measurement: With growing concerns of the cost, management difficulty and expressiveness of hardware network switches, there is a new trend of moving measurement and other network functions to software switches at end-hosts. I implemented a subset of

measurement algorithms in software to re-evaluate their accuracy and performance for traffic traces with different properties [1]. I observed that modern multicore computer architectures have significantly increased their cache efficiency and cache size to the extent that it can fit the working set of many measurement tasks with a usually skewed access pattern. As a result, complex algorithms that trade off memory for CPU and access many memory entries to compress the measurement data structure are harmful to packet processing performance. Then I developed an expressive scalable measurement system on servers, Trumpet [8], 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.

Scalable control: 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. I 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 [5,7] 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. I 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. I solved this as a new bin-packing problem by a novel approximation algorithm with a proved bound. I modeled the resource usage of rule processing at end-hosts and generalized the solution to both hardware switches and end-hosts. 3) Traffic patterns change over time. vCRIB minimizes traffic overhead using an online greedy algorithm that adaptively changes the location of partitions in the face of traffic changes and VM migration. I demonstrate that vCRIB can find feasible rule placements with less than 10% traffic overhead when traffic-optimal rule placement is infeasible.

Timely control: Current SDN interface, OpenFlow, requires the centralized controller to be involved actively in any stateful decision even though the event and action happen on the same switch. This adds 10s of ms delay on packet processing and huge computation overhead on the controller, which makes it hard for operators to implement middlebox functionalities in SDN. I proposed a new control primitive in SDN, flow-level state machines, that enables the controller to proactively program switches to run dynamic actions based on local information without involving the controller. I developed FAST [4], the controller and the switch architecture using components already available in commodity switches to support the new primitive. This motivated a collaboration with Tsinghua University on HiPPA [9] project that dynamically chains state machines in hardware and software in order to improve the performance of software-based middleboxes and the flexibility of hardware-based ones.

Future Work

My dissertation work focuses more on measurement, but once a controller can observe network events in a scalable, timely and accurate fashion, it can achieve a lot using that information. In the future, I will focus on how far we can push these attributes in network control and what new services they will make possible. My approaches are (a) Defining new primitives that allow the

right delegation of network control functionalities among end-hosts, hardware switches and the controller. (b) Exploring the tradeoff between scalability, timeliness, accuracy and other aspects such as quality of service (QoS), privacy and power efficiency. (c) Defining the right abstraction based on the tradeoff that hides the complexities inside the network from operators and developing systems that implement the abstraction in an efficient and reliable way.

Ensuring network isolation and fairness: One of the main concerns of businesses for moving services to public clouds is how collocating applications of different businesses on the same set of servers and the same network affects their performance. Different applications require different classes of service from a network. Different classes should not affect each other, and flows in the same class should receive fair service with minimum overhead. Current solutions are not scalable and timely enough: First, the scale of the problem is large as there are millions of flows per second in a datacenter, and a flow may compete for resources with other flows at multiple places around the network. However, CPU cores at servers and traffic shapers (queues) at switches are limited. Second, contention can happen in very small time-scales. My observation is that there are different places in a datacenter that can apply fairness and isolation with different tradeoffs and improve scalability, e.g., hypervisor and NIC at servers and traffic shapers at switches inside the network. I will explore the tradeoff of accuracy, timeliness and resource usage among these options. In addition, a timely measurement system can help timely reaction to flow contentions. I will design a scalable and fast coordinating system that responds to flow interactions in small time-scales and provides the abstraction of an isolated fair network to operators.

Scalable middlebox control: The number of middleboxes (usually stateful devices that inspect and manipulate traffic instead of just forwarding it, e.g., intrusion detection) and their management overhead is comparable to network switches. Today, datacenters use expensive hardware to go through traffic in line rate for tasks such as attack detection and load balancing. This solution is not scalable to the fast increasing traffic inside datacenters where different tenants use each other's services. On the other hand, software middleboxes (network function virtualization) impose overhead on CPUs at servers, network bandwidth and packet latency. To make a scalable solution, part of the computation inside middleboxes can be delegated to network measurement in order to filter their input traffic, reduce their overhead and guide their scale. For example, I want to explore how network measurement can dynamically select the traffic that must go through the middleboxes, what is the right primitive to let middleboxes delegate their computations to measurement elements, and how to attribute the resource usage of a middlebox to traffic properties in order to scale out/up middlebox resources efficiently.

Accurate network control by packet-level network-wide validation & diagnosis: 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. I plan to develop a system to automatically translate control policies and network-wide invariants to the right packet-level measurement tasks and validate them. In addition, I will explore the solutions to diagnose the root-cause of a validation failure.

Controlling heterogeneous networks: In the near future, new networking technologies such as Intelligent NICs with TCAM, optical switches with fast configuration capabilities and wireless communication among racks will be deployed in production datacenters and will co-exist with current technologies. However, still there is no complete control system to help operators decide when and how to use such technologies. For example, I want to explore what types of applications benefit from intelligent NICs and how to dynamically push network functionalities to them without involving the operator. Moreover, I am interested in detecting traffic demands online and automatically scheduling them on Ethernet, optical networks and wireless networks to improve quality of service (QoS).

The marriage of measurement primitives: The work on measurement is not finished. Although, in my previous work, I explored the tradeoffs for different measurement primitives, I have never examined how to combine different primitives to leverage their strength points to cover each other weaknesses. For example, while sketches may find heavy hitters fast, their output always has some random errors. In contrast, flow-counters always provide exact values but must iterate to find a heavy hitter. In applications such as accounting where we need exact counters for heavy users, these two primitives can collaborate to detect heavy hitters fast with exact numbers. Similarly, expressive code-fragments at end-hosts can guide expensive measurements inside the network. There are other interesting cases for combining sampling with flow-counters and so on.

References

1. Omid Alipourfard, **Masoud Moshref**, Minlan Yu, "Re-evaluating Measurement Algorithms in Software", HotNets, Philadelphia, PA, 2015
2. **Masoud Moshref**, Minlan Yu, Ramesh Govindan, Amin Vahdat, "SCREAM: Sketch Resource Allocation for Software-defined Measurement", CoNEXT, Heidelberg, Germany, 2015
3. **Masoud Moshref**, Minlan Yu, Ramesh Govindan, Amin Vahdat, "DREAM: Dynamic Resource Allocation for Software-defined Measurement", SIGCOMM, Chicago, 2014
4. **Masoud Moshref**, Apoorv Bhargava, Adhip Gupta, Minlan Yu, Ramesh Govindan, "Flow-level State Transition as a New Switch Primitive for SDN", HotSDN, Chicago, 2014
5. **Masoud Moshref**, Minlan Yu, Abhishek Sharma, Ramesh Govindan, "Scalable Rule Management for Data Centers", NSDI, Lombard, 2013
6. **Masoud Moshref**, Minlan Yu, Ramesh Govindan, "Resource/Accuracy Tradeoffs in Software-Defined Measurement", HotSDN, Hong Kong, 2013
7. **Masoud Moshref**, Minlan Yu, Abhishek Sharma, Ramesh Govindan, "vCRIB: Virtualized Rule Management in the Cloud", HotCloud, Boston, 2012
8. **Masoud Moshref**, Minlan Yu, Ramesh Govindan, Amin Vahdat, "Trumpet: Timely and Precise Triggers in Data Centers", In preparation for SIGCOMM, 2016
9. Shuyong Zhu, Jun Bi, Chen Sun, Haoxian Chen, Zhilong Zheng,Hongxin Hu, Minlan Yu, **Masoud Moshref**, Chenghui Wu, Cheng Zhang, "HiPPA: a High-Performance and Programmable Architecture for Network Function Virtualization", In preparation for SIGCOMM, 2016

Masoud Moshref Javadi - Teaching Statement

Teaching and advising are essential and exciting parts of an academic career. I can teach networking and operating systems along with introductory programming courses in undergraduate level and present software-defined networking, cloud computing and distributed systems in graduate level. I look forward to mentoring students to learn and practice critical thinking, problem-solving and presentation skills.

I love learning, and I enjoy helping others experience the learning process. That is why I started teaching back in my undergraduate study in Sharif University of Technology where I was TA for many courses. An exceptional experience was the Computer Workshop lab for freshmen that is taught completely by senior and graduate students under the supervision of a faculty and a head TA. I presented for two semesters and was the head TA once. In USC during my Ph.D., I was TA for undergraduate Operating Systems (OS) course that involved a major programming project. I presented lectures for guiding 70 students through the project, held office hours and graded the project assignment. I have also been the guest lecturer for two courses and advised undergraduate and graduate students. In the following, I describe my teaching philosophy to facilitate learning and my goals in course development.

Learning by doing: I believe students will be more excited to learn about a topic once they use it in experience even in an emulated environment. When something does not work in the experience, students have to refer back to theories and techniques they may have overlooked during the lecture. Even better, once students work on real systems (many available open-source), they learn the complexities of actual systems, an experience that is also valued in the industry. When I was TA for the OS course, I held office hours to answer students question for a project on Pintos (an instructional OS). I had to review different concepts such as synchronization and memory management for students to help them finish the project.

Learning by immediate feedback: Learning is not possible without feedback. There are many ways to reduce the time to give feedback to students: (a) For the hands-on projects, the public test cases uncover (sometimes trivial) mistakes for students in an instance. Thus, they do not need to wait until office hours. (b) For many of the hands-on experiences, it is possible to make a competition. I remember I had a cloud computing course in USC, and the goal was to implement the fastest distributed system to process texts and photos in a photo-sharing service. Being able to compare the performance of my implementation with other groups was a continuous motivation. (c) I found the online discussion forums (e.g., Piazza) very helpful for students to get answers on recurring questions fast. In addition, they provide a platform for students to share knowledge.

Learning through research in graduate courses: I believe graduate courses should be designed around research through different practices: (a) Reviewing basic (sometimes old) papers because in computer science old ideas come up again and again in new contexts. I was thrilled during Advanced Operating Systems course in USC to know how much designing a fast remote procedure call has common with new techniques for bypassing network stack. (b) Reviewing state-of-the-art papers by a presentation from authors or guest lecturers. I have been the guest lecturer in two courses in USC describing my work and a topic in my research. I enjoyed the experience that students could ask directly about the process of coming up with the idea, the research challenges and future work, the opportunity that is not possible just by reading papers. (c) Performing a research project on a cutting edge problem but in small scale. This provides a taste of what happens in research

laboratories, but it can result in a publication (most likely a workshop paper) in a semester. I have advised a few graduate students on their course projects, which ended up with a workshop paper. Because the project was small, two graduate students could finish it during a semester and even have enough time to make mistakes and try different solutions.

Course development: I am looking forward to presenting courses on special topics such as cloud computing, datacenter networks, distributed systems, optimization in networking and software-defined networking for graduate level. Especially, I will pitch the courses around real-world networking problems when computing happens at large scale with low delay and high availability requirements. I show how to abstract a problem to apply theory and how to make it practical using system building techniques. I also want students to read papers about operational networking systems in the experience track of conferences in order to understand how research ideas are realized in operation and what challenges happen in deploying the systems. For undergraduate level, I can teach courses on networks, operating systems, data structures and introductory programming courses.

DREAM: Dynamic Resource Allocation for Software-defined Measurement

Masoud Moshref† Minlan Yu† Ramesh Govindan† Amin Vahdat*
†University of Southern California *Google and UC San Diego

ABSTRACT

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 TCAM counters in hardware switches to monitor traffic. However, the TCAM memory at switches is fundamentally limited and the accuracy of the measurement tasks is a function of the resources devoted to them on each switch. This paper describes an adaptive measurement framework, called DREAM, that dynamically adjusts 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 does not assume an a priori characterization of this trade-off, but instead dynamically searches for a resource allocation that is sufficient to achieve a desired level of accuracy. A prototype implementation and simulations with three network-wide measurement tasks (heavy hitter, hierarchical heavy hitter and change detection) and diverse traffic show that DREAM can support more concurrent tasks with higher accuracy than several other alternatives.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]; C.2.3 [**Network Operations**]: Network monitoring; C.2.4 [**Distributed Systems**]: Network operating systems

Keywords

Software-defined Measurement; Resource Allocation

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 [7, 11, 13, 19, 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 [39, 25, 31] 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. In a cloud setting, each tenant can issue distinct measurement tasks. Some cloud services have a large number of tenants [1], and cloud providers already offer simple per-tenant measurement services [2].

Unlike prior work [39, 35, 31, 29], which has either assumed specialized hardware support on switches for measurement, or has explored software-defined measurements on hypervisors, our paper focuses on TCAM-based measurement in switches. TCAM-based measurement algorithms can be used to detect heavy hitters and significant changes [31, 41, 26]. These algorithms can leverage existing TCAM hardware on switches and so have the advantage of immediate deployability. However, to be practical, we must address a critical challenge: TCAM resources on switches are fundamentally limited for power and cost reasons. Unfortunately, measurement tasks may require multiple TCAM counters, and the number of allocated counters can determine the accuracy of these tasks. Furthermore, the resources required for accurate measurement may change with traffic, and tasks may require TCAM counters allocated on multiple switches.

Contributions. In this paper, we discuss the design of a system for TCAM-based software-defined measurement, called DREAM. Users of DREAM can dynamically instantiate multiple concurrent measurement tasks (such as heavy hitter or hierarchical heavy hitter detection, or change 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 the traffic for each task may need to be measured at multiple switches, DREAM needs to allocate switch resources to each task.

To do this, DREAM first leverages two important observations. First, although tasks become more accurate with more TCAM 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 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 allocat-

SIGCOMM'14, August 17–22, 2014, Chicago, Illinois, USA.

http://dx.doi.org/10.1145/2619239.2626291.

ing just enough resources to tasks at switches and over time might provide *spatial and temporal statistical multiplexing benefits*.

DREAM uses both of these observations to permit more concurrent tasks than is possible with a static allocation of TCAM resources. To do this, DREAM needs to estimate the TCAM resources required to achieve the desired accuracy bound. Unfortunately, the relationship between resource and accuracy for measurement tasks cannot be characterized *a priori* because it depends upon the traffic characteristics. If this relationship could have been characterized, an optimization-based approach would have worked. Instead, DREAM contains a novel resource adaptation strategy for determining the right set of resources assigned to each task at each switch. This requires measurement algorithm-specific estimation of task accuracy, for which we have designed *accuracy estimators* for several common measurement algorithms. Using these, DREAM increases the resource allocated to a task at a switch when its global estimated accuracy is below the accuracy bound and its accuracy at the switch is also below the accuracy bound. In this manner, DREAM decouples resource allocation, which is performed locally, from accuracy estimation, which is performed globally. DREAM continuously adapts the resources allocated to tasks, since a task's accuracy and resource requirements can change with traffic. Finally, if DREAM is unable to get enough resources for a task to satisfy its accuracy bound, it *drops* the task.

DREAM is at a novel point in the design space: it permits multiple concurrent measurements without compromising their accuracy, and effectively maximizes resource usage. 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.

2. RESOURCE-CONSTRAINED SOFTWARE-DEFINED MEASUREMENT

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. Unfortunately, the number of TCAM entries are 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.

2.1 TCAM-based Measurement

In this paper, we focus on TCAM-based measurement tasks on hardware switches. Other work has proposed more advanced measurement primitives like sketches [39], which are currently not available in commercial hardware switches and it is unclear when they will be available. For this reason, our paper explicitly focuses on TCAM-based measurement, but many of the techniques proposed in this paper can be extended to sketch-based measurement (we leave such extensions to future work). In more constrained environments like data centers, it may be possible to perform measurement in software switches or hypervisors (possibly even using sketches), but this approach (a) can be compromised by malicious code on end-hosts, even in data-center settings [37, 9], (b) does not generalize to wide-area deployments of SDN [24], and (c) introduces additional constraints (like hypervisor CPU usage) [22].

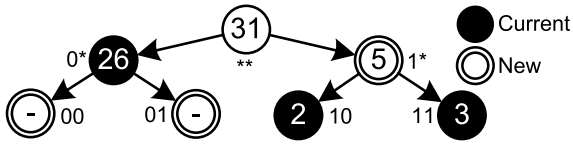


Figure 1: TCAM-based task example

To understand how TCAM memory can be effectively used for measurement, consider the heavy hitter detection algorithm proposed in [26]. 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 1 shows a prefix trie of two bits as part of source IP prefix trie of a task that finds heavy hitter source IPs (IPs 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 IPs (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 1, suppose the task 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 Task Diversity and Resource Limitations

While the previous sub-section described a way to measure heavy hitters at a single switch, the focus of our work is to design an SDM system that (a) permits multiple types of TCAM-based measurement tasks across multiple switches that may each contend for TCAM memory, and (b) adapts the resources required for concurrent tasks without significantly sacrificing accuracy.

SDM needs to support a large number of concurrent tasks, and dynamic instantiation of measurement tasks. In an SDN-capable WAN, network operators may wish to track traffic anomalies (heavy hitters, significant changes), and simultaneously find large flows to effect preferential routing [7], and may perform each of these tasks on different traffic aggregates. Operators may also instantiate tasks dynamically to drill down into anomalous traffic aggregates. In an SDN-capable multi-tenant data center, individual tenants might each wish to instantiate multiple measurement tasks. Modern cloud services have a large number of tenants; for example, 3 million domains used AWS in 2013 [1]. Per-tenant measurement services are already available — Amazon CloudWatch provides tenant operators very simple network usage counters per VM [2]. In the future, we anticipate tenants instantiating many measurement tasks to

achieve distinct goals such as DDoS detection or better bandwidth provisioning [10, 38].

Each measurement task may need hundreds of TCAM entries for sufficient accuracy [31, 26, 41], but typical hardware switches have only a limited number of TCAMs. There are only 1k-2k TCAM entries in switches [19, 23], 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.

2.3 Dynamic Resource Allocation for SDM

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

Leverage: Diminishing returns in accuracy for measurement. The *accuracy* of a measurement task depends on the resources allocated to it [31, 39]. For example, for heavy hitter (HH) detection, *recall*, the fraction of true HHs that are detected, is a measure of accuracy. Figure 2 shows the result of our HH detection algorithm on a CAIDA traffic trace [3] with a threshold of 8 Mbps (See Section 5 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(a)). There is a point of diminishing returns for many measurement tasks [17, 30, 28, 27] 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 [31].

Another important aspect of the relationship between accuracy and resource usage of TCAM-based 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 TCAM counters to these tasks to enable them to achieve the specified accuracy, adapts TCAM 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 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. This presents an opportunity to *statistically multiplex* TCAM 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. This dependence on TCAM resources with traffic is shown in Figure 2(a), where the recall of the HH detection task with 256 entries decreases in the presence of more HHs and we need more resources to keep its recall above 50%. 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.

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 IPs on flows of a prefix that come from multiple switches. Figure 2(b) 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 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.

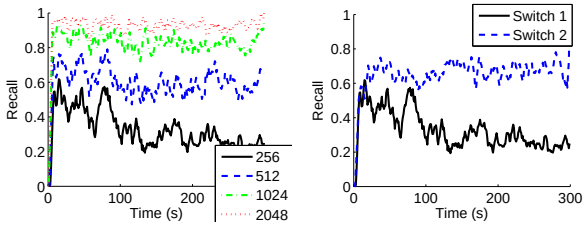
These leverage points suggest that it may be possible to efficiently use TCAM resources to permit multiple concurrent measurement tasks by (a) permitting operators[2] to specify desired accuracy bounds for each task, and (b) adapting the resources allocated to each task in a way that permits temporal 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 because it depends on the traffic, the task type, the measurement algorithms, and the parameters [31]. 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 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 5, 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

[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] 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.



(a) Resource change (b) Spatial diversity

Figure 2: Accuracy of HH detection

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 leverages 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.

3. DREAM OVERVIEW

DREAM enables resource-aware software-defined measurement. It supports dynamic instantiation of measurement tasks with a specified accuracy, and automatically adapts TCAM resources allocated to each task across multiple switches. DREAM can also be extended to other measurement primitives (like sketches) and tasks for which it is possible to estimate accuracy.

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

Our current implementation supports three *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 IPs contributing large traffic in the network.

Hierarchical Heavy Hitter (HHH) Some fields in the packet header (such as the source/destination IP addresses) embed hierarchy, and many applications such as DDoS detection require aggregates computed at various levels within a hierarchy [33]. Hierarchical heavy hitter (HHH) detection is an extension of HH detection that finds longest prefixes that exceed a certain threshold even after *excluding* any HHH descendants in the prefix trie [15].

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 IPs have been used for anomaly detection [41].

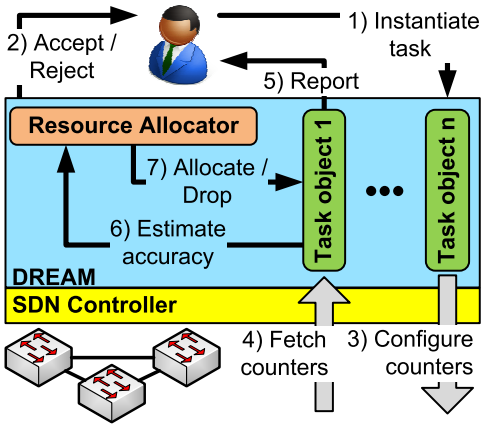


Figure 3: DREAM System Overview

Each of these tasks takes four parameters: a *flow filter* specifying the traffic aggregate to consider for the corresponding phenomenon (HH, HHH or CD); a *packet header field* on which the phenomenon is defined (e.g., source IP address); a *threshold* specifying the minimum volume constituting a HH or HHH or CD; 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 responsibility to install measurement rules at all relevant switches.

Workflow. Figure 3 shows the DREAM workflow, illustrating both the interface to DREAM and the salient aspects of its internal operation. 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 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 reconfigure one or more switches (step 3). If a task is dropped for lack of resources, DREAM removes its task object and de-allocates the task's TCAM counters.

DREAM Generality. These *network-wide* measurement tasks have many applications in data centers and ISP networks. For example, they are used for multi-path routing [7], optical switching [12], network provisioning [21], threshold-based accounting [20], anomaly detection [42, 41, 26] and DDoS detection [33].

Furthermore, DREAM can be extended to more general measurement primitives beyond TCAMs. Our tasks are limited by TCAM capabilities because TCAM counters can only measure traffic volumes for specific prefixes. Moreover, TCAM-based tasks need a few epochs to drill down to the exact result. However, DREAM's

key ideas — using accuracy estimators to allocate resources, and spatially multiplexing resource allocation — can be extended to other measurement primitives not currently available on commodity hardware, such as sketches. Sketches do not require controller involvement to detect events and can cover a wider range of measurement tasks than TCAMs (volume and connection-based tasks such as Super-Spreader detection) [39]. We can augment DREAM to use sketches, since sketch accuracy depends on traffic properties and it is possible to estimate this accuracy [17]. We leave discussion of these extensions to future work.

There are two main challenges in DREAM, discussed in subsequent sections: the design of the resource allocator, and the design of task-specific accuracy estimators.

4. DYNAMIC RESOURCE ALLOCATION

DREAM allocates TCAM resources to measurement tasks on multiple switches. 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), 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 TCAM constraints. This optimization technique requires a characterization of the resource-accuracy curve, a function that maps target accuracy to TCAM 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, 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. Simply using g_i to make allocation decisions can be misleading: 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%. Conversely, $l_{i,s}$ may be low, but adding measurement resources to i at switch s may be unnecessary if g_i is already above the accuracy bound. For the above example, we do not want to increase resources on switch A when the accuracy bound is 40%.

This discussion 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 3.

The Per-switch Resource Allocator. The heart of DREAM is the per-switch resource allocator (Figure 3), which runs on the controller and maps TCAM counters to tasks for each switch.[3] 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*

[3] 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.

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.

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.[4] 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 4 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 $R_{i,s}$.

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.

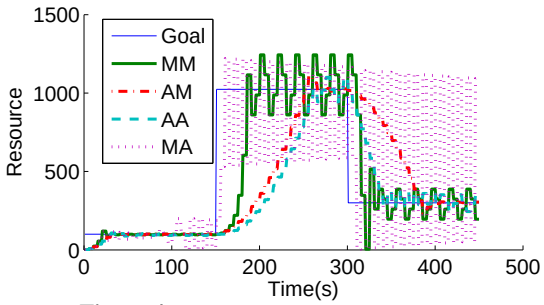


Figure 4: Comparing step updates algorithms

However, because it is impossible to predict traffic variability, DREAM may infrequently drop tasks when headroom is insufficient.[5] 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 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.

5. TASK OBJECTS AND ACCURACY ESTIMATION

In DREAM, task objects implement individual task instances. 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, such as the *accuracy estimator*.

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.

[4] 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.

[5] 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.

Algorithm 1: DREAM task object implementation

```
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);
8 end
```

5.1 A Generic Algorithm for Task Objects

A DREAM task object implements Algorithm 1; 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 3). 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 4) 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).

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`.

5.2 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 Algorithm 1). 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 5 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.

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. Algorithm 2 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.

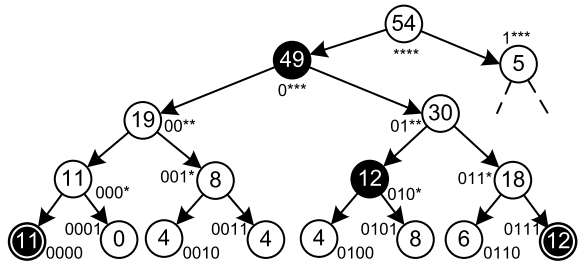


Figure 5: A prefix trie of source IPs where the number on each node shows the bandwidth used by the associated IP prefix in Mb in an epoch. With threshold 10, the nodes in double circles are heavy hitters and the nodes with shaded background are hierarchical heavy hitters.

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 2-4). 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 5-16). First, it picks the counter with maximum score to divide (line 6) and determines if that results in overloaded switches, designated by the set F (line 7). 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 10). Next, if the score of the counter is worth the cost, we apply divide and merge (lines 12-15). After dividing, the score of children will be half of the parent's score [42]. 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 [26], HHH [31], and traffic changes [41]). In this paper, we provide a general task-independent algorithm, ensuring that the algorithm uses bounded resources and adapting to resource changes on multiple switches.

Algorithm 2: Divide and Merge

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

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 5 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 $00**$. [6]

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 [36].

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.

5.3 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 we consider two task 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 or change detection. Depending on the type of measurement task, DREAM estimates one of these accuracy measures to determine TCAM resource allocation.

The task-dependent algorithms for these tasks are summarized in Table 1, but we discuss some of the non-trivial algorithms below.

Heavy hitters: A heavy hitter is a traffic aggregate that exceeds a specified volume. For example, we can define heavy hitters as the source IPs whose traffic exceeds a threshold θ over a measurement epoch. Figure 5 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.

[6] 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	Estimate accuracy	Score
HH	Report exact counters with $volume > \theta$	Estimate recall by estimating missed HHs	$\frac{volume}{\#wildcards+1}$
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 [15]	Estimate precision by finding if a detected HHH is a true one	$volume$
CD	Report exact counters with $\lvert volume - mean \rvert > \theta$	Estimate recall by estimating missed changes	$\frac{\lvert volume-mean \rvert}{\#wildcards+1}$

Table 1: Task dependent methods

Accuracy Estimation: 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 5 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 4). 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 are 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 hitters: A variant of heavy hitters, called *Hierarchical Heavy Hitters (HHHs)* [15] is useful for anomaly detection [42] and DDoS detection [33]. 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 5, 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.

Accuracy Estimation: 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 5), 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 5, 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: 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, prefix scoring, and accuracy estimation are similar to those for HH tasks (Table 1): wherever volume is used in HH tasks, $|volume - mean|$ is used for CD.

6. 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.

6.1 Evaluation Methodology

DREAM Implementation: We have implemented the DREAM resource allocator and the task objects in Java on the Floodlight controller [4]. Our implementation interfaces both with hardware OpenFlow switches, and with Open vSwitch [5]. We have also implemented alternative resource allocation strategies, described below. Our total implementation is nearly 20,000 lines of code.

DREAM 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 of DREAM to these parameters is explored in Section 6.4.

Tasks: Our workload consists of the three types of tasks, HH, HHH and CD, 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 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 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 and other alternatives.

Finally, we use a 5-hour CAIDA packet trace [3] 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 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).

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. 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 [6]) 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 [5] 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 6.5). In our experiments, the DREAM prototype runs on a Floodlight controller [4] on a quad core 2.4 Ghz Xeon processor connected to the switches through a 1Gbps shared link with ping delay of 0.25ms.

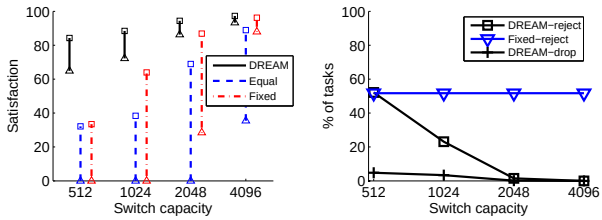
6.2 Results from Prototype

Figure 6 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 7). For example, Figure 6(b) 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%.

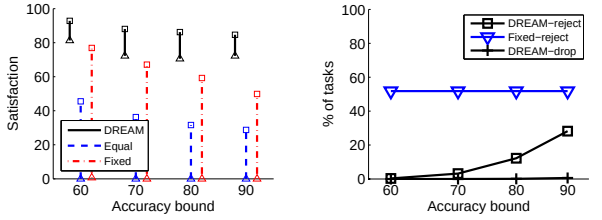
Highly resource-constrained switches. For smaller switches, where our workload overloads the resources on switches, DREAM leverages rejection to limit the load and keep active tasks satisfied by multiplexing resources. For example, in Figure 6(a) for a switch with 512 counters, DREAM rejects about 50% 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 7(a)) 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) Satisfaction (b) Rejection and drop

Figure 8: Large scale simulation (combined workload)



(a) Satisfaction (b) Rejection and drop

Figure 9: Accuracy bound paramter sensitivity analysis

6.3 Results from Simulation at Scale

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 and is validated for the same setting [32].

DREAM's superior performance is also evident in larger networks. Figure 8 compares the satisfaction and rejection ratio of the combined workload on 32 switches with 4096 tasks (results for individual task types are quantitatively similar). 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.

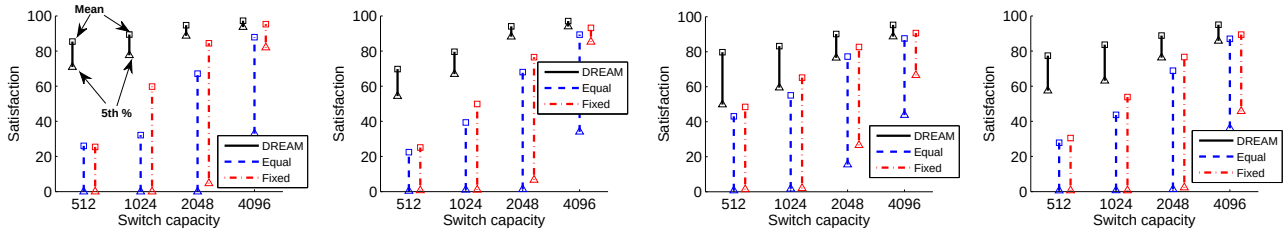
6.4 Parameter Sensitivity Analysis

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 9). A companion report [32] evaluates sensitivity to HHH threshold, number of switches per task, task duration and task arrival rate. 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 (Figure 9). DREAM is also uniformly better than Equal allocation because it effectively multiplexes resources across tasks.

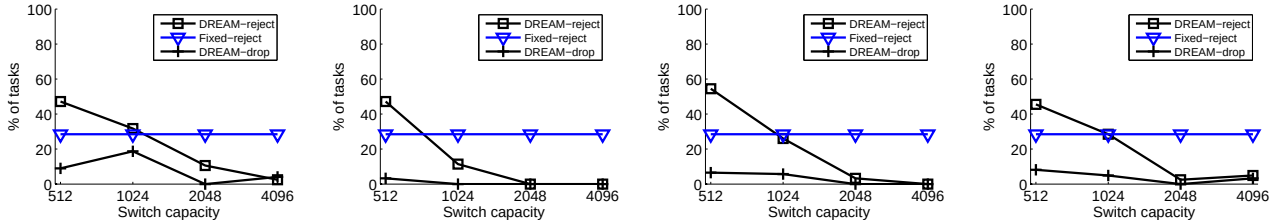
Headroom is important to keep drop rate low. If DREAM does not reject tasks a priori, many tasks will starve just after joining the system. For example, Figure 10(b) shows a drop rate of 30% for DREAM when there is no headroom at an allocation interval of 2s. Interestingly, the level of headroom does not 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 10(a) shows



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

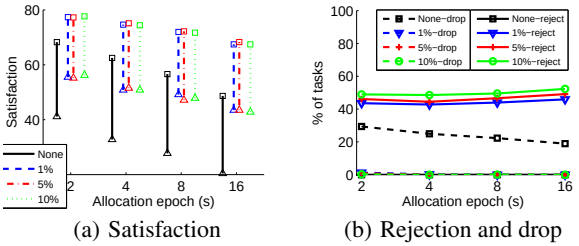
Figure 6: Satisfaction in prototype



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

Figure 7: Rejection and drop in prototype

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.



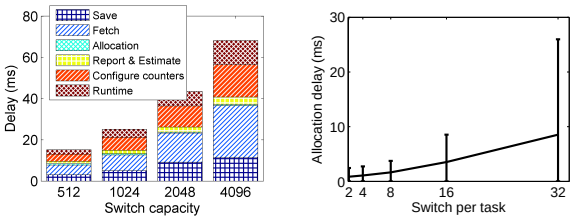
(a) Satisfaction (b) Rejection and drop

Figure 10: Headroom and allocation epoch (combined workload)

6.5 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.

Figure 11(a) 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 admit-



(a) Prototype delay (b) Allocation delay

Figure 11: Control loop delay (combined workload)

ted; (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 11(b) 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.

7. RELATED WORK

Software-defined measurement and programmable measurement: Prior work has explored different measurement primitives [20, 35], but, unlike DREAM, 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 [39, 31, 25] and programmable measurement [18, 40] has shown the benefits of

allowing operators or cloud tenants to customize the measurement for their traffic with different measurement primitives. Amazon CloudWatch [2] also provides simple customized measurement interface for tenants. Like these, DREAM allows measurement tasks to specify the flows and traffic characteristics to measure, but, beyond prior work, provides dynamic resource allocation solutions to enable more and finer-grained measurement tasks.

Resource allocation of measurement tasks: OpenSketch [39] uses *worst case* theoretical bounds of sketches to allocate resources on a single switch to measurement tasks. CSAMP [34] uses consistent sampling to distribute flow measurement on multiple switches for a single measurement task and aims at maximizing the flow coverage. Volley [29] uses a sampling-based approach to monitor state changes in the network, with the goal of minimizing the number of sampling operations. Payless [14] 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. 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.

TCAM-based measurement and accuracy estimators: Previous TCAM-based algorithms for specific measurement tasks either only work on a single switch [31, 26, 25] or do not adjust counters for bounded resources at switches [41, 26]. We designed a generic divide-and-merge measurement framework for multiple switches with resource constraints. Previous work has proved the theoretical bounds for the *worst case* resource usage for only hash-based measurements [17, 16]. We proposed heuristics for estimating the accuracy of TCAM-based measurement algorithms by exploiting relationships between counters already collected.

8. CONCLUSIONS

Measurement is fundamental for network management systems. DREAM enables operators and cloud tenants to flexibly specify their measurement tasks in a network, and dynamically allocates TCAM resources to these tasks based on the resource-accuracy tradeoffs for each task. DREAM ensures high accuracy for tasks, while taking network-wide resource constraints as well as traffic and task dynamics into account.

9. ACKNOWLEDGEMENTS

We thank our shepherd Rodrigo Fonseca and SIGCOMM reviewers for their helpful feedback. This paper is partially supported by Software R&D Center at Samsung Electronics, Cisco, Google, and USC Zumberge.

10. REFERENCES

[1] http://news.netcraft.com/archives/2013/05/20/amazon-web-services-growth-unrelenting.html.

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

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

[4] Floodlight. http://www.projectfloodlight.org/floodlight/.

[5] Open vSwitch. http://openvswitch.org/.

[6] Pica8 P-3290 switch. http://www.pica8.com/documents/pica8-datasheet-48x1gbe-p3290-p3295.pdf.

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

[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] A. M. Azab, P. Ning, and X. Zhang. SICE: A Hardware-level Strongly Isolated Computing Environment for x86 Multi-core Platforms. In *CCS*, 2011.

[10] H. Ballani, P. Costa, T. Karagiannis, and A. I. Rowstron. Towards Predictable Datacenter Networks. In *SIGCOMM*, 2011.

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

[12] 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.

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

[14] 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 NOMS*, 2014.

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

[16] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms*, 55(1), 2005.

[17] G. Cormode and S. Muthukrishnan. Summarizing and Mining Skewed Data Streams. In *SIAM Conference on Data Mining (SDM)*, 2005.

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

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

[20] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting. *SIGCOMM Computer Communication Review*, 32(4):323–336, 2002.

[21] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving Traffic Demands for Operational IP Networks: Methodology and Experience. *Transactions on Networking*, 9(3), 2001.

[22] 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.

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

[24] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a Globally-deployed Software Defined WAN. In *SIGCOMM*, 2013.

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

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

[27] 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.

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

[29] S. Meng, A. K. Iyengar, I. M. Rouvellou, and L. Liu. Volley: Violation Likelihood Based State Monitoring for Dataceners. *ICDCS*, 2013.

[30] M. Mitzenmacher, T. Steinke, and J. Thaler. Hierarchical Heavy Hitters with the Space Saving Algorithm. *arXiv:1102.5540*, 2011.

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

[32] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. DREAM: Dynamic Resource Allocation for Software-defined Measurement. Technical Report 14-945, Computer Science, USC, 2014. http://www.cs.usc.edu/assets/007/91037.pdf.

[33] 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.

[34] 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.

[35] V. Sekar, M. K. Reiter, and H. Zhang. Revisiting the Case for a Minimalist Approach for Network Flow Monitoring. In *IMC*, 2010.

[36] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., 2001.

[37] Z. Wang and X. Jiang. Hypersafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-flow Integrity. In *SP*, 2010.

[38] D. Xie, N. Ding, Y. C. Hu, and R. Kompella. The Only Constant is Change: Incorporating Time-varying Network Reservations in Data Centers. *SIGCOMM Computer Communication Review*, 42(4), 2012.

[39] M. Yu, L. Jose, and R. Miao. Software Defined Traffic Measurement with OpenSketch. In *NSDI*, 2013.

[40] L. Yuan, C.-N. Chuah, and P. Mohapatra. ProgME: Towards Programmable Network MEasurement. *Transactions on Networking*, 19(1), 2011.

[41] Y. Zhang. An Adaptive Flow Counting Method for Anomaly Detection in SDN. In *CoNEXT*, 2013.

[42] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund. Online Identification of Hierarchical Heavy Hitters: Algorithms, Evaluation, and Applications. In *IMC*, 2004.

Scalable Rule Management for Data Centers

Masoud Moshref[†] *Minlan Yu*[†] *Abhishek Sharma*[†*] *Ramesh Govindan*[†]
[†] *University of Southern California* [*] *NEC Labs America*

Abstract

Cloud operators increasingly need more and more fine-grained rules to better control individual network flows for various traffic management policies. In this paper, 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.

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 [8, 32, 28, 17]. 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 2). Given the expected scale in the number of rules, rule processing in future data 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 paper, 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 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.

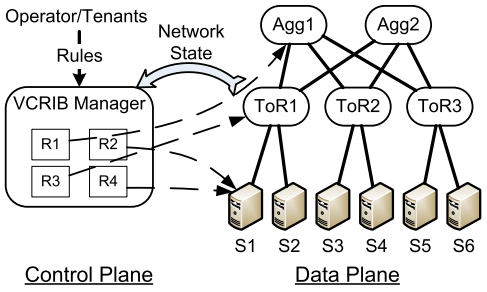


Figure 1: *Virtualized Cloud Rule Information Base (vCRIB)*

vCRIB's rule placement algorithms achieve resource-feasible, low-overhead rule placement by off-loading 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 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 feasi-

ble low overhead placement.

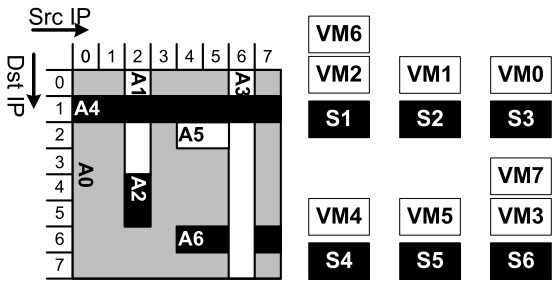
We evaluate (Section 4) vCRIB through large-scale simulations, as well as experiments on a prototype built on Open vSwitch [4] and POX [1]. Our results demonstrate that vCRIB is able to find feasible placements with a few percent traffic overhead, even for a particularly adversarial setting in which 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.

2 Motivation and Challenges

Today, tenants in data centers operated by Amazon [5] 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 paper, 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 [32], fair sharing [29]). For example, to ensure each tenant gets a fair share of the bandwidth, Seawall [32] 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 [8, 11] or energy efficiency [17], 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 [8] performs specific traffic engineering for large flows, while VLAN-based traffic management solutions [28] use different VLANs to route packets. Most of these policies,



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

Figure 2: *Sample ruleset (black is accept, white is deny) and VM assignment (VM number is its IP)*

expressed in high level languages [18, 37], 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 [29] or per VM [29], and access control policies might require one rule per VM pair [30]. 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 [22], 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 [8, 11, 17] 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 [10]. 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.

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 paper 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 2(a) shows a flow-space with source and

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

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 [32]), 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 [6]. 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 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 2(b)). If the hypervisor at S1 does not have enough resources to process the deny rule $A3$ in Figure 2(a), we can install the rule at ToR1, introducing more traffic overhead. Indeed, some commercial products already support offloading rule processing from hypervisors to ToRs [7]. 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 [15], 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 [15].

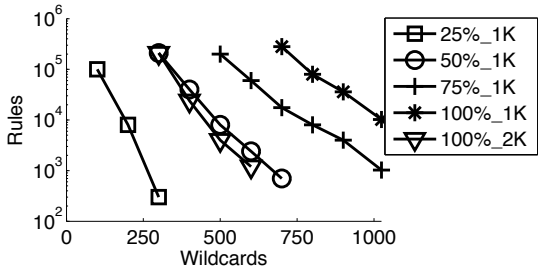


Figure 3: *Performance of openvswitch (The two numbers in the legend mean CPU usage of one core in percent and number of new flows per second.)*

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 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 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.

3 vCRIB Automated Rule Management

To address these challenges, we propose the design of a system called vCRIB (virtual Cloud Rule Information Base) (Figure 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 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 reac-*

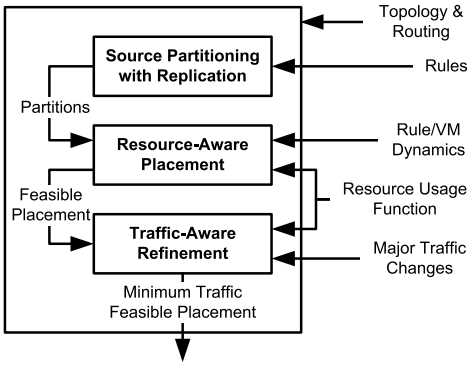


Figure 4: *vCRIB controller architecture*

tive 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 paper. A hybrid approach, where some rules can be inserted reactively, is left to future work.

Designs \ Challenges	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 1: *Design choices and challenges mapping*

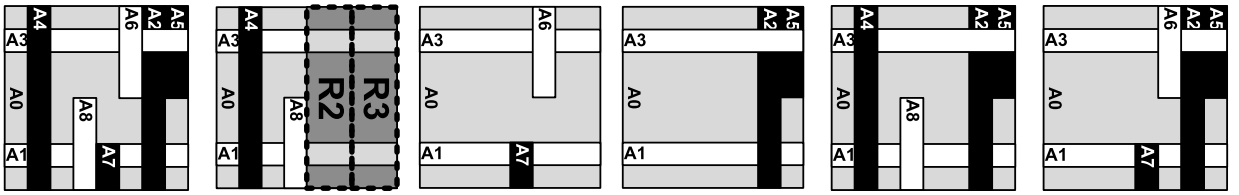
vCRIB makes several carefully chosen design decisions (Figure 4) that help address the diverse challenges discussed in Section 2 (Table 1). It partitions the rule space to break dependencies between rules, where each partition contains rules that can be co-located with each other; thus, a partition is the unit 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.

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 [32], 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 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* [38]. Aggressive rule splitting can create many small partitions making it flexible to place the partitions at different switches [26], 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.



(a) Ruleset (b) Partition-with-replication (c) P1 & P3 on a device (d) P2 & P3 on a device

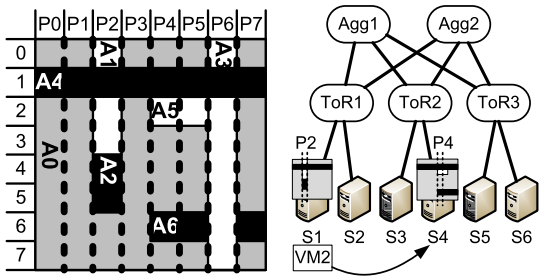
Figure 5: *Illustration of partition-with-replications (black is accept, white is deny)*

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 5(a), we can form the three partitions shown in Figure 5(b). 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 5(c), 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 6(a) 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



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

Figure 6: *Rule partition example*

one redirection rule installed at the source 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.

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] This problem, even for partition-with-splitting, is equivalent to the *generalized assignment problem*, which is NP-hard and even APX-hard to approximate [14]. Moreover, existing approximation algorithms for this problem are inefficient. We refer the reader to a technical report which discusses this in greater depth [27].

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

[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.

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 [20]. 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 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 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 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 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]

[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.

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 [21]. 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.

Algorithm 1 First Fit Decreasing Similarity Algorithm

```
P = set of not placed partitions
while |P| > 0 do
    Select a partition P_i randomly
    Place P_i on an empty device M_k.
    repeat
        Select P_j ∈ P with maximum similarity to P_i
    until Placing P_j on M_k Fails
end while
```

We use a heuristic algorithm for bin-packing similar partitions called *First Fit Decreasing Similarity* (FFDS) (Algorithm 1) which extends the traditional FFD algorithm [33] 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.

For the memory usage model, since we use per-source partitions, we can quickly find partitions similar to a given partition, and improve the execution time of the algorithm from a few minutes to a second. Since per-source partitions are ordered in the source IP dimension and the rules are always contiguous blocks crossing only

[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.

neighboring partitions, we can prove that the most similar partitions are always the ones adjacent to the partition [27]). For example, $P4$ has 4 common rules with $P5$ but 3 common rules with $P7$ in Figure 6(a). So in the third step of FFDS, we only need to compare left and right unassigned partitions.

To illustrate the algorithm, suppose each server in the topology of Figure 1 has a capacity of four rules to place the partitions and switches have no capacity. Considering the ruleset in Figure 2(a), 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 $P7$.

We have proved that, FFDS algorithm is 2-approximation for resource-aware placement in networks with only memory-constrained devices [27]. Approximation bounds for CPU-constrained devices is left to future work.

Our FFDS algorithm is inspired by the tree-based placement algorithm proposed in [33], 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 since 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). (The construction of sub-trees is discussed in a technical report [27]).

Finally, it might seem that, because vCRIB uses per-source partitions, it cannot efficiently handle a rule with a wildcard on the source IP dimension. Such a rule would have to be placed in every partition in the source IP range specified by the wildcard. Interestingly, in this case vCRIB works quite well: since all partitions on a machine will have this rule, our similarity-based placement will result in only one copy of this rule per device.

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 6(a), assume the traffic for each rule (excluding A0) is proportional to the area it covers and generated from servers in topology of Figure 6(b). 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 1); (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 1). 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:

$$M(P_i, j) = (T(P_i, d(P_i)) - T(P_i, j)) + (Q_{d(P_i)} - Q_j) \quad (1)$$

Our traffic-aware refinement algorithm is *benefit-greedy*, as described in Algorithm 2. The algorithm is given a time budget (a "timeout") to run; in practice, we

Algorithm 2 Benefit-Greedy algorithm

```
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
```

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.[6]

We now illustrate the benefit-greedy algorithm (Algorithm 2) using our running example in Figure 6(b). 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 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 $S4$.

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 [20] 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 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.

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

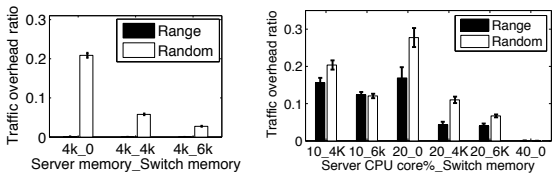
4.1 Simulation Setup

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 [31], 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 [35] 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. 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) [8]. 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 [10], 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 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 [10]. 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 [10]. We run each experiment multiple times with different random seeds to get a stable mean and standard deviation.



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

Figure 7: *Traffic overhead and resource constraints tradeoffs*

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.

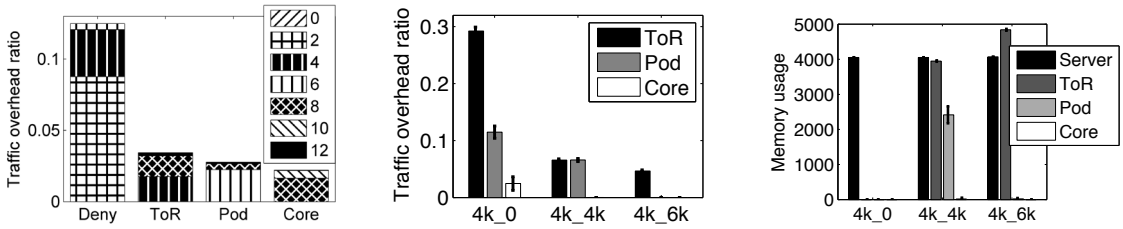
vCRIB uses similarity to find feasible solutions when SourcePlacement is infeasible. With *Range* IP allocation, partitions in the Source IP dimension which are similar to each other are saved on one server, so the average load on machines is smaller for SourcePlacement. However, there may still be a 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 7(a) 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 7(b)). 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.

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.



(a) Traffic overhead for different rules (b) Traffic overhead on different links (c) Memory usage on different devices

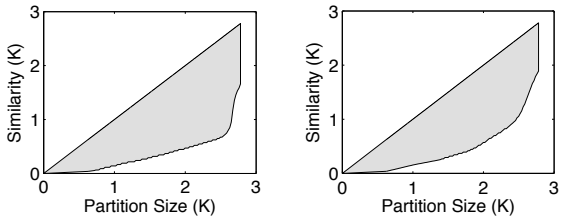
Figure 8: *Spatial distribution of traffic and resource usage*

vCRIB is effective in leveraging on-path and nearby devices. Figure 8(a) 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 8(b) 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 8(c) 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. Note that the servers are always full as they are the best place for saving partitions.

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. We have also evaluated a skewed distribution of the number of IPs/VMs per machine but have not seen major changes in the traffic overhead. In this case, vCRIB was still able to find a nearby machine with lower load. We also conducted another experiment with different traffic locality patterns, which showed that having more non-local flows gives vCRIB more choices to offload rule processing and reach feasible solutions with lower traffic overhead. Finally, experiments on FFDS performance for different machine capacities [27] also validates its superior performance comparing to the tree-based placement [33]. Beyond these kinds of analyses, we have also explored the parameter space of similarity and partition size, which we discuss next.



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

Figure 9: *vCRIB working region and ruleset properties*

vCRIB uses similarity to accommodate larger partitions. We have explored two properties of the rules in partitions by changing the ruleset. In Figure 9, 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.

Figure 9(a) 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 9(b)) 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 [27].

4.5 Reaction to Cloud Dynamics

Figure 10 compares benefit-greedy (with timeout 10 seconds) with overhead-greedy and a randomized algorithm[7] after a single VM migration for the 4K_0 case. Each point in Figure 10 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 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.

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 11 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.

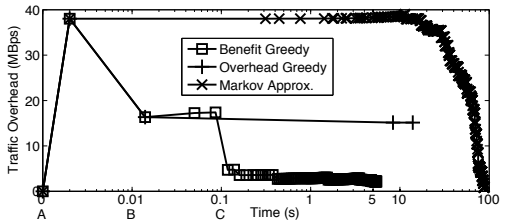


Figure 10: *Traffic refinement for one VM migration*

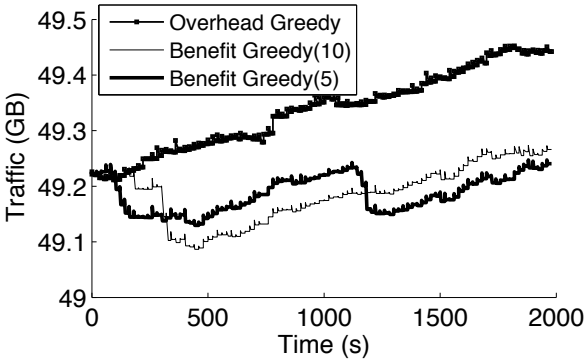


Figure 11: *The trend of traffic during multiple VM migration*

4.6 Prototype Evaluation

We built vCRIB prototype using Open vSwitch [4] 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[8] 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 [11]. 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

[7] Markov Approximation [20] with target switch selection probability $\propto$ exp(traffic reduction of migration step)

[8] 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.

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 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.

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 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.

5 Related Work

Our work is inspired by several different strands of research, each of which we cover briefly.

Policies and rules in the cloud: Recent proposals for new policies often propose customized systems to manage rules on either hypervisors [4, 13, 32, 30]) or switches [3, 8, 29]. 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 [12, 23, 38, 13]. vCRIB focuses on data centers, which are more dynamic, more sensitive to traffic overhead, and face heterogeneous resource constraints.

Distributed firewall: Distributed firewalls [9, 19], 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 [36, 34, 16, 25, 24] 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 [38], 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 [26], 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.

6 Conclusion

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.

References

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

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

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

[4] Open vSwitch. http://openvswitch.org/.

[5] Private conversation with Amazon.

[6] Private conversation with rackspace operators.

[7] Virtual networking technologies at the server-network edge. http://h20000.www2.hp.com/bc/docs/support/SupportManual/c02044591/c02044591.pdf.

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

[9] S. M. Bellovin. Distributed Firewalls. *;login:*, November 1999.

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

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

[12] M. Casado, M. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker. Rethinking Enterprise Network Control. *IEEE/ACM Transactions on Networking*, 17(4), 2009.

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

[14] C. Chekuri and S. Khanna. A PTAS for the Multiple Knapsack Problem. In *SODA*, 2001.

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

[16] P. Gupta and N. McKeown. Packet Classification using Hierarchical Intelligent Cuttings. In *Hot Interconnects VII*, 1999.

[17] 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.

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

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

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

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

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

[23] 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.

[24] V. Kriakov, A. Delis, and G. Kollios. Management of Highly Dynamic Multidimensional Data in a Cluster of Workstations. *Advances in Database Technology-EDBT*, 2004.

[25] 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.

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

[27] M. Moshref, M. Yu, A. Sharma, and R. Govindan. vCRIB: Virtualized Rule Management in the Cloud. Technical Report 12-930, Computer Science, USC, 2012. http://www.cs.usc.edu/assets/004/83467.pdf.

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

[29] L. Popa, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing The Network In Cloud Computing. In *HotNets*, 2011.

[30] L. Popa, M. Yu, S. Y. Ko, I. Stoica, and S. Ratnasamy. CloudPolice: Taking Access Control out of the Network. In *HotNets*, 2010.

[31] S. Rupley. Eyeing the Cloud, VMware Looks to Double Down On Virtualization Efficiency, 2010. `http://gigaom.com/2010/01/27/eyeing-the-cloud-vmware-looks-to-double-down-on-virtualization-efficiency`.

[32] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the Datacenter Networks. In *NSDI*, 2011.

[33] M. Sindelar, R. K. Sitaram, and P. Shenoy. Sharing-Aware Algorithms for Virtual Machine Colocation. In *SPAA*, 2011.

[34] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification Using Multidimensional Cutting. In *SIGCOMM*, 2003.

[35] D. E. Taylor and J. S. Turner. ClassBench: A Packet Classification Benchmark. *IEEE/ACM Transactions on Networking*, 15(3), 2007.

[36] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. Efficuts: Optimizing Packet Classification for Memory and Throughput. In *SIGCOMM*, 2010.

[37] A. Voellmy, H. Kim, and N. Feamster. Procera: A Language for High-Level Reactive Network Control. In *HotSDN*, 2010.

[38] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-Based Networking with DIFANE. In *SIGCOMM*, 2010.

SCREAM: Sketch Resource Allocation for Software-defined Measurement

Masoud Moshref† Minlan Yu† Ramesh Govindan† Amin Vahdat*
†University of Southern California *Google

ABSTRACT

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 for using sketches for network measurement. However, sketches in hardware switches use constrained resources such as SRAM memory, and the accuracy of measurement tasks is a function of the resources devoted to them on each switch. This paper presents SCREAM, a system for allocating resources to sketch-based measurement tasks that ensures a user-specified minimum accuracy. SCREAM estimates the instantaneous accuracy of tasks so as to dynamically adapt the allocated resources for each task. Thus, by finding the right amount of resources for each task on each switch and correctly merging sketches at the controller, SCREAM can multiplex resources among network-wide measurement tasks. Simulations with three measurement tasks (heavy hitter, hierarchical heavy hitter, and super source/destination detection) show that SCREAM can support more measurement tasks with higher accuracy than existing approaches.

CCS Concepts

•**Networks** → **Network resources allocation; Network monitoring; Data center networks;** *Programmable networks;*

Keywords

Software-defined Measurement; Sketches; Resource Allocation

1. INTRODUCTION

Traffic measurement plays an important role in network management. For example, traffic accounting, traffic engineering, load balancing and performance diagnosis require measuring traffic on multiple switches in the network [6, 10]. Software-defined Measurement (SDM) [40, 33] facilitates controller-directed network-wide measurement: with SDM, operators or tenants can submit measurement tasks to the SDN controller, and the SDN controller configures switches to monitor traffic for each task, then collects statistics and produces measurement reports.

A recent prior work in software-defined measurement [33] has relied on flow-based counters. These counters are often implemented using TCAM memory, which is expensive and power hungry. Moreover, flow-based counters are limited to supporting volume-based measurement tasks such as heavy hitter detection and often require a large number of counters. For example, 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 [32, 41, 24]. Such prefix-based summarization has two drawbacks: it cannot be applied to many tasks such as flow-size distribution and entropy calculation, and it can take multiple measurement epochs to reconfigure counters (e.g., to zoom into 32 levels in the IP prefix tree) [32].

In contrast, this paper focuses on hash-based counters, or *sketches* [40]. 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 [16], super-spreaders [17], large changes [26], flow-size distribution [27], flow quantiles [16], and flow-size entropy [29]. Finally, they can capture the right set of flow properties in the data plane without any iterative reconfiguration from the controller.

Any design for sketch-based 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. Furthermore, in a cloud setting, each tenant can issue distinct measurement tasks within its own virtual network.

The second challenge is that sketch-based measurement tasks may require significant resources. To achieve a re-

CoNEXT'15 December 01–04, 2015, Heidelberg, Germany
 ISBN 978-1-4503-3412-9/15/12…$15.00
DOI: http://dx.doi.org/10.1145/2716281.2836106

quired accuracy, each task may need up to a million counters, and the number of counters is bounded by resources such as 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. 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). These resource estimates are also dependent on traffic. Prior work [40] 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, and dynamically adapts the resource allocation as traffic changes over time and across switches.

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. Our paper makes two contributions: *(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 [40] 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 a dynamic resource allocation algorithm [33] 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.

Finally, SCREAM builds upon our prior work on DREAM [33], which establishes a framework for dynamic resource allocation for TCAM-based measurement tasks. SCREAM deliberately preserves many of the elements of DREAM (Section 3), to permit a unified system that multiplexes resources across different types of measurement tasks.

2. BACKGROUND AND MOTIVATION

Sketch-based SDM. 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 [40]. Note that the same accuracy estimation and similar resource allocation technique 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 [16] 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 IPs 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 $x \in (0 \ldots D)$ with size I_x, the switch computes d pairwise independent hash functions and updates counters, $A[i,h_i(x)] += I_x, i \in (1 \ldots d)$. 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 4). 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 [28] or multiple rounds of approximation over all detected prefixes [30], add more computation overhead to the controller, and their resource-accuracy trade-off is not known in advance.

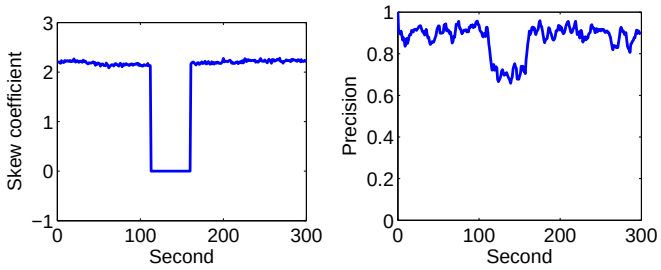
Many sketches have been proposed for counting distinct items [21, 20]. We use HyperLogLog [21] as its space usage is near-optimal, and it is easier to implement than the optimal algorithm [23]. First, we hash each item and count the num-

ber 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.

Why is sketch-based SDM resource constrained? SDM permits multiple instances of measurement tasks execute concurrently in a network which together require a lot of resources. 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 [6] and finding sources that make many connections for anomaly detection [39]. 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 [1], and Google Andromeda allows SDN-based network functionality virtualization for tenants [38]. Besides, modern clouds service a large number of tenants (3 million domains used AWS in 2013 [2]), so SDM with many measurement tasks will be common in future clouds.

However, switches have limited memory and bandwidth resources for storing these counters. Today's switches have 128 MB SRAM capacity (HP 5120-48G EI [4]) which can support 4-128 tasks where each task needs 1-32 MB SRAM counters [40]. 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) [19]. 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 fewer counters to reduce memory usage, lower network overhead and send counters more frequently to the controller to report in a short time-scale.

Why dynamic allocation? Prior work [40] 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



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

Figure 1: Variation of traffic skew and sketch accuracy over time

5.5 MB[1]. However, the total traffic T of a link may not always reach the link capacity. In addition, Cormode [18] 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 example, if the link utilization is 20% in average with a skew factor of 1.3 [18], each row will need only 1040 counters which require 260 KB of SRAM. Other sketches also exhibit traffic-dependent accuracy [13, 27].

These trade-off formulations are loose because the optimal resource requirement of a sketch for a given accuracy depends on the *traffic* that changes over time. For example, Figure 1a shows the skew of traffic from source IPs in CAIDA trace [3] over time. (Our skew metric is the *slope* of a fitted line on the log-log diagram of traffic volume from IPs vs. their rank (ZipF exponent).) The skew decreases from time 110 to 160 because of a DDoS attack. Figure 1b 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. This presents an opportunity to *statistically multiplex* SRAM and bandwidth resources across tasks on a single switch by dynamically adjusting the size of sketch.

Besides, we may need sketches with different sizes on different switches for tasks that monitor traffic on multiple switches. For example, we may need to find heavy hitter source IPs on flows coming from two switches (say A and B). These switches monitor different traffic with different properties such as skew, thus they need different number of counters. This allows *spatial statistical multiplexing*: A sketch may need more counters on switch A vs. B while another may need more on switch B vs. A.

3. SCREAM OVERVIEW

SCREAM provides sketch-based software-defined measurement with limited resources (Figure 2). It allows users

[1] 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 [14].

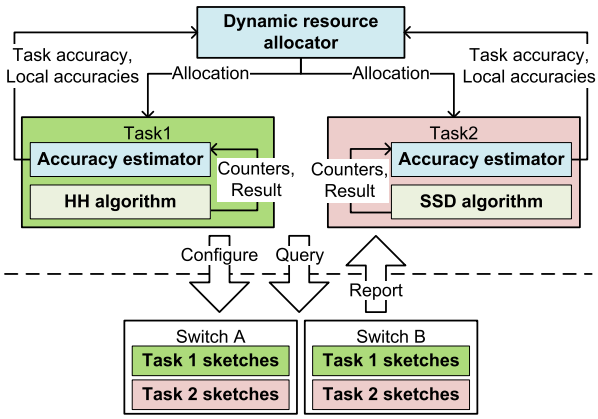


Figure 2: Resource allocation overview

to dynamically instantiate measurement tasks and specify a required accuracy. Specifically, the user instantiates a task by specifying its type, flow filter, its parameters and its accuracy bound. For example, she may instantiate a heavy hitter detection task on a five tuple flow filter <srcIP=10/8, dstIP=16.8/16,*,*,*> that reports sources sending traffic more than 10 Mbps with a precision (the fraction of detected items that are true heavy hitter) of at least 80%.

SCREAM can run multiple concurrent instances of different task types. Each task instance (henceforth, simply task) configures counters at switches and periodically queries counters from switches. Periodically, SCREAM distributes resources to each task on each switch based on the traffic observed at the switch to satisfy its requirement. As a result, tasks update the sketch parameters based on allocated resources and re-configure their counters at switches. In the following, we describe the two components of SCREAM (tasks and dynamic resource allocation).

Tasks: Sketches can support a diverse range of measurement tasks [40]. We consider three examples in this paper:

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 IPs contributing large volumes of traffic.

Hierarchical Heavy Hitter (HHH): Hierarchical heavy hitters (HHHs), used for detecting DDoS [36], 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 [15]. 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.

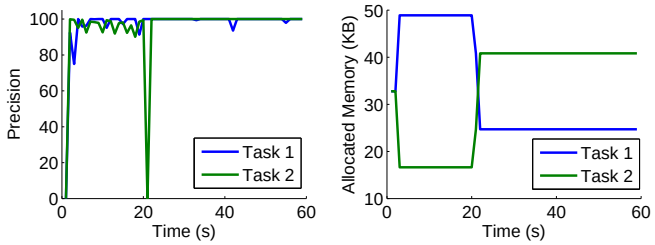
Super source and destination (SSD): A super source is a source IP that communicates with a more than a threshold number of *distinct* destination IP/port pairs. A super destination is defined in a similar way. SSDs are used for detecting worms, port-scans, P2P super nodes or DDoS targets.

Dynamic resource allocation: At the heart of SCREAM is its resource allocation mechanism. We use the *instantaneous accuracy* of a task as its feedback for an iterative allocation algorithm (Figure 2). Each task periodically shares its result and counters with an accuracy estimator module. The accuracy estimator estimates the accuracy of current results for resource allocation algorithm which in turn determines the number of counters for each sketch based on those estimates. Then tasks tune sketch parameters based on the number of assigned counters and re-configure switches. Thus, the resource allocation mechanism requires two components, a dynamic resource allocator and an accuracy estimator per task type.

SCREAM's resource allocator, borrowed from DREAM [33] (see below), requires tasks to provide a global task accuracy and local accuracies per switch, and runs parallel per-switch resource allocators that use the maximum of global and local accuracies as follows. If the accuracy estimate is below the specified accuracy bound ("poor" task), it receives more resources; these resources are taken away from "rich" tasks whose accuracy estimate is well above their bound. The allocator algorithm achieves fast but stable convergence by dynamically adapting the step size of resource exchange for each task. It also contains an admission control algorithm that rejects new tasks when necessary: without this, no task may receive sufficient resources. However, since resource demands for a task can change over time, resource overloading can occur even without the arrival of new tasks. In this case, the allocator assigns resources to poor tasks based on assigned priorities, and when it cannot, it may drop tasks.

We now illustrate how the resource allocator works using a simple example (Figure 3). 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 [3] starting from time 0. Figure 3a shows the estimated accuracy in terms of *precision* (the fraction of detected true HHs over detected ones) of tasks over time and Figure 3b 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.

An alternative approach to design an allocation mechanism would have been 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 [25]). However, quantifying the effect of traffic properties (e.g., skew parameters) on the accuracy is complicated, and dynamically estimating them may require significant resources [25].



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

Figure 3: Resource allocation example

Relationship to DREAM [33]. SCREAM is inspired by DREAM, our prior work on efficient and dynamic resource management for TCAM-based measurement tasks. In particular, SCREAM *deliberately* reuses the dynamic resource allocator (described above) proposed in DREAM [33]. This reuse has the advantage that it can enable a unified framework to support a variety of measurement tasks that uses either sketches or counters. Such a framework can simplify the configuration of measurement tasks and provide a unified interface for specifying these tasks and processing the results. Our next step in the near future is to develop such a unified framework.

However, SCREAM is different from DREAM in several ways because it is sketch-based instead of TCAM-based. First, by using sketches, SCREAM can support tasks that cannot be supported using TCAM-based counters. For example, in this paper, we implement the SSD task using a Count-Min sketch and a distinct counter instead of a volume counter (we leverage an existing technique [40, 17] for this, but adapt the design to provide unbiased results). Also, unlike flow-based counters, sketches do not need iterative reconfiguration. The iterative re-configuration is less accurate for time-varying traffic because it takes multiple measurement epochs to reconfigure counters (e.g., to zoom into 32 levels in IP prefix tree) [32].

Second, in SCREAM, different switches may be assigned different-sized sketches by the dynamic allocator because they see differing amounts of traffic. Combining different-sized sketches is non-trivial, and we present an approach to merge the counters from sketches of different sizes, together with a general algorithm to use Count-Min sketch in a prefix tree to run the three measurement task types (Section 4).

Finally, the primary enabler for dynamic resource allocation in SCREAM is the estimation of instantaneous accuracy of a task. We present a solution that does not assume an *a priori* traffic model or run parallel measurement tasks for each of the implemented task types (Section 5).

Generality: In this paper, we build three measurement tasks in SCREAM, which cover various important applications in data centers and ISP networks such as multi-path routing [6], anomaly detection [24], worm detection [39], P2P seed server detection [8], port scan [8], network provisioning, threshold-based accounting, and DDoS detection [36].

Moreover, although we have implemented measurement tasks based on Count-Min sketch and its variants, SCREAM can support other sketches for a different resource-accuracy

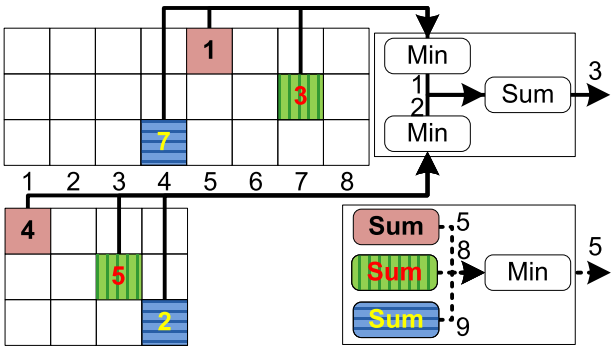


Figure 4: Merging two Count-Min sketches with different sizes

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 [9], 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 [13]. Kumar [27] 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 those sketches to SCREAM, we need to estimate their accuracy, which have left to future work.

4. DESIGN OF 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 [40], 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). However, a heavy hitter prefix may have traffic from multiple switches. One approach [5] 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 differ-

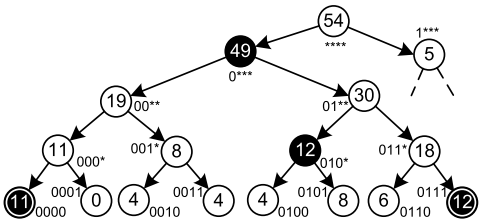


Figure 5: A prefix trie of source IPs where the number on each node shows the bandwidth used by the associated IP prefix in Mb in an epoch. With threshold 10, the nodes in double circles are heavy hitters and the nodes with shaded background are hierarchical heavy hitters.

ently, so each sketch may have an array of different widths and cannot be summed. For example, Figure 4 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 4 (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 4, 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).

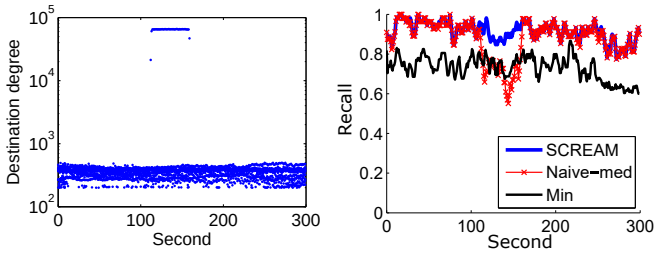
Hierarchical Heavy Hitter (HHH): Recall that HHHs are defined by the longest prefixes exceeding a certain threshold in aggregate volume after *excluding* any HHH descendants in the prefix tree. Figure 5 shows an example of a prefix tree for four bits. With a threshold of $\theta = 10Mb$, prefix 010* 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 [15]. 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 [15].

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 [17]. Therefore, each sketch has $w \times d$ distinct counters; we used the HyperLogLog [21] distinct counter because its space usage is near-optimal and it is easy to implement [23]. 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, and we pick the minimum, it is more likely to pick the one that under-approximates. Figure 6b shows the recall (detected fraction of true SSDs) of SSD detection given fixed resources in simulation over a CAIDA traffic trace [3]. 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 [17, 40, 22], 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 1 where A is the Count-Min sketch array of width w and T is the sum of distinct items of prefixes. Equation 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 [26].

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



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

Figure 6: 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 1 can result in missing SSDs. For example, Figure 6a 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 6b shows that the recall for an approach that uses Equation 1, named Naive-med, drops during the DDoS attack. These missed SSDs result from the fact that Equation 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

```
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 7: Approximate function for SSD

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 6b). This does not come at the cost of increased false positives, and SCREAM's precision is also high.

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 [21]: $M_{new,k} = max(M_{DC_1,k}, M_{DC_2,k})$ 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 [22]. Instead, we find corresponding distinct counters for each specific query in each row and merge them.

Figure 7 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 [21] (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).

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 [14]. 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 [2]. Note that to find HH/SSD IP prefixes that are not exact, we can start the tree from a level > 0.

```
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 8: Generic algorithm to create output

Figure 8 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 5, 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) [16, 7, 35]. At the core, all use multiple sketches and apply group testing to reverse the sketch, but their grouping is different. We use hierarchical grouping [16] 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 [35] with 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 [14, 16].

5. 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

[2] 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 $g > 1$.

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.

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. There are two key ideas in our accuracy estimator: (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 hitters: 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) 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) [16]. 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 3:

$$P(c_j^{real} > \theta) = P(c_j^{est} - e_{cm} > \theta) = 1 - P(e_{cm} \geq c_j^{est} - \theta) \quad (2)$$

$$P(c_j^{real} > \theta) > 1 - (\frac{T}{w(c_j^{est} - \theta)})^d \quad (3)$$

Unfortunately, as Figure 9 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

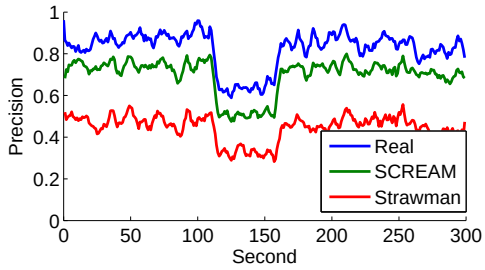


Figure 9: HH detection accuracy estimation for Count-Min sketch ($w = 340, d = 3$)

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 4.

$$P(c_j^{real} > \theta) > 1 - \prod_{i=1}^{d} \frac{T}{w(c_{i,j}^{est} - \theta)} \quad (4)$$

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[3]. 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 4, we can estimate p_j in Equation 5. However, we do not know $c_{k \in A}^{real}$ because of the over-approximations of counts in the sketch. Thus, as an estimate, we use $c_{k \in A}^{est}$ after reducing the average collision error of only small items from it. Figure 9 shows that our estimation based on Equation 5 is close to the real precision, even under traffic dynamics. In Section 6.4, 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.

[3] Cormode [18] 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.

$$P(c_j^{real} > \theta) > 1 - \prod_{i=1}^{d} \frac{T - \sum_{k \in A} c_k^{real}}{w(c_{i,j}^{est} - \sum_{k \in B_{i,j}} c_k^{est} - \theta)} \quad (5)$$

Multiple switches: As described in Section 3, our resource allocator estimates a global accuracy for the task, as well as a per-switch local accuracy [33]. 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) [11]. However, since the collision on a sketch is independent from the collision on another, we can replace Markov's bound with Chernoff's bound [11] to get a more accurate estimation of p_j (see our technical report [34]). We still use the Markov's inequality to estimate precision if a HH has traffic from a single switch.

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: If a detected HHH has no descendant HHH (e.g., 0000, 010*, 0111 in Figure 5), 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 5, 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 $\hat{e}_{D(j)}^{cm}$, for each descendant HHH of j, $D(j)$, is the upper bound on its error, $e_{D(j)}^{cm}$: $P(e_{D(j)}^{cm} < \hat{e}_{D(j)}^{cm}) > 0.1$. [4] We find this upper bound using Markov's inequality for HHHs originated from a single switch and Chernoff's bound otherwise. For example, Equation 6 derived from Equation 5 shows the maximum error that a descendant HHH at a single switch at level l can have while keeping $p_j \geq 0.1$.

$$e_{D(j)}^{cm} \leq \frac{T - \sum_{k \in A_l} c_k^{real}}{w \sqrt[d]{0.9}} \quad (6)$$

Multiple switches: For the global accuracy, we just replace the Markov inequalities in HH tasks and Equation 6 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 6) 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 [15] and no non-exact HHH prefix can have a size $\geq 2\theta$ [33], detecting a wrong HHH (low precision) will also be at the cost of missing a true HHH (low recall).

Super source or destination: 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 [22]. For a false positive SSD, j, that has counter $c_{i,j}^{est}$ for ith hash function, the error, $e_{i,j}^{dc} + e_{i,j}^{cm}$ must have been greater than $c_{i,j}^{est} - \theta'$ where θ' is computed based on the threshold θ and our version of Equation 1 (see our technical report [34]). If the SSD has $d' \leq d$ such counters (remember we choose median instead of minimum), p_j is computed using Equation 7. We can compute p_j, based on the *individual* error distributions of Count-Min sketch and the distinct counter (see formulations in the technical report [34]). 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 [21]. The collision error because of Count-Min sketch is also bounded using Markov inequality as before.

$$P(c_j^{real} > \theta') = 1 - \prod_{i=1}^{d'} P(e_{i,j}^{dc} + e_{i,j}^{cm} \geq c_{i,j}^{est} - \theta') \quad (7)$$

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 [22]. 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 mul-

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

tiple 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.

6. EVALUATION

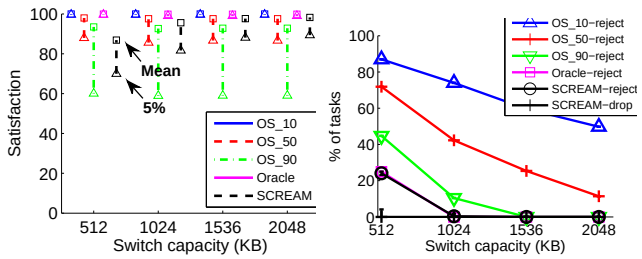
In this section, we use simulations driven by realistic traffic traces to show that SCREAM performs significantly better than OpenSketch, and is comparable to an oracle both on a single switch and on multiple switches.

6.1 Evaluation setting

Simulator: Our event-based simulator runs sketches on 8 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 every second. The reject and drop parameters of the resource allocator are set the same as DREAM [33]. The resource allocator is scalable to more switches, and the average number of switches that a task has traffic from is the dominating factor for controller overhead [33]. Therefore, we make each task to have traffic from all 8 switches and put the evaluation for more switches for future work.

Tasks and traffic: Our workload consists of three types of tasks: HHs, HHHs and SSDs. In a span of 20 minutes, 256 tasks with randomly selected types appear according to a Poisson process. The threshold for HH and HHH tasks is 8 Mbps and the threshold for SSD tasks is 200 sources per destination IP. We choose 80% as the accuracy bound for all tasks since we have empirically observed that to be the point at which additional resources provide diminishing returns in accuracy. Each task runs for 5 minutes on a part of traffic specified by a random /12 prefix. We use a 2-hour CAIDA packet trace [3] from a 10 Gbps link with an average of 2 Gbps load. 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. For scenarios with multiple switches, we assign /16 prefixes to each switch randomly and replay the traffic of a task that matches the prefix on that switch. This means that each task has traffic from all 8 switches. In a data center, SCREAM would monitor traffic on the source switches of traffic (the ToRs), thus network topology is irrelevant to our evaluation.

Evaluation metrics: The *satisfaction* rate is the percentage of task lifetime for which accuracy is above the bound. We show the average and 5^{th}% for this metric over all tasks. The 5^{th}% value of 60 means that 95% of tasks had an accuracy above the bound for more than 60% of their lifetime: it is important as a resource allocator must keep all tasks accurate, not just on average. The *drop* ratio shows the percentage of tasks that the SCREAM resource allocator drops to lower the load if it cannot satisfy accepted tasks, and the *rejection* ratio shows the ratio of tasks that had been rejected at instantiation in each algorithm. These metrics are important because



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

Figure 10: Comparison for OpenSketch (different relative error%) and the oracle at a single switch

a scheme can trivially satisfy tasks by rejecting or dropping a large fraction of them.

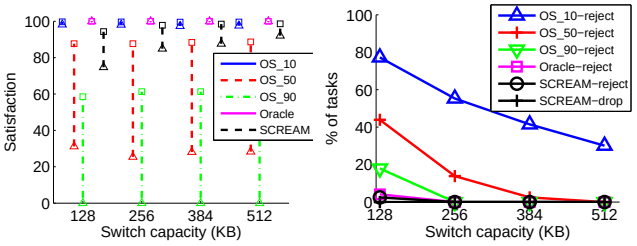
Comparison with OpenSketch: OpenSketch [40] allocates resources to a task based on worst-case traffic to reach a given relative error at a single switch. To bound the error to x% of the threshold θ on a HH/HHH detection task that has an average traffic of T, it configures a Count-Min sketch with $w = \frac{eT}{x\theta}$. 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 [40].

At task arrival, OpenSketch finds the required amount of resources for the relative error guarantee and reserves its resources if there is 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 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 ($T/8$).

Comparison with 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.

6.2 Performance at a single switch

SCREAM supports more accurate tasks than OpenSketch.



(a) Satisfaction (b) Rejection and drop

Figure 11: Comparison for OpenSketch (different relative error%) and the oracle at multiple switches

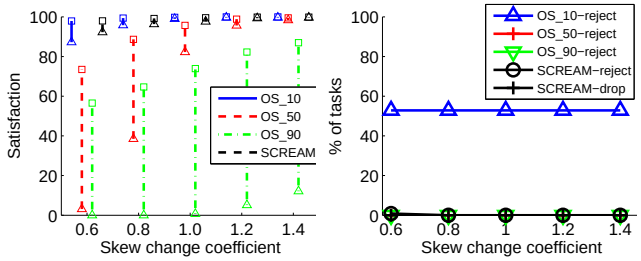
Figure 10 compares SCREAM to OpenSketch with three different relative error percentages for a combination of different types of tasks over different switch memory sizes [5]. We ran the experiment 5 times with different task arrival patterns and show the error bars in Figure 10b which are very tight. 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 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 10). 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.

6.3 Performance on multiple switches

Figure 11 shows that SCREAM can keep all task types satisfied. 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. Our experiments for each individual task type, described in a technical report [34], also show the superiority of SCREAM over OpenSketch.

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



(a) Satisfaction (b) Rejection and drop

Figure 12: Changing skew for HH detection at multiple switches with capacity 64 KB

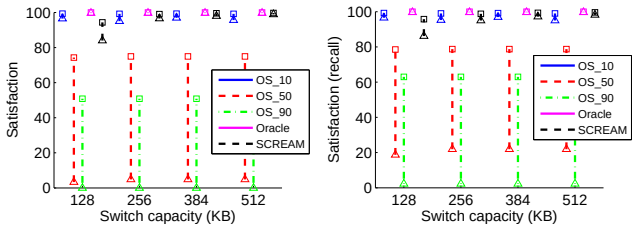
Like the single switch case, SCREAM can also achieve a satisfaction comparable to the oracle for multiple switches (Figure 11a). In Figure 11b, SCREAM has a lower rejection rate than oracle for small switches but still has no drop. This is because SCREAM does not know the future resource requirements 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 12 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 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).

6.4 Accuracy estimation

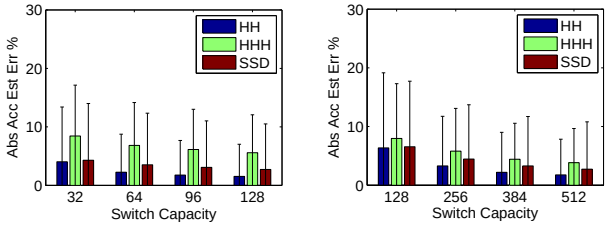
SCREAM's superior performance requires low accuracy estimation error. Our experiments show that our accuracy estimation has within 5% error on average. Although we define accuracy based on precision, SCREAM achieves high recall in most cases.

SCREAM accuracy estimation has low errors. We calculated the accuracy estimation error (the percentage difference of the estimated accuracy and the real accuracy) of



(a) based on precision **(b) based on recall**

Figure 13: HHH satisfaction on multiple switches



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

Figure 14: Accuracy estimation error

tasks for the single switch and multiple switches cases. Figure 14 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 5), 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 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 13b 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 5). Hence, the curves for satisfaction based on recall (Figure 13b) are similar to the curves for satisfaction based on precision (Figure 13a). As mentioned in Section 5, 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.

7. RELATED WORK

Sketch-based measurement on individual tasks: 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 [14], HHH [15], and SSD [17]. Other works [5, 22] 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.

Resource allocation for measurement tasks: Most resource allocation solutions focus on sampling-based measurement. CSAMP [37] uses consistent sampling to distribute flow measurement on multiple switches for a single measurement task and aims at maximizing the flow coverage. Volley [31] uses a sampling-based approach to monitor state changes in the network, with the goal of minimizing the number of sampling. Payless [12] 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 [40] 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. On the other hand, 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.

DREAM [33] focuses on flow-based counters in TCAM, and dynamically allocates TCAM resources to multiple measurement tasks to achieve their given accuracy bound. DREAM develops accuracy estimators for TCAM-based zoom-in/out algorithms, and its paper's evaluations show that DREAM is better than simple *task-type agnostic* schemes such as equal TCAM allocation. In contrast, SCREAM explores the accuracy estimation for sketch-based tasks, where the sketch counters are not accurate compared to TCAM counters because of random hash collisions. We show that SCREAM supports 2 times more accurate tasks than a *task-type aware* allocation, OpenSketch [40], and has comparable performance as an oracle that knows future task requirements.

8. CONCLUSION

Sketches are a promising technology for network measurement because they require lower resources and cost with higher accuracy compared to flow-based counters. To support sketches in Software-defined Measurement, we design and implement SCREAM, a system that dynamically allocates resources to many sketch-based measurement tasks and ensures a user-specified minimum accuracy. SCREAM estimates 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 supports more accurate tasks than current practice, OpenSketch [40]. In the future, we plan to add more sketch-based tasks such as flow-size distribution and entropy estimation.

9. ACKNOWLEDGEMENTS

We thank our shepherd Laurent Mathy and CoNEXT reviewers for their helpful feedback. This work is supported in part by the NSF grants CNS-1423505, CNS-1453662, and a Google faculty research award.

10. REFERENCES

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

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

[3] CAIDA Anonymized Internet Traces 2014. http://www.caida.org/data/passive/passive_2014_dataset.xml.

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

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

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

[7] T. Bu, J. Cao, A. Chen, and P. P. C. Lee. Sequential Hashing: A Flexible Approach for Unveiling Significant Patterns in High Speed Networks. *Computer Networks*, 54(18):3309–3326, 2010.

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

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

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

[11] H. Chernoff. A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations. *The Annals of Mathematical Statistics*, 23(4), 1952.

[12] 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.

[13] G. Cormode. Sketch Techniques for Approximate Query Processing. In P. H. G. Cormode, M. Garofalakis and C. Jermaine, editors, *Synposes for Approximate Query Processing: Samples, Histograms, Wavelets and Sketches, Foundations and Trends in Databases*. NOW publishers, 2011.

[14] G. Cormode and M. Hadjieleftheriou. Finding Frequent Items in Data Streams. In *VLDB*, 2008.

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

[16] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms*, 55(1), 2005.

[17] G. Cormode and S. Muthukrishnan. Space Efficient Mining of Multigraph Streams. In *PODS*, 2005.

[18] G. Cormode and S. Muthukrishnan. Summarizing and Mining Skewed Data Streams. In *SIAM International Conference on Data Mining*, 2005.

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

[20] P. Flajolet and G. N. Martin. Probabilistic Counting Algorithms for Data Base Applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.

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

[22] M. Hadjieleftheriou, J. W. Byers, and J. Kollios. Robust Sketching and Aggregation of Distributed Data Streams. Technical report, Boston University Computer Science Department, 2005.

[23] 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.

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

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

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

[27] 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.

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

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

[30] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani. Counter braids: A Novel Counter Architecture for Per-flow Measurement. *SIGMETRICS Performance Evaluation Review*, 36(1):121–132, 2008.

[31] S. Meng, A. K. Iyengar, I. M. Rouvellou, and L. Liu. Volley: Violation Likelihood Based State Monitoring for Dataceners. *ICDCS*, 2013.

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

[33] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *SIGCOMM*, 2014.

[34] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. SCREAM: Sketch Resource Allocation for Software-defined Measurement. Technical Report 15-960, Computer Science, USC, 2015. http://www.cs.usc.edu/assets/007/96891.pdf.

[35] R. Schweller, A. Gupta, E. Parsons, and Y. Chen. Reversible Sketches for Efficient and Accurate Change Detection over Network Data Streams. In *IMC*, 2004.

[36] 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.

[37] 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.

[38] A. Vahdat. Enter the Andromeda zone - Google Cloud Platform's Latest Networking Stack. http://goo.gl/smN6W0.

[39] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum. New Streaming Algorithms for Fast Detection of Superspreaders. In *NDSS*, 2005.

[40] M. Yu, L. Jose, and R. Miao. Software Defined Traffic Measurement with OpenSketch. In *NSDI*, 2013.

[41] Y. Zhang. An Adaptive Flow Counting Method for Anomaly Detection in SDN. In *CoNEXT*, 2013.