Online Measurement of Large Traffic Aggregates on Commodity Switches

Lavanya Jose, Minlan Yu, and Jennifer Rexford
Princeton University; Princeton, NJ

Abstract

Traffic measurement plays an important role in many network-management tasks, such as anomaly detection and traffic engineering. However, existing solutions either rely on custom hardware designed for a specific task, or introduce a high overhead for data collection and analysis. Instead, we argue that a practical traffic-measurement solution should run on commodity network elements, support a range of measurement tasks, and provide accurate results with low overhead. Inspired by the capabilities of OpenFlow switches, we explore a measurement framework where switches match packets against a small collection of rules and update traffic counters for the highest-priority match. A separate controller can read the counters and dynamically tune the rules to quickly "drill down" to identify large traffic aggregates. As the first step towards designing measurement algorithms for this framework, we design and evaluate a hierarchical heavy hitters algorithm that identifies large traffic aggregates, while striking a good balance between measurement accuracy and switch overhead.

1 Introduction

Network-management tasks, such as anomaly detection and traffic engineering, rely on timely and accurate measurements of network traffic. Unfortunately, *online* traffic monitoring is typically expensive, relying on either heavy-weight collection of per-flow statistics using NetFlow [1] or customized hardware designed to measure specific statistics. Instead, we argue that a practical solution for network measurement should: (i) run directly on *commodity network elements*, (ii) impose *minimal overhead* on the network and the collection system, (iii) be *generic* enough to support many measurement tasks [12], and (iv) still provide *accurate* measurement results.

Existing measurement solutions do not satisfy all of these goals. While NetFlow runs directly on the routers and offers fine-grain measurements, maintaining and exporting per-flow state introduces considerable overhead. As a result, networks typically perform aggressive sampling (*e.g.*, 1% or 0.1% of packets), reducing the accuracy. In contrast, researchers have proposed ways to compute specific statistics (*e.g.*, identifying heavy-hitter traffic or estimating the number of active flows) [6, 14, 4, 8, 15], but they each rely on custom hardware. While sharing our goal of a generic monitoring platform, ProgME [13] performs multiple passes over the same packets; this has scalability limitations and cannot capitalize on commodity hardware.

In this paper, we explore a measurement framework that exploits the capabilities of the OpenFlow switches [10] available from multiple vendors (*e.g.*, HP, NEC, and Quanta) and deployed in several campus and backbone networks. OpenFlow switches can count traffic based on *wildcard* rules that match on bits in the packet header, including IP addresses and TCP/UDP port numbers. Wildcard rules fit naturally with the Ternary Content Addressable Memory (TCAM) available in many switches. When processing a packet, the switch identifies the matching rules, picks the rule with the highest priority, updates the associated counter, and performs some action (*e.g.*, dropping or forwarding the packet). A separate controller can read the counters and install new rules. Although OpenFlow switches can direct data packets to the controller, we intentionally restrict ourselves to a setting where *all* data packets are handled by the switches to reduce controller overhead.

Unlike custom streaming algorithms that modify complex data structures on demand, our measurement framework relies on simple match-and-count rules where the controller only adjusts rules periodically. We argue that this simple framework is sufficient for many measurement tasks, such as DoS detection and "heavy hitter" identification. However, due to the switch constraints (*i.e.*, the limited number of rules and overhead for updating rules), algorithms built on our framework sacri-

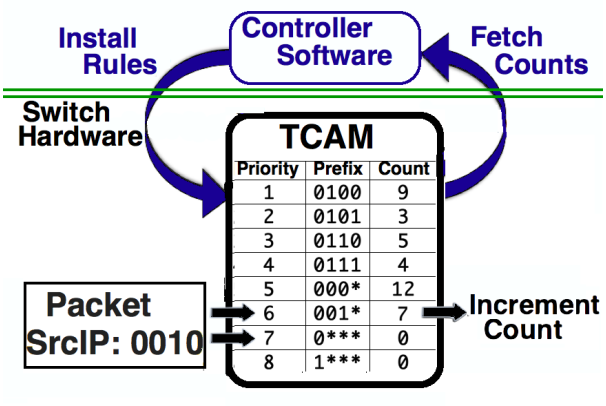

Figure 1: Illustration of the measurement framework.

fice *accuracy* in exchange for a low-overhead, commodity solution. For example, an algorithm can detect large traffic aggregates by iteratively adjusting the wildcard rules—leading to a short delay in detecting the traffic, while producing intermediate useful results at a somewhat coarser level of aggregation. Our broader research goal is to understand the algorithmic trade-offs in this measurement framework, and to create useful algorithms that can run on commodity OpenFlow switches.

To explore our measurement framework, we consider a specific measurement problem. The *hierarchical heavy hitters* (HHH) problem [7, 14, 6] identifies the large traffic aggregates, where the aggregates are tailored to the traffic (*e.g.*, a single source IP *address* responsible for 11% of the traffic, and a particular source IP *prefix* responsible for another 12%). We selected this problem because it provides useful measurement data, has been studied in previous work on custom data structures, and introduces a clear trade-off between hardware complexity and measurement accuracy. We present a new HHH algorithm for our measurement framework. Our experiments with packet traces show that our algorithm achieves relatively high accuracy with low overhead.

2 Traffic Measurement Framework

In this section we describe a measurement framework based on commodity OpenFlow switches. We first discuss the practical constraints of the switches and then show how to support a wide range of measurement tasks.

2.1 Commodity Switch Constraints

The measurement framework contains two parts, as shown in Figure 1. In the data plane, the TCAM matches packets with wildcard rules at line speed; in the control plane, the controller reads counters and installs rules.

Data plane: Matching rules and counting packets at line rate. For each incoming packet, the switch compares the packet header simultaneously against a collection of monitoring rules, picks the matching rule with the highest priority, and increments its associated counter. Since TCAMs are expensive and power hungry, a switch has a limit N on the number of rules we can use for traffic monitoring. We expect N to be a very small fraction of the total rules available since the majority of the rules will be used for other data-plane functions, such as access control and packet forwarding.

The emerging OpenFlow 1.1 specification [2] supports multiple stages of rules for different purposes, allowing us to have a separate stage for traffic monitoring, or to support multiple stages of monitoring (*e.g.*, separately measuring HHHes on both the source and destination IP addresses). Switches supporting only one stage of rules would use the "cross-product" of the rules needed for different purposes; for example, we may need $N = 400$ rules if we have 20 rules for identifying HHH source addresses, and 20 rules for destination addresses.

Control plane: A controller adapting rules at fixed intervals. The controller can run directly on the switch or on a separate machine managing the entire network. The controller reads the counters from the TCAM rules at a fixed *measurement interval* M, analyzes the counters, and generates statistics to report to the network operators. The controller also dynamically adapts the rules based on the counter values from previous measurement intervals. In practice, M is limited by how quickly the controller can read the counters from the TCAM and generate the rules for the next interval. In our experiments, M ranges from seconds to minutes.

Our proposed measurement framework relies only on features available in OpenFlow switches. In fact, for efficiency and simplicity, we restrict how the controller interacts with the switches. To reduce the controller overhead, we do *not* allow the switches to direct data packets to the controller, relegating the controller to installing rules and reading counters. For simplicity, we consider only rules that monitor traffic, rather than other actions such as forwarding or dropping packets. In addition, we assume the controller only adapts rules at a fixed interval; in practice, the controller can read different counters at different times to spread the load over the interval.

2.2 Example Measurement Tasks

Our measurement framework is useful for many measurement tasks that require (i) constructing a baseline understanding of "normal" traffic (by selectively monitoring different portions of the traffic over time, and combining these measurements into a model of normal behavior) and (ii) quickly pinpointing large traffic aggregates (by "drilling down" into a portion of the traffic us-

ing finer-grain rules). We give several example measurement tasks that can use our measurement model:

Measuring normal traffic pattern: Network operators often need to understand the basic traffic patterns in their network, *e.g.*, the clients that typically communicate with a web server, the average traffic volume directed to each Web site, etc. A controller application could install rules that measure a subset of the client addresses at each interval, cycling through all the source address blocks to identify the common senders and their traffic volumes.

Identifying large traffic changes: Identifying significant traffic changes is the key challenge for anomaly detection. Based on the normal traffic pattern, the controller can install a few rules to monitor different groups of traffic (*e.g.*, based on address or port number). If the traffic volume for one group has changed significantly, the controller can expand the rules to ultimately find the specific senders responsible for the traffic change.

Pinpointing denial-of-service (DoS) attacks: To detect DoS attacks, the controller could first observe a large increase in traffic at a Web server, and then collect finer-grain measurements to distinguish a "flash crowd" from an attack. For example, the controller could *expand* the rules to identify the offending source IP addresses, while *collapsing* other rules to ensure enough monitoring resources are available. The controller can also install rules on different switches to collect more measurement data, and to simultaneously identify traffic from multiple senders close to where the attacks enter the network.

Because of practical constraints on commodity switches, our framework cannot solve all measurement problems effectively. For example, similar to NetFlow, we cannot solve problems that require deep packet inspection at line rate. In addition, like any measurement solution that samples or aggregates traffic, our framework cannot easily detect low-volume traffic anomalies (*e.g.*, a "ping of death" attack). Detecting short-lived attacks is also difficult, because the controller only reads counters and updates rules periodically. An attacker could stop an attack before the controller had enough time to "drill down" to identify the culprit; still, this limits the severity of attacks, since attackers must constrain their behavior to evade detection. Ultimately, detecting and diagnosing low-rate or short-lived attacks requires other monitoring techniques, such as commonly-used intrusion detection systems. Instead, we focus on monitoring tasks that detect and diagnose large traffic aggregates.

3 Identifying Hierarchical Heavy Hitters

In this section, we take the hierarchical heavy hitter problem as an example to show the use of the measurement framework and understand the tradeoff between hardware complexity and measurement accuracy. We formulate the Hierarchical Heavy Hitters (HHH) problem using our measurement framework. Then, we discuss the intuition behind how we minimize the number of TCAM entries we use to identify HHHes. Based on these insights, we present algorithms that report HHH and adapt the monitoring rules to traffic changes. Finally, our evaluation with realistic packet traces shows that we can achieve high accuracy with low overhead.

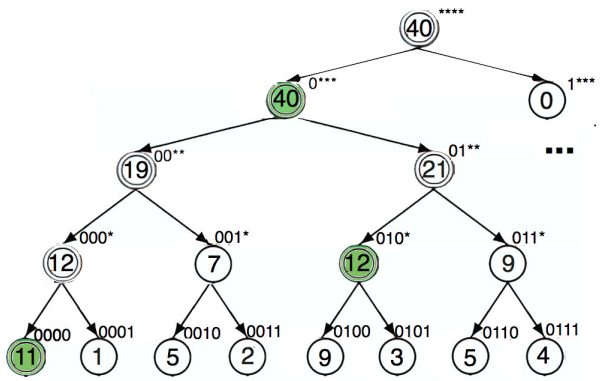

Figure 2: A trie (prefix tree) of source IP addresses, where each node contains the volume of traffic sent by that IP prefix during interval i. The nodes in double circles are heavy hitters and those with shaded circles are both heavy hitters and hierarchical heavy hitters. Before interval i, we assume there are eight rules in the TCAM as shown in Figure 1.

3.1 Hierarchical Heavy Hitters Problem

Definition of hierarchical heavy hitters: Network operators need to identify the important traffic in the network. One simple way to define "important" traffic is the *Heavy Hitters* along some dimension, *e.g.*, the source IP prefixes that contribute more than a fraction T of link capacity over the past p packets or q seconds. For example, Figure 2 illustrates the traffic volume for each source IP prefix, represented as a *trie* (or *prefix tree*) where the leaves correspond to IP addresses and the other nodes are prefixes that aggregate the traffic of their descendants. Applying a threshold of 10% to the leaves would only report node 0000, ignoring the large contribution from the prefix 010*. However, reporting *all* large prefixes would report too much (redundant) information, such as 000* and 01** that are large only because of a large child.

A more concise way to summarize the traffic is to report the *Hierarchical* Heavy Hitters (HHH) [7, 14, 6], *i.e.*, the longest IP prefixes that contribute a large amount of traffic (*i.e.*, at least a fraction T of link capacity), after excluding any HHH descendants. In Figure 2, the 010* node is an HHH because the two children *collectively* contribute more than 10% of the link capacity, but each child individually contributes *less* than the threshold. In

contrast, node 000* is *not* an HHH, despite contributing 12% of the link capacity, because nearly all of this traffic comes from the descendant 0000 that is an HHH in its own right. Yet, the node 0*** *is* an HHH, because its non-HHH descendants 0001, 001*, and 01** collectively contribute more than 10% of the link capacity. A network has at most $1/T$ HHH nodes (*e.g.*, a threshold of $T = 0.10$ leads to at most 10 HHH nodes).

Prior solutions: A simple solution to the HHH problem is to run an *offline* algorithm over the traffic counts for all leaf nodes in the trie. However, this approach is slow and has high measurement overhead. Instead, an online algorithm can identify the "important" prefixes and the associated traffic. Previous work assumed custom hardware that is not available in commodity switches. These algorithms adjust which prefixes to monitor, in one of two ways: The first way is to adjust the prefixes based on each packet [13, 4, 6, 14], at the expense of custom hardware. The other way is to adjust the prefixes at a periodic interval. Similar to our model, the authors in [9] assume the switch hardware applies simple rules that count the packets matching each prefix, where the rules change only periodically (*e.g.*, seconds) rather than for each packet. However, the work in [9] focuses on the (non-hierarchical) *heavy hitter* problem; although the HHHes can be computed from the HHes, this would require far more rules than solving the HHH problem directly. As a result, must use a large number of rules to track *all* heavy hitter prefixes. They also allow each packet to match *multiple* rules and increment multiple counters. Given the limited TCAM space in the switch hardware, they must be conservative in expanding the rules; for example, to expand one prefix, they may first need to collapse other prefixes. Recent work [11] designs new algorithms for the HHH problem but assumes more custom operations in the TCAM than this paper.

HHH problem in our framework: In our measurement framework, the HHH problem can be reformulated as: given the maximum number of rules N, the measurement interval M, and a threshold T, find the traffic aggregates that consume at least T of the link capacity in each interval. A solution generates the rules to install in the next interval, and computes the HHHes from the counters collected in the previous interval. Since the rules are not updated on each packet arrival, the algorithm cannot always produce an accurate report of the HHHes and their traffic counts. Previous work considers two metrics—recall and precision—to quantify accuracy [5]. *Recall* is the total number of true HHHes reported by our algorithm over the real number of HHHes reported by an offline algorithm; the higher the recall, the lower the false negatives. *Precision* is the total number of true HHHes reported over the total number of answers reported; the higher the precision, the lower the false positives.

3.2 Minimizing Number of TCAM Entries

The monitoring rules for the HHH problem can be easily implemented with TCAMs. If we knew the HHHes for a given interval, we could simply monitor their counts by installing a wildcard prefix rule for each one in the TCAM. We can set higher priority to those rules with longer prefixes. If one packet matches multiple rules, the switch selects the rule with highest priority and thus only increments the counters for the longest prefix. For example, in Figure 1, we use a group of eight wildcard rules as shown in the TCAM to monitor the traffic illustrated in Figure 2. A packet with source IP address 0010 matches two prefixes 001* and 0***, but the switch only increments the counter for the longer prefix 001*.

The number of rules required in the TCAM is inversely proportional to T, which is the threshold we use to identify HHHes. This is because there are at most $1/T$ HHHes whose traffic is more than a fraction T of the link capacity. However, we do not know the HHHes in advance because the traffic keeps changing, and we do not have enough rules to monitor all the prefixes. To reduce the number of rules we monitor and discover new HHHes when they appear, we have three approaches:

Always monitor the root: We make sure we always monitor the root in every interval.[1] The root's count will give us the number of packets that do not match any other rule. If this number exceeds the threshold, then we know that there is at least one HHH that is not being counted by any other rule. We may have to install new rules in the next interval to narrow down on the HHH.

Monitor the children of HHH prefixes with at most $2/T$ rules: This will alert us when there is a more-specific HHH being counted by the prefix. For example, suppose we installed a single rule to monitor the HHH prefix 010* (in Figure 2) during the next interval. If the children's counts change from 9 and 3 to 10 and 2, respectively, then 0100 becomes a new HHH; however, the counter for 010* would still indicate 12, suggesting no change. By monitoring each child of the HHH prefix, we can identify when a child's counter exceeds the threshold. We need at most $2/T$ rules to identify all the HHHes that exceed fraction T of the link capacity.

Monitor the parents of HHH prefixes with the extra rules: Sometimes a monitored prefix drops below threshold, but the prefix and its (non-HHH) sibling collectively exceed the threshold, making their parent a new HHH. From the installed rules (Figure 1) for the traffic

[1] To be consistent with the next solution (monitor the children of the HHHes), we actually monitor the root's two children (*e.g.*, , 0*** and 1*** in Figure 1).

in Figure 2, we see that one child 000* of the monitored prefix 00** exceeds the threshold, while the other child 001* with a count of 7 does not. We can no longer report 00** as an HHH. However its parent 0*** has become an HHH, because—after excluding its HHH descendants 0000 and 010*—its count exceeds the threshold. Alternatively, we could simply add the non-HHH child's count to the nearest upstream prefix and drill down until we reach the HHH parent of the monitored prefix. However, with n levels between the root and the new HHH parent, identifying the new HHH can take n intervals. In contrast, if we know the sibling's count, we can determine immediately if the parent has become an HHH.

3.3 Adaptive Monitoring Algorithms

We first describe the basic algorithm that only monitors the children of HHHes. Then we present an enhanced algorithm that leverages the remaining rules in the TCAM to monitor the parents of HHHes.

Basic algorithm: We start by monitoring the root (*e.g.*, installing two rules to monitor the children of the root 0*** and 1*** in Figure 2). At each measurement interval, our algorithm reads the counters from the TCAM, reports the identified HHHes, and then adapts all the monitoring rules to capture new HHHes in the next interval. For each prefix we monitor, we install the rules for its two children. For each prefix p whose children were monitored in the previous interval i, we report if p or its children are HHHes and decide which prefixes to monitor in the next interval $i+1$. There are three conditions:

(1) Keep the rules: If the count of p is larger than fraction T of the link capacity but none of its children exceed the threshold, then we report p as an HHH and continue to monitor p's children in the next interval.

(2) Expand the rules: If one or both the children of p exceed the threshold, p is no longer an HHH in this interval. For the child k that exceeds the threshold, we report it as an HHH, and monitor it (install the rules for its two children) in the next interval. For the child k' whose count does not exceed threshold, we add its count to the nearest upstream prefix in the TCAM. For example, suppose we have the eight rules shown in Figure 2 to monitor the four prefixes 010*, 011*, 00** and **** at interval i. During interval i, the monitored prefix 00**'s child 000* exceeds the threshold, so we install rules for its children 0000 and 0001 in the next interval $i+1$. The other child 001* with count 7 is still below the threshold, so we add its count to nearest upstream prefix in the TCAM 0***.

(3) Collapse the rules: If the number of packets matching p is below the threshold, we do not report p as an HHH in this interval or monitor it during the next interval. Instead, we just add p's count to the nearest upstream prefix in the TCAM. For example in Figure 2, since the monitored prefix 011* no longer exceeds the threshold, its count 9 will instead be added to 0***.

Enhanced algorithm: The basic algorithm may not use all the $2/T$ rules in the TCAM, because we may have fewer than $2/T$ HHHes if some HHH takes far more than T fraction or the link is not heavily loaded. We can use the s spare rules to monitor the parents of existing HHHes, to detect cases where an existing HHH drops below the threshold and its parent becomes a new HHH. We first sort all the existing HHHes in decreasing order of their counts. Assuming that the prefixes with the lowest traffic volume are most likely to drop below the threshold, we monitor the parents of the bottom s HHHes.[2]

	5% Threshold		10% Threshold	
	Basic	Enhanced	Basic	Enhanced
Precision	78-87%	79-88%	85-93%	90-95%
Recall	76-86%	77-87%	81-90%	88-94%

Table 1: The precision and recall for HHH algorithms.

3.4 Evaluation with Realistic Packet Trace

To evaluate the basic and enhanced algorithms for identifying HHHes, we used a one-hour packet trace collected at a backbone link of a Tier-1 ISP in San Jose, CA, at 12 pm on Sept. 17, 2009 [3]. We built a simulator that runs our algorithm in every measurement interval to report HHHes, and update the rules. We evaluated different interval values M ranging from 1 to 60 seconds. The HHHes for each interval were aggregated at different prefixes based on the 32-bit source IP address field. We set the threshold T to be 0.1 and 0.05, as the fraction of the maximum amount of traffic in all the measurement intervals. We set the total number of rules K to $2/T$, which is 20 and 10 when $T = 0.1$ and 0.05, respectively.

We compared our reported HHH to the list of exact HHH that we get with an offline algorithm in each measurement interval. We counted the reported HHH as correct only if it was in the list of actual HHHes, and its reported count was identical.[3] We measured the precision and recall as defined in Section 3 for the different values of T and M. The running average of precision and recall in all the experiments converged within 200 intervals. Table 1 presents the ranges of precision and recall we get from the basic and enhanced algorithms for the different M. We made the following observations:

[2] In fact, we install rules to monitor these HHHes' siblings in the next interval, so that we can learn the counts of these HHHes' parents by combining the counts of the HHHes and their siblings.

[3] We also evaluated the metric that a reported HHH is viewed as correct when it was in the list of the exact HHH but its reported count is different. The precision and recall improve by at most 1%.

(1) Precision is always better than recall: We observed that the precision is better than recall for any given interval. This is because missing the HHH prefix hurts our recall, and reporting a non-HHH prefix as an HHH affects our precision. We report a prefix incorrectly whenever we aggregate an HHH at the wrong level, so we have at least as many missed HHHes as incorrect prefixes. For example, in Figure 2, if our algorithm misses the HHH 0000 and instead reports the more general prefix 000*, we reduce both the recall and the precision. If both 0000 and 0001 are true HHHes, but we report 000*, then we miss two HHHes and report one wrong HHH. As a result, we always achieve better precision than recall.

(2) Higher accuracy for 10% threshold than 5%: Our algorithm performs better for the higher threshold. This is because prefixes with more traffic (above 10% threshold) are more stable in traffic volume, while prefixes with less traffic (around 5%) are much more volatile—leading to more changes in HHHes from one interval to the next.

(3) Monitoring parents of HHHes with a few more rules improves both the precision and recall: There is an improvement of between 1 to 5% for the enhanced algorithm, compared with the basic algorithm. This is because the enhanced algorithm makes use of the extra rules in the TCAM to monitor the parents of HHHes.

(4) The accuracy depends on the traffic changes in intervals: The precision and recall vary with the interval size (as shown by the ranges in Table 1). This is because the accuracy closely relates to the amount of traffic changes in the packet trace and the resulting HHH changes for different interval values. We did not see a clear trend in the relationship between accuracy and interval size as the size increased from 1 to 60 seconds. However, the accuracy dropped a lot for measuring intervals as long as 5 minutes. Our accuracy is better for shorter intervals because the traffic is stable on this scale and the set of HHHes does not change much from one interval to the next (*i.e.*, not many entries need to be updated after each interval). As a result, we can adapt the rules more frequently and find the new HHHes faster.

(5) Quantifying accuracy in different ways: Even when our algorithms report an incorrect HHH, or miss a true HHH, the measurement results could still be useful to the network operator. For example, when a new HHH 0000 appears, we may take several measurement intervals to locate it. In the meantime, we may report a coarser-grained prefix 000*, which is still useful for the operators. We have defined a new metric for the relaxed accuracy, which is omitted due to lack of space.

4 Conclusion and Future Work

We proposed a practical measurement model based on emerging OpenFlow switches. The proposed model runs on commodity hardware, is generic across different measurement tasks, and achieves high accuracy with low overhead. We studied the HHH problem to understand the tradeoff between accuracy and overhead, and proposed new HHH algorithms that are useful for network operators managing networks of OpenFlow switches.

We plan to extend our HHH algorithms to the multi-dimensional HHH problem (*e.g.*, considering both source and destination addresses). Furthermore, we will explore ways to combine measurement results from different switches and understand the overhead on them as we extend our solutions to other measurement problems such as detecting DoS attacks and traffic changes.

References

[1] `http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html`.

[2] `http://openflow.org/documents/openflow1.1-allmerged-draft1.pdf`.

[3] CAIDA packet trace. `http://www.caida.org/data/monitors/passive-equinix-sanjose.xml`.

[4] N. Bandi, A. Metwally, D. Agrawal, and A. E. Abbadi. Fast data stream algorithms using associative memories. In *ACM SIGCOMM*, 2007.

[5] G. Cormode and M. Hadjieleftheriou. Methods for finding frequent items in data streams. *VLDB Journal*, 2010.

[6] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding hierarchical heavy hitters in streaming data. *ACM Transactions on Knowledge Discovery from Data*, Jan. 2008.

[7] C. Estan, S. Savage, and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *ACM SIGCOMM*, 2003.

[8] C. Estan and G. Varghese. New directions in traffic measurement and accounting. *SIGCOMM*, 2002.

[9] N. Kammenhuber and L. Kencl. Efficient statistics gathering from tree-search methods in packet processing systems. *IEEE ICC*, 2005.

[10] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *ACM Computer Communication Review*, Apr. 2008.

[11] M. Mitzenmacher, T. Steinke, and J. Thaler. Hierarchical heavy hitters with the space saving algorithm. *arXiv:1102.5540*, 2011.

[12] G. Varghese and C. Estan. The measurement manifesto. *HotNets*, 2003.

[13] L. Yuan, C.-N. Chuah, and P. Mohapatra. ProgME: Towards programmable network measurement. In *ACM SIGCOMM*, 2007.

[14] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund. Online identification of hierarchical heavy hitters: Algorithms, evaluation, and applications. In *Internet Measurement Conference*, 2004.

[15] Q. Zhao, J. Xu, and Z. Liu. Design of a novel statistics counter architecture with optimal space and time efficiency. In *ACM SIGMETRICS*, 2006.